Extending Dreamweaver MX

macromedia®

Acknowledgments

Project Management: Sheila McGinn

Writing: Robert Berry, David Jacowitz, Elisa Ma, and Jerry Pope

Editing: Mary Ferguson, Mary Kraemer, and Lisa Stanziano

Production Management: Patrice O'Neill

Multimedia Design and Production: Aaron Begley, Benjamin Salles, and Noah Zilberberg

Print and Help Design and Production: Caroline Branch and John Francis

Web Editing and Production: George Brown, Rebecca Godbois, Jeff Harmon, and Jon Varese

Special thanks to Winsha Chen, Jake Cockrell, George Comninos, Kristin Conradi, David Deming, Chris Denend, Randy Edmunds, Dave George, Nick Halbakken, Lori Hylan, Narciso (nj) Jaramillo, Craig Jennings, Ken Karleskint, Amit Kishnani, Sho Kuwamoto, David Lenoe, Jay London, Bonnie Loo, Sam Matthews, Susan Morrow, Masayo Noda, Jeff Schang, Sam Schillace, Mike Sundermeyer, Jorge Taylor, Venu Venugopal, Heidi Bauer Williams and the entire Dreamweaver engineering and QA teams.

First Edition: June 2002

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

Part I
Overview

CHAPTER 1
Introduction 11

Customizing or extending? 11
Installing an extension 11
Additional resources available to extension writers 12
Errata 12
Conventions used in this guide 12
What's new in Extending Dreamweaver MX 13

CHAPTER 2
Extending Dreamweaver MX 17

What makes extending possible 17
Application programming interfaces in Dreamweaver 17
Extension folders 18
Types of extension APIs in Dreamweaver 20
How Dreamweaver processes JavaScript in extensions 21
Working with the Extension Manager 22
Extensible document types in Dreamweaver 22

CHAPTER 3
User Interfaces for Extensions 31

Designing an extension UI 31
Dreamweaver HTML rendering control 32
Using custom UI controls in extensions 33

CHAPTER 4
The Dreamweaver Document Object Model 41

Which document DOM? 41
The Dreamweaver DOM 42

Part II
Extension APIs

CHAPTER 5
Objects 51

How object files work 51
Defining the Insert bar 52
Insert bar definition tags 52
Insert bar tag attributes 54
Adding Objects to the Insert menu 56
The Objects API 57

CHAPTER 6
Commands 61

How commands work 61
The Command API 62
Adding commands to the Commands menu 66

CHAPTER 7
Menu Commands 67

How menu commands work 67
The Menu Commands API 68

CHAPTER 8
Toolbars 77

How toolbars work 77
The toolbar definition file 79
Toolbar item tags 83
Item Tag Attributes 88
The Toolbar Command API 93

CHAPTER 9
Reports 103

How site reports work 103
How stand-alone reports work 104
The Reports API 104

CHAPTER 10
Tag Libraries and Editors 107

Tag Library file format 107
The Tag Chooser 111
Creating a new tag editor 113
Tag editor APIs 117

CHAPTER 11
Property Inspectors 119

How Property inspector files work 120
The Property inspector API 121

CHAPTER 12
Floating Panels 125
How floating panel files work 125
The Floating panel API 126

CHAPTER 13
Behaviors 135
How Behaviors work 135
The Behaviors API 136

CHAPTER 14
Server Behaviors 145
Dreamweaver architecture 146
How the Server Behavior API functions are called 149
The Server Behavior API 151
Server behavior implementation functions. 156
Editing EDML files. 158
Group EDML file tags 160
Participant EDML files 165
Using the Extension Data Manager 181
Server behavior techniques 183

CHAPTER 15
Data Sources 191
How data sources work 191
The Data Sources API 193

CHAPTER 16
Server Formats 199
How data formatting works. 200
When the data formatting functions are called 201
The Data Formatting API 202

CHAPTER 17
Components. 205
Component panel files 205
Component panel API functions 207

CHAPTER 18
Server Models 217
The Server Model API. 217

CHAPTER 19
Data Translators 225
How data translators work. 225
Determining what kind of translator to use 229
Adding a translated attribute to a tag. 229
Locking translated tags or blocks of code. 234
Finding bugs in your translator 241

CHAPTER 20
JavaScript Debugger Modules 243
How the JavaScript Debugger module works 243
The JavaScript Debugger module API 245

CHAPTER 21
C-Level Extensibility 251
C-level extensibility and the JavaScript interpreter 253
Data Types 253
The C-level API 254
File Access and Multiuser Configuration API 260
Calling a C function from JavaScript 267

Part III Utility APIs

CHAPTER 22
The File I/O API 271
Accessing configuration folders 271
The File I/O API 271

CHAPTER 23
The HTTP API 281
The HTTP API 281

CHAPTER 24
The Design Notes API 287
How Design Notes work 287
The Design Notes JavaScript API 288
The Design Notes C API 292

CHAPTER 25
The Fireworks Integration API 299

CHAPTER 26
The Flash Objects API 307

CHAPTER 27
The Database API 311
Database connection functions 312
Database access functions 324

CHAPTER 28
The Database Connectivity API 337
The Connection API 338
The generated include file 341
The definition file for your connection type 343

CHAPTER 29
The JavaBeans API 345

CHAPTER 30
The Source Control Integration API 349
Integration with Dreamweaver 349
Adding source control system functionality 350
The Source Control Integration API required functions 350
The Source Control Integration API optional functions 355
Enablers 363

Part IV
JavaScript API

CHAPTER 31
The Dreamweaver JavaScript API 371
Understanding the objects in the API 371
How this chapter is organized 372
About enablers 372
Assets panel functions 372
Behavior functions 380
Clipboard functions 388
Code hints functions 392
Command functions 400
Components functions 401
Conversion functions 402
CSS Styles functions 403
Data source functions 408
Enablers 409
External application functions 441
File manipulation functions 447
Find/replace functions 460
Frame and frameset functions 464
General editing functions 466
Global application functions 480
Global document functions 483
History functions 487
HTML style functions 495
JavaScript debugger functions 498
Keyboard functions 502
Layer and image map functions 508
Layout environment functions 511
Layout view functions 516
Library and template functions 521
Live data functions 526
Menu functions 530
Path functions 532
Print function 534
Quick Tag Editor Functions 534

Report Functions 536
Results window functions 537
Selection functions 546
Server behavior functions 551
Server model functions 552
Site functions 558
Snippets panel functions 582
String manipulation functions 585
Source view functions 588
Table editing functions 602
Tag editor and tag library functions 608
Tag inspector functions 612
Timeline functions 614
Toggle functions 620
Toolbar functions 637
Translation functions 640
Window functions 642

APPENDIX A
Deprecated JavaScript API functions 653

INDEX 661

Part I
Overview

Orient yourself to concepts fundamental to writing Dreamweaver extensions These concepts include understanding available API categories, creating new document types, working effectively with the Dreamweaver user interface, and understanding the Dreamweaver Document Object Model (DOM).

- Chapter 1, “Introduction”
- Chapter 2, “Extending Dreamweaver MX”
- Chapter 3, “User Interfaces for Extensions”
- Chapter 4, “The Dreamweaver Document Object Model”

CHAPTER 1
Introduction

This book contains descriptions of the tools that are available for developers to extend Dreamweaver using Dreamweaver application programming interfaces (APIs) and provides basic information about their use. It assumes that you are familiar with Dreamweaver, HTML and XML markup, and JavaScript programming. For users who are implementing C extensions, the book assumes that you know how to create and use C dynamic linked libraries (DLLs). If you are interested in writing extensions for building web applications, you should also be familiar with server-side scripting on at least one platform, such as Active Server Pages (ASP), ASP.net, PHP: Hypertext Preprocessor (PHP), ColdFusion, or Java Server Pages (JSP).

Customizing or extending?

Before you begin writing Macromedia Dreamweaver MX extensions, please review "Customizing Dreamweaver" on the Macromedia Support Center. In addition to procedures for changing Dreamweaver panels, menus, dialog boxes, and HTML formats, "Customizing Dreamweaver" contains information about the most common Dreamweaver extensions—how to edit Dreamweaver commands and how to add third-party tags. If you plan to create extensions that work with databases, you might also want to review the sections in *Getting Started with Dreamweaver MX* about making connections to databases.

Installing an extension

As you become familiar with the process of writing extensions, you might want to explore the extensions and resources that are available through the Macromedia Exchange website (http://www.macromedia.com/exchange/). Installing an existing extension introduces you to some of the tools that you need to use when working with your own extensions.

To install an extension, use the following procedure:

1. Download and install the Extension Manager, which is available on the Macromedia Downloads website (http://www.macromedia.com/software/downloads/).
2. Log on to the Macromedia Exchange website (http://www.macromedia.com/exchange).
3. From the available extensions, select one that you want to use. Click the download link to download the extension package.

4 Save the extension package in the Dreamweaver MX\Downloaded Extensions folder of your installed Dreamweaver folder.

5 In the Extension Manager, select File > Install Extension. In Dreamweaver, select Commands > Manage Extensions to launch the Extension Manager.

The Extension Manager automatically installs the extension from the Downloaded Extension folder into Dreamweaver.

Some extensions need Dreamweaver to restart before you can use them. If you are running Dreamweaver when you install the extension, you might be prompted to quit and restart the application.

To view basic information on the extension after its installation, go to the Extension Manager (Commands > Manage Extensions) in Dreamweaver.

Additional resources available to extension writers

To communicate with other developers who are involved in extension writing, you might want to join the Dreamweaver extensibility newsgroup. You can access the web site for this newsgroup at this URL: http://www.macromedia.com/go/extending_newsgrp/.

Errata

A current list of known issues can be found in the Extensibility section of the Dreamweaver Support Center http://www.macromedia.com/go/extending_errata.

Conventions used in this guide

The following typographical conventions are used in this guide:

- `Code` font indicates code fragments and API literals, including class names, method names, function names, type names, scripts, SQL statements, and both HTML and XML tag and attribute names.
- *`Italic code`* font indicates replaceable items in code.
- The continuation symbol (¬) indicates that a long line of code has been broken across two or more lines. Due to margin limits in this book's format, what is otherwise a continuous line of code must be split. When copying the lines of code, eliminate the continuation symbol and type the lines as one line.
- Curly braces ({ }) surrounding a function argument indicate that the argument is optional.

The following naming conventions are used in this guide:

- You—the developer who is responsible for writing extensions.
- The user—the person using Dreamweaver.
- The visitor—the person who views the web page that the user created.

What's new in Extending Dreamweaver MX

Dreamweaver MX includes the following new features and interfaces that you can access to develop extensions for the product:

- An enhanced user interface
- Multiple user configurations
- Enhanced code editing
- Expanded document type support
- Enhanced server model extensibility
- Improved database connection handling
- Enhanced external application integration

The following sections describe how you can use these features and interfaces to extend Dreamweaver MX.

Enhanced user interface

Toolbars

Dreamweaver MX adds support for extensible toolbars, which lets you customize the functionality of the existing document toolbars or add your own. The Toolbars API lets you control the functions of the various fields and buttons on a toolbar. See "Toolbars" on page 77, and "Toolbar functions" on page 637.

Tag Dialogs

Users can use Tag Dialogs to insert new tags, edit existing tags, access reference information about tags, and validate tags. Tag Dialogs reference Tag Libraries that come with Dreamweaver, which catalogs the tags that are used in different markup languages. Tag Dialogs can also be extended to work with customized implementations of markup and scripting languages. You can create custom Tag Libraries to contain custom Tags, create files for related reference information, and make fully functional sets of custom Tag Dialogs available to users. You can also customize how the Tag Chooser organizes tags for display. See "Tag Libraries and Editors" on page 107.

Multiple document interface mode

Dreamweaver MX introduces a new type of user interface, or workspace, known as the Dreamweaver MX workspace. The Dreamweaver MX workspace, also known as the multiple document interface (MDI), organizes many of the floating and overlapping Dreamweaver windows within a frame. In the Dreamweaver MX workspace, new functions let you cascade and tile document windows. See "Window functions" on page 642 and "The Floating panel API" on page 126.

You can also choose to continue operating in the Dreamweaver 4 workspace, which is known as classic mode.

Results windows

Dreamweaver MX implements a more traditional multiple-document results window that resides, by default, at the bottom part of the workspace. New functions let you browse, locate, select, save, cut, copy, and paste the contents of a results window. See "Results window functions" on page 537.

Importing/exporting sites

You can programmatically export a Dreamweaver MX site to an XML file, which can then be imported by any Dreamweaver instance on any computer. This feature lets users share sites and move them among host computers. For more information on importing and exporting sites, see "Site functions" on page 558.

Resource cloaking

In Dreamweaver MX, you can hide from view (cloak) selected files and folders. Cloaking selected site resources excludes them from site operations that Dreamweaver MX performs, which makes those operations more efficient. Dreamweaver also lets developers uncloak previously cloaked resources. You can programmatically cloak and uncloak files and folders by using functions described in "Site functions" on page 558.

File browsing

A new function has been added to the Site object, which lets you get the name of the site that is associated with a specific URL. For more information on this function, see "Site functions" on page 558.

Individual configurations

Multiple users

Dreamweaver MX supports multiple user configurations for the multiuser operating systems of Windows XP, Windows 2000, Windows NT, and Macintosh OS X. Users can customize Dreamweaver to best fit their needs without disturbing the customizations that other users have made on the same system. API functions let you create, remove, and access configuration files as well as access and set configuration attributes.

Enhanced code editing

Print code

Dreamweaver MX lets you print code in Code view from the File > Print menu and from the context menu. A new JavaScript function lets you open the print dialog box in Code view. See "Print function" on page 534.

Code Hints

When the user types a certain sequence in Code view (such as '<'), a pop-up menu shows a list of possible text entries, such as a list of tag names. You can select a Code Hint entry from the menu as a typing shortcut.

The CodeHints.xml file, and a set of XML tags, let you create new Code Hints menus. New JavaScript functions let you dynamically add menus and functions to a Code Hints menu. You can also pop up a Code Hints menu at the current location in Code view when Code Hints are not enabled. See "Code hints functions" on page 392.

Snippets panel

Dreamweaver MX users can edit and save reusable blocks of code in the new Snippets panel and retrieve them as needed. New JavaScript functions let you edit existing and create new snippets, organize them, and store them folders with user-friendly names.

New document types

Extensible document types

Dreamweaver MX lets you create new document types, including types that have file extensions that are identical to those of built-in Dreamweaver document types (such as .asp, which is associated with the default ASP-JS document type). You can define a new JavaScript function (`canRecognizeDocument()`) to help Dreamweaver determine which server model (when more than one server model claims a particular file extension) Dreamweaver should use to control a new document. For more information on this new function, see "The Server Model API" on page 217. For more information on extensible document types, see "Extensible document types in Dreamweaver" on page 22.

Also, a new property has been added to the document object model (DOM) to facilitate working with new document types.

XHTML document types

Using four new JavaScript functions, you can create a new or clean up an existing XHTML document, determine whether a document is an XHTML document, and convert an HTML document to XHTML. For more information on XHTML functions, see "File manipulation functions" on page 447.

Enhanced server model extensibility

Dreamweaver MX makes it easier to add new server models. A new JavaScript function (`serverModel.getServerIncludeUrlPatterns()`) lets the code that translates `include` file statements access the translator URL patterns. You can also add server-side set-up instructions for users. For more information on these new functions, see "Server model functions" on page 552.

Improved database connection handling

Implementing new connection types

Dreamweaver MX simplifies the process of defining new types of database connections and handling connections at runtime, as described in the following list:

- The following associations are now implicit within Dreamweaver MX: ASP sites use ADO connections, JSP sites use JDBC connections, and ColdFusion sites use ColdFusion data sources that are accessed through Remote Development Service (RDS).
- The Define Connection dialog box has been replaced by server-model-specific dialog boxes that are defined in the Extensibility layer.
- Connections are shared through use of an include file that contains the connection parameters, which automatically reflects changes to connection parameters everywhere the connection is used.

There are three new functions that you can define to implement a new connection type. Dreamweaver uses these functions to create a shared include file that defines the parameters that are needed to make a database connection. For more information on these new functions, see "The Connection API" on page 338.

Enhanced database exploration

Dreamweaver MX enhances database exploration in the following ways.

- For ColdFusion, you have the option of accessing the RDS server using ColdFusion data sources.
- When working with connection dialog boxes you can create new types of connections and let users duplicate or edit an existing connection.
- When you manage connections for a particular table, you can get a list of column objects, each of which holds the name and type of a column, or you can get a list of columns comprising the primary key. For a particular connection, you can get a list of procedure objects. For a particular procedure, you can get a list of parameter objects.

  You can also delete a connection.

There are 24 new functions supporting this new feature. Some of these functions handle database connections and the others handle database access. For more information on these new functions, see "The Database API" on page 311.

Enhanced external application integration

Flash Integration

Dreamweaver MX lets you determine whether Flash 6 is installed and, if it is, it gets the name of the Flash editor and its location so you can launch the Flash editor programmatically. For more information on these two new functions, see "External application functions" on page 441.

CHAPTER 2
Extending Dreamweaver MX

Most Dreamweaver extensions are written in HTML and JavaScript. Extensions typically perform the following types of tasks:

- Automating changes to the user's current document, such as inserting HTML, CFML, or JavaScript; changing text or image properties; or sorting tables
- Interacting with the application to automatically open or close windows, open or close documents, change keyboard shortcuts, and more
- Connecting to data sources, which lets Dreamweaver users create data-driven pages
- Inserting and managing blocks of server code in the current document

You might want to write an extension to handle a commonly used, and therefore repetitive, task, so this type of extension could be useful to many web developers. You might have a unique need that can be solved by writing an extension, which might be used only within a specific setting. In either case, Dreamweaver provides an extensive set of tools that you can use for adding to or customizing its functionality.

What makes extending possible

There are three main components to Dreamweaver extensibility:

- An HTML parser (also called a renderer), which makes it possible to design user interfaces for extensions using form fields, layers, images, and other HTML elements. Dreamweaver has its own HTML parser.
- A JavaScript interpreter, which executes the JavaScript code in extension files. Dreamweaver MX uses the Netscape Navigator JavaScript 1.5 interpreter. For more information about changes between this version of the interpreter and previous versions, see "How Dreamweaver processes JavaScript in extensions" on page 21.
- A series of APIs that provide access to Dreamweaver functionality through JavaScript.

Application programming interfaces in Dreamweaver

Three types of application programming interfaces (API)s are documented in *Extending Dreamweaver*:

- Extension APIs, which are discussed in the section, "Extending Dreamweaver MX" on page 17
- Utility APIs, which are discussed in the section, "Utility APIs" on page 269
- Dreamweaver JavaScript APIs, which are discussed in the section, "The Dreamweaver JavaScript API" on page 371

Extension APIs

The extension APIs provide the framework that you use to add functionality to Dreamweaver. You write the bodies of the functions as described in these APIs, and you specify the return values as required. After writing an extension, you must save it to the correct folder for it to work properly. The Extension Manager facilitates the process of saving extensions correctly.

Dreamweaver automatically calls any extension that exists in an appropriate Configuration folder when specified conditions are met. In most cases, this means that a user initiates a task, and then Dreamweaver identifies a related extension in the Configuration folder, calls the various functions in the extension, and expects a valid return value from each.

For developers who want to work directly in the C programming language, there is a C extensibility API that lets you create DLLs. The functionality that is provided in these APIs wraps your C DLLs in JavaScript so that your extension can work seamlessly within Dreamweaver.

The documentation of extension APIs outlines what each function does when it is called and what it is expected to return.

Utility APIs

The utility APIs provide functions that can assist you with specialized tasks. You should use the functions that are available within these APIs if your extension needs to do any of the following actions:

- Connect with databases
- Create Flash or Fireworks files
- Read and write files on disk
- Read and write Design Notes
- Get and send information to and from a remote web server using HTTP
- Work with JavaBeans

JavaScript API

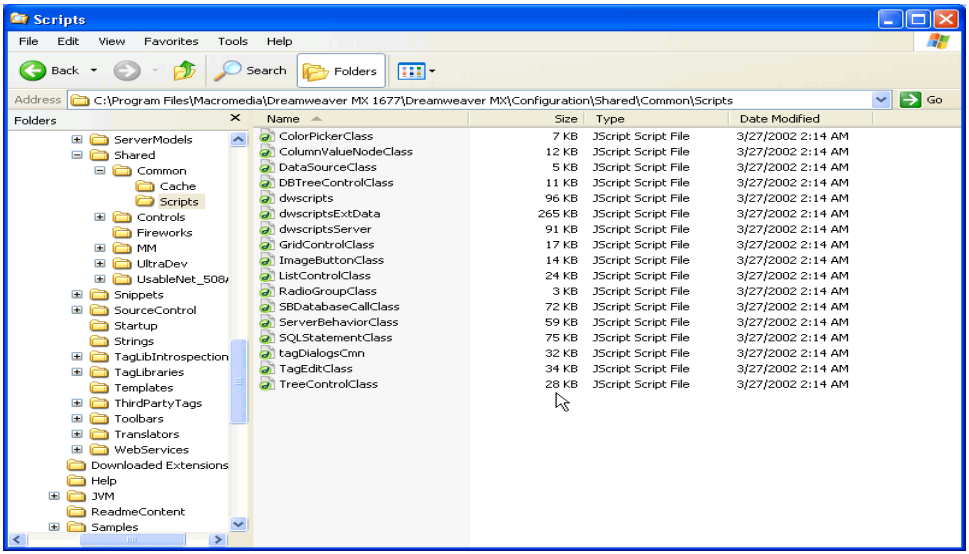
The JavaScript API provides JavaScript access to Dreamweaver. You can call any function that is available in this core JavaScript API from your extension, and Dreamweaver returns the appropriate value. In some cases, functions within this API make use of corollary functions that you write to determine the return value. For example, the `dom.serverModel.getDisplayName()` function in the core JavaScript API (“dom.serverModel.getDisplayName()” on page 553) calls and makes use of the value that the `getServerModelDisplayName()` function returns (“getServerModelDisplayName()” on page 222) that you write.

Extension folders

The folders and files that are stored in the Configuration folder contain the extensions that come with Dreamweaver. When you write an extension, you must save the files in the proper folder for Dreamweaver to recognize them. If you download and install an extension from the Macromedia Exchange website (www.macromedia.com/exchange), the Extension Manager automatically saves the extension files to the proper folders.

You can use the files that come with the product within the Configuration folder as examples, but these files are generally more complex than the average extension that is available on the Macromedia Exchange website. For more information on the contents of each subfolder within the Configuration folder, view the Configuration_ReadMe.htm file.

One folder within the Configuration folder does not correspond to a specific extension type. The Configuration/Shared folder is the central repository for utility functions, classes, and images that are used by more than one extension. The files in the Configuration/Shared/Common folder are designed to be useful to a broad range of extensions. Look here first for the functions that perform specific tasks, such as creating a valid DOM reference to an object, testing whether the current selection is inside a particular tag, escaping special characters in strings, and more. If you create common files, you should create a separate subfolder within the Configuration/Shared/Common folder.



Configuration/Shared/Common/Scripts file structure

Each file within the Configuration/Shared folder is fully commented. These files are useful both as examples of JavaScript techniques and as utilities.

Multiuser configuration folders

For the multiuser operating systems of Windows XP, Windows 2000, Windows NT, and Macintosh OS X, Dreamweaver MX creates a separate configuration folder for each user in addition to the Dreamweaver Configuration folder. Any time Dreamweaver MX or a JavaScript extension writes to the Configuration folder, Dreamweaver MX automatically writes to the user configuration folder instead. In this way, Dreamweaver MX lets each user customize the Dreamweaver MX configuration settings without disturbing the customized configurations of other users. See “File Access and Multiuser Configuration API” on page 260 for more information.

Types of extension APIs in Dreamweaver

The following list describes the types of extension APIs that are documented in this guide:

Object extensions create changes in the Insert bar. An object is typically used to automate the inserting code into a document. It can also contain a form that gathers input from the user and JavaScript that processes the input. Object files are stored in the Configuration/Objects folder.

Command extensions can perform almost any specific task, with or without input from the user. Command files are typically invoked from the menu system, but they can also be called from other extensions. Command files are stored in the Configuration/Commands folder.

Tag Dialog extensions work with Tag Dialog and the associated Tag Library files. Tag Dialog extensions can modify attributes of existing Tag Dialogs, create new Tag Dialogs, and add tags to the Tag Library. Tag dialog and tag library extension files are stored in the Configuration/TagLibraries folder.

Code Snippet extensions add new code snippets to the Snippets panel. You can create code snippets (CSN) files and install them into the Snippets directory so they appear in the Snippets panel. Code Snippets files are sorted in the Configuration/Snippets folder.

Code Hint extensions add new code hints for tags, objects, or script key words. Code hints provide information about `HTML`, `XML` and script tags that the user can view as they edit their documents. New code hints are incorporated into the *.vtm files that you create for new tags. Code hints extensions are stored in the Configuration/TagLibraries/servermodel folder.

Toolbar extensions can add menu items to existing toolbars or create new toolbars in the Dreamweaver user interface. Editing the toolbar is usually used to add new menu items to the site, browser, and code option pop-up menus. New toolbars appear below the default toolbar in the user interface. Toolbar files are stored in the Configuration/Toolbars folder.

Panel extensions add floating panels to the Dreamweaver user interface. Panels can interact with the selection, the document, or the task, or they can display useful information. Floating panel files are stored in the Configuration/Floaters folder.

Inspector extensions appear in the Property inspector panel. Most of the inspectors in Dreamweaver are part of the core product code and cannot be modified, but custom Property inspector files can override the built-in Dreamweaver Property inspector interfaces or create new ones to inspect custom tags. Inspectors are stored in the Configuration/Inspectors folder.

Behavior extensions let users add JavaScript code to their documents. The JavaScript code performs a specific task in response to an event when the document is viewed in a browser. Behavior extensions appear in the plus (+) menu in the Dreamweaver Behaviors panel. Behavior files are stored in the Configuration/Behaviors/Actions folder.

Server Behavior extensions add blocks of server-side code (ASP, JSP, or ColdFusion) to the document. The server-side code performs tasks on the server when the document is viewed in a browser. Server behaviors appear in the plus (+) menu in the Dreamweaver Server Behaviors panel. Server behavior files are stored in the Configuration/Server Behaviors folder.

Help Book extensions implement Integrated OS Help by installing a new compiled help (*.chm) file into the Help directory and adding a new `<book-id>` tag to the help.xml file. All help files are stored in the Dreamweaver MX/Help folder.

Data Translator extensions convert non-HTML code into HTML that appears in the Design view of the Document window. These extensions also lock the non-HTML code to prevent it from being parsed by Dreamweaver. Translator files are stored in the Configuration/Translators folder.

Data Source extensions let you build a connection to a custom data source. Data source extensions appear in the plus (+) menu of the Bindings panel. Data source files are stored in the Configuration/Data Sources folder.

Server Model extensions let you add support for new server models. Dreamweaver supports the most common server models (ASP, JSP, ColdFusion, PHP, and ASP.NET). Server model extensions are needed only for custom server solutions, different languages, or a customized server. Server model files are stored in the Configuration/ServerModels folder.

Document Type extensions define how Dreamweaver works with different document types. Information about document types for server models is stored in the Configuration/DocumentTypes folder.

How Dreamweaver processes JavaScript in extensions

Dreamweaver checks Configuration/Extensions during startup. If it encounters an extension file within the folder, Dreamweaver processes the JavaScript by completing the following steps:

- Compiling everything between the opening and closing `SCRIPT` tags.
- Executing any code within `SCRIPT` tags that is not part of a function declaration.

  Note: This procedure is necessary during startup because some extensions may require initialization of global variables.

For any external JavaScript files that are specified in the `SRC` attributes of `SCRIPT` tags, Dreamweaver performs the following actions:

- Reads in the file
- Compiles the code
- Executes the procedures

  Note: If any JavaScript code in your extension files contains the string `'</SCRIPT>'`, the JavaScript interpreter reads this as an actual closing `SCRIPT` tag and reports an unterminated string literal error. To avoid this problem, break the string into pieces and concatenate them, as shown in the following example: `'<' + '/SCRIPT>'`.

Dreamweaver executes code in the `onLoad` event handler (if one appears in the `BODY` tag) when the user chooses the command or action from a menu for the following extension types:

- Command
- Behavior action

Dreamweaver executes code in the `onLoad` event handler on the `BODY` tag if the body of the document contains a form for object extensions.

Dreamweaver ignores the `onLoad` handler on the `BODY` tag in the following extensions:

- Data translator
- Property inspector
- Floating panel

For all extensions, Dreamweaver executes code in other event handlers (for example, `onBlur="alert('This is a required field.')"`) when the user interacts with the form fields to which they are attached.

Dreamweaver MX supports the use of links within extensions. Event handlers in links must use syntax as shown in the following example:

```
<aref="#" onMouseDown="alert('hi')">link text</a>
```

Plug-ins (set to `play` at all times) are supported in the `BODY` of extensions. The `document.write()` statement, Java applets, and ActiveX controls are not supported in extensions.

Running scripts at startup or shutdown

If you place a command file in the Configuration/Startup folder, the command runs as Dreamweaver starts up. Startup commands load before the menus.xml file, before the files in the ThirdPartyTags folder, and before any other commands, objects, behaviors, inspectors, floating panels, or translators. You can use startup commands to modify the menus.xml file or other extension files. You can also show warnings, prompt the user for information, or call "dreamweaver.runCommand()" on page 400. However, from within the Startup folder, you cannot call a command that expects a valid DOM.

Similarly, if you place a command file in the Configuration/Shutdown folder, the command runs as Dreamweaver shuts down. From the shutdown commands, you can call "dreamweaver.runCommand()" on page 400, show warnings, or prompt the user for information, but you cannot stop the shutdown process.

For more information about commands, see "Commands" on page 61.

Working with the Extension Manager

If you are creating extensions for others users, you must package them according to the guidelines on the Macromedia Exchange website under Help > How to Create an Extension. After you have written and tested an extension in the Extension Manager, choose File > Package Extension. After the extension is packaged, you can submit it to the Exchange from the Extension Manager by choosing File > Submit Extension.

The Extension Manager comes with Dreamweaver MX. Details about its use are available in its Help files and on the Macromedia Exchange website.

Extensible document types in Dreamweaver

XML provides a rich system for defining complex documents and data structures. Dreamweaver MX uses several different XML schemas to organize information about server behaviors, tags and tag dialogs, components, document types, and reference information.

When you create and work with extensions in Dreamweaver, you find there are many instances in which you can create or modify existing XML files to manage the data that your extension uses. In many cases, you can copy an existing file from the appropriate subfolder within the Configuration folder to use as a template that you can change according to your needs.

Document type definition file

The central component of extensible document types is the document type definition file. There might be several definition files, all of which are located in the Configuration/DocumentTypes folder. Each definition file contains information about at least one document type. For each document type, essential information such as server model, color coding style, descriptions, and so forth, is described.

Note: Do not confuse Dreamweaver MX document type definition files with what are called DTDs in XML literature. Document type definition files in Dreamweaver MX contain a set of `<documenttype>` elements, each of which defines a predefined collection of tags and attributes that are associated with a document type. When it launches, Dreamweaver parses the document type definition files and creates an in-memory database of information regarding all defined document types.

Dreamweaver MX provides an initial document type definition file. This file, named MMDocumentTypes.xml, contains all Macromedia-provided document type definitions:

Document Type	Server Model	Internal Type	File Extensions	Previous Server Model
ASP.NET C#	ASP.NET-Csharp	Dynamic	`aspx, ascx`	
ASP.NET VB	ASP.NET-VB	Dynamic	`aspx, ascx`	
ASP JavaScript	ASP-JS	Dynamic	`asp`	
ASP VBScript	ASP-VB	Dynamic	`asp`	
ColdFusion	ColdFusion	Dynamic	`cfm, cfml`	UltraDev 4 ColdFusion
ColdFusion Component		Dynamic	`cfc`	
JSP	JSP	Dynamic	`jsp`	
PHP	PHP	Dynamic	`php, php3`	
Library Item		DWExtension	`lbi`	
ASP.NET C# Template		DWTemplate	`axcs.dwt`	
ASP.NET VB Template		DWTemplate	`axvb.dwt`	
ASP JavaScript Template		DWTemplate	`aspjs.dwt`	
ASP VBScript Template		DWTemplate	`aspvb.dwt`	
ColdFusion Template		DWTemplate	`cfm.dwt`	
HTML Template		DWTemplate	`dwt`	
JSP Template		DWTemplate	`jsp.dwt`	
PHP Template		DWTemplate	`php.dwt`	
HTML		HTML	`htm, html`	
ActionScript		Text	`as`	
CSharp		Text	`cs`	
CSS		Text	`css`	
Java		Text	`java`	
JavaScript		Text	`js`	
VB		Text	`vb`	
VBScript		Text	`vbs`	

Document Type	Server Model	Internal Type	File Extensions	Previous Server Model
Text		Text	txt	
EDML		XML	edml	
TLD		XML	tld	
VTML		XML	vtm, vtml	
WML		XML	wml	
XML		XML	xml	

If you need to create a new document type, you can either add your entry to the document definition file that Macromedia provides (MMDocumentTypes.xml) or add your own definition file to the Configuration/DocumentTypes folder.

Note: The NewDocuments subfolder resides in the Configuration/DocumentTypes folder. This subfolder contains default pages (templates) for each document type.

Structure of document type definition files

The following example shows what a typical document type definition file might look like:

```
<?xml version="1.0" encoding="utf-8"?>
<documenttypes
      xmlns:MMString="http://www.macromedia.com/schemes/data/string/">
  <documenttype
    id="dt-ASP-JS"
    servermodel="ASP-JS"
    internaltype="Dynamic"
    winfileextension="asp,htm, html"
    macfileextension=asp, html"
    previewfile="default_aspjs_preview.htm"
    file="default_aspjs.htm"
    priorversionservermodel="UD4-ASP-JS" >
    <title>
      <loadString id="mmdocumenttypes_0title" />
    </title>
    <description>
      <loadString id="mmdocumenttypes_0descr" />
    </description>
  </documenttype>
  ...
</documenttypes>
```

Note: Color coding for document types are specified in the XML files that reside in the Configuration/CodeColoring folder.

In the preceding example, the `<loadstring>` element identifies the localized strings that Dreamweaver MX should use for the title and description for ASP-JS type documents. For more information on localized strings, see “Localized strings” on page 28.

The following table describes the tags and attributes that you can use within a document type definition file.

Element Type			
Tag	**Attribute**	**Required**	**Description**
`documenttype` *(root)*		Yes	Parent node
	`id`	Yes	Unique identifier across all document type definition files.

	`servermodel`	No	Specify the associated server model (case sensitive); by default, these are the valid values:
			`ASP.NET C#`
			`ASP.NET VB`
			`ASP VBScript`
			`ASP JavaScript`
			`ColdFusion`
			`JSP`
			`PHP MySQL`
			These names are the names returned by a call to the `getServerModelDisplayName()` functions that are defined in the server model implementation files, which are located in the Configuration/ServerModels folder.
			Extension developers can create new server models, which would extend this list.

	`internaltype`	Yes	A broad classification of how a file is treated in Dreamweaver. The internal type identifies whether the Design view is enabled for this document and handles special cases such as Dreamweaver Templates or Extensions.
			Valid values are:
			`Dynamic`
			`DWExtension` (has special display regions)
			`DWTemplate` (has special display regions)
			`HTML`
			`HTML4`
			`Text` (Code view only)
			`XHTML1`
			`XML` (Code view only)
			All server model-related document types should map to `Dynamic`. HTML should map to `HTML`. Script files such as .css, .js, .vb, and .cs should map to `Text`.
			If `internaltype` is `DWTemplate`, you should also specify `dynamicid`. If you omit `dynamicid` in this case, the new blank template that is created from the New Document dialog box does not have its document type recognized by the Server Behavior or Bindings panel. Instances of this template do not know they are supposed to be anything besides HTML.

	`dynamicid`	No	A reference to the unique identifier of a dynamic document type. This attribute is meaningful only when `internaltype` is `DWTemplate`. This attribute lets you associate a dynamic template with a dynamic document type.

Element Type			
Tag	**Attribute**	**Required**	**Description**
	`winfileextension`	Yes	The file extension that is associated with the document type on Windows. You specify multiple file extensions by using a comma-separated list. The first extension in the list is the extension that Dreamweaver MX uses when the user saves a document of type `documenttype`. If two nonserver model-associated document types have the same file extension, Dreamweaver recognizes the first one as the document type for the extension.
	`macfileextension`	Yes	The file extension that is associated with the document type on the Macintosh. You specify multiple file extensions by using a comma-separated list. The first extension in the list is the extension Dreamweaver MX uses when the user saves a document of type `documenttype`. If two nonserver model-associated document types have the same file extension, Dreamweaver recognizes the first one as the document type for the extension.
	`previewfile`	No	The file that is rendered in the Preview area of the New Document dialog box.
	`file`	Yes	The file located in the DocumentTypes/NewDocuments folder that contains template content for new documents of type `documenttype`.
	`priorversionservermodel`	No	If this document's server model has a Dreamweaver UltraDev 4 equivalent, specify the name of the older version of the server model. UltraDev 4 ColdFusion is a valid prior server model.
`title` *(subtag)*		Yes	The string that appears as a category item under Blank Document in the New Document dialog box. You can place this string directly in the definition file or point to it indirectly for localization purposes. For more information on localizing this string, see "Localized strings" on page 28. Formatting is not allowed, so HTML tags cannot be specified.
`description` *(subtag)*		No	The string that describes the document type. You can place this string directly in the definition file or point to it indirectly for localization purposes. For more information on localizing this string, see "Localized strings" on page 28. Formatting is allowed, so HTML tags can be specified.

Note: When the user saves a new document, Dreamweaver MX examines the list of extensions for the current platform that are associated with the document type (`winfileextension` and `macfileextension`). Dreamweaver selects the first string in the list and uses it as the default file extension. To change this default file extension, you need to reorder the extensions in the comma-separated list so that the new default is listed first.

When Dreamweaver MX launches, it reads all document type definition files and builds a list of valid document types. Dreamweaver treats any entries within the definition files that have nonexistent server models as nonserver model document types. Dreamweaver ignores entries that have bad contents or IDs that are not unique.

If, while scanning the Configuration/DocumentTypes folder, Dreamweaver MX finds no document type definition files or if any of the definition files appear to be corrupt, Dreamweaver closes with an error message.

Dynamic templates

You can create templates that are based on dynamic document types. These templates are called *dynamic templates*. The following two elements are essential to defining a dynamic template:

- The value of the `internaltype` attribute for the new document type must be `DWTemplate`.
- The `dynamicid` attribute must be set and the value must be a reference to the identifier of an existing dynamic document type.

If, for example, you have defined the following dynamic document type:

```
<documenttype
    id="PHP_MySQL"
    servermodel="PHP MySQL"
    internaltype="Dynamic"
    winfileextension="php,php3"
    macfileextension="php,php3"
    file="Default.php"
>
    <title>PHP</title>
    <description><![CDATA[PHP document]]></description>
</documenttype>
```

You can then define the following dynamic template, which is based on this `PHP_MySQL` dynamic document type:

```
<documenttype
    id="DWTemplate_PHP"
    internaltype="DWTemplate"
    dynamicid="PHP_MySQL"
    winfileextension="php.dwt"
    macfileextension="php.dwt"
    file="Default.php.dwt"
>
    <title>PHP Template</title>
    <description><![CDATA[Dreamweaver PHP Template document]]></description>
</documenttype>
```

When a Dreamweaver MX user creates a new blank template of type `DWTemplate_PHP`, Dreamweaver lets the user create PHP server behaviors in the file. Furthermore, when the user creates instances of the new template, the user can create PHP server behaviors in the instance.

In the previous example, when the user saves the template, Dreamweaver MX automatically adds a .php.dwt extension to the file. When the user saves an instance of the template, Dreamweaver adds the .php extension to the file.

Document extensions

After creating a new document type, extension developers need to update the appropriate Extensions.txt file. If the user is on a system that supports multiple users (such as Windows XP, Windows 2000, or Mac OS X), the user has another Extensions.txt file in their Configuration folder. This Extensions.txt file is the one that the user needs to update because this file is the instance that Dreamweaver looks for and parses.

The location of the user's Configuration folder depends on the user's platform.

For Windows 2000 and Windows XP platforms:

```
<drive>:\Documents and Settings\<username>\ ¬
  Application Data\Macromedia\Dreamweaver MX\Configuration
```

For Windows NT platforms:

```
<drive>:\WinNT\profiles\<username>\ ¬
  Application Data\Macromedia\Dreamweaver MX\Configuration
```

For Mac OS X platforms:

```
<drive>:Users:<username>:Library:Application Support: ¬
  Macromedia:Dreamweaver MX:Configuration
```

If Dreamweaver MX cannot find Extensions.txt in the user's Configuration folder, Dreamweaver looks for it in the Dreamweaver Configuration folder.

Note: On multiuser platforms, if you edit the copy of Extensions.txt that resides in the Dreamweaver Configuration folder and not the one located in the user's Configuration folder, Dreamweaver is not aware of the changes because Dreamweaver parses the copy of Extensions.txt in the user's Configuration folder, not in the Dreamweaver Configuration folder.

Sometimes you might want to create a new document extension. To create a new document extension, you can either add the new extension to an existing document type or create a new document type, which is explained in preceding paragraphs.

To add a new extension to an existing document type, perform the following steps:

1. Edit MMDocumentTypes.xml.
2. Add the new extension to the `winfileextension` and `macfileextension` attributes of the existing document type.
3. Add the new extension to the appropriate Extensions.txt file, as described at the beginning of this section. Suppose you have a new document type called FOO and that it has three file extensions that are associated with it: FE, FI, and FO. The following example shows how to add those extensions to the Extensions.txt file:

```
HTM,HTML,...,VTML,FE,FI,FO:All Documents
...
FE,FI,FO:FOO Files
```

Localized strings

Within a document type definition file, the `<title>` and `<description>` subtags specify the display title and description for the document type. You can use the `MMString:loadstring` directive in the subtags as a placeholder for providing localized strings for the two subtags. This process is similar to server-side scripting where you specify a particular string to use in your page by using a string identifier as a placeholder. For the placeholder, you can use a special tag or you can specify a tag attribute whose value is replaced.

To provide localized strings, perform the following steps:

1 Place the following statement at the top of the document type definition file:

```
<?xml version="1.0" encoding="utf-8"?>
```

2 Declare the `MMString` name space in the `<documenttypes>` tag:

```
<documenttypes
    xmlns:MMString="http://www.macromedia.com/schemes/data/string/">
```

3 At the location in the document type definition file where you want to provide a localized string, use the `MMString:loadstring` directive to define a placeholder for the localized string. You can specify this placeholder in one of two ways:

```
<description>
   <loadstring>myJSPDocType/Description</loadstring>
</description>
```

or

```
<description>
   <loadstring id="myJSPDocType/Description" />
</description>
```

In these examples, `myJSPDocType/Description` is a unique string identifier that acts as a placeholder for the localized string. The localized string is defined in the next step.

4 In the Configuration/Strings folder, create a new XML file (or edit an existing file) that defines the localized string. For example, the following code, when placed in the Configuration/Strings/strings.xml file, defines the `myJSPDocType/Description` string:

```
<strings>
...
   <string id="myJSPDocType/Description"
        value=
        "<![CDATA[JavaServer Page with <em>special</em> features]]>"
   />
...
</strings>
```

Note: String identifiers, such as `myJSPDocType/Description` in the preceding example, must be unique within the Dreamweaver MX application. Dreamweaver, when it launches, parses all XML files within the Configuration/Strings folder and loads these unique strings.

Rules for document type definition files

Dreamweaver MX lets document types that are associated with a server model share file extensions. For example: ASP-JS and ASP-VB can claim .asp as their file extension. (For information on which server model gets preference, see “canRecognizeDocument()” on page 217.)

Dreamweaver MX does not let document types that are not associated with a server model share file extensions.

If a file extension is claimed by two document types where one type is associated with a server model and the other is not, the latter document type gets preference. Suppose you have a document type called SAM, which is not associated with a server model, that has a file extension of .sam, and you add this file extension to the ASP-JS document type. When a Dreamweaver MX user opens a file that has a .sam extension, Dreamweaver assigns the SAM document type to it, not ASP-JS.

Opening a document in Dreamweaver

When a user opens a file, Dreamweaver MX follows a series of steps to identify the document type based on the file's extension.

If Dreamweaver successfully finds a unique document type, Dreamweaver uses that type and loads the associated server model (if any) for the document that the user is opening. If the user has selected to use Dreamweaver UltraDev 4 server behaviors, Dreamweaver MX loads the appropriate UltraDev 4 server model.

If the file extension maps to more than one document type, Dreamweaver performs the following actions:

- If a static document type is among the list of document types, it gets preference.
- If all the document types are dynamic, Dreamweaver MX creates an alphabetical list of the server models that are associated with these document types and then calls the `canRecognizeDocument()` function in each server model (see "canRecognizeDocument()" on page 217). Dreamweaver collects the return values and determines which server model returned the highest valued positive integer. The document type whose server model returns the highest integer is the document type that Dreamweaver assigns to the document being opened. If, however, more than one server model returns the same integer, Dreamweaver goes through the alphabetical list of those server models, picks the first in the list, and uses that document type. For example, if both ASP-JS and ASP-VB claim an .asp document and if their respective `canRecognizeDocument()` functions return equal values, Dreamweaver assigns the document to ASP-JS (because, alphabetically, ASP-JS is first).

If Dreamweaver MX cannot map the file extension to a document type, Dreamweaver opens the document as a text file.

CHAPTER 3
User Interfaces for Extensions

Most extensions are built to receive information from the user through a user interface (UI). If you plan to submit your extension for Macromedia certification, you need to follow the guidelines that are available within the Extension Manager files (http://www.macromedia.com/exchange/). These guidelines are not intended to limit your creativity; their purpose is to ensure that certified extensions work effectively within the Dreamweaver UI, and that the extension UI design does not detract from its functionality.

Designing an extension UI

Typically, an extension is built to perform a task that a set of users encounters frequently. Certain parts of the task are repetitive and, therefore, can be automated. Some steps in the task can change, or specific attributes of the code that the extension processes can change. You build the UI to handle user inputs for these variable values.

As an example, an extension could automate updates for a web catalog where users need to change values for image sources, item descriptions, and prices periodically, but the procedures for taking these values and formatting the information for display on the website remains the same. A simple extension can automate the formatting while letting users manually input the new, updated values for the three variables. A more advanced extension could automate the process of pulling a set of values for image sources, item descriptions, and prices directly from a database, with variables for time intervals input by the user.

So the purpose of your extension UI is to receive the user inputs that are needed to handle the variable aspects of a repetitive task that the extension performs. Dreamweaver supports HTML and JavaScript form elements as the basic building blocks for creating extension UI controls and displays the UI using its own HTML renderer. Therefore, an extension UI can be as simple as an HTML file that contains a two-column table with text descriptions and form input fields.

Most extension developers design their extension UI after coding most of the functionality of their extension in JavaScript. After you begin writing code, it is often easy to discern what variables are necessary and what form inputs can best handle them.

Consider the following basic guidelines as you design an extension UI:

- If you want a name for your extension, place the name in the Title Tag of your HTML file. Dreamweaver displays the name in the Extension title bar.
- Keep text labels on the left side of your UI, aligned right, with text boxes on the right side, aligned left. This arrangement lets the user's eyes easily locate the beginning of any text box. Minimal text can follow the text box as explanation or units of measure.
- Keep checkbox and radio button labels on the right side of your UI, aligned left.
- For readable code, assign logical names to your text boxes. If you use Dreamweaver to create your extension UI, you can use the Property inspector or the Quick tag editor to assign names to the fields.

In a typical scenario, after you create the UI, test the extension code to see that it properly performs the following UI-related tasks:

- Getting the values from the text boxes
- Setting default values for the text boxes or gathering values from the selection
- Applying changes to the user document

Dreamweaver HTML rendering control

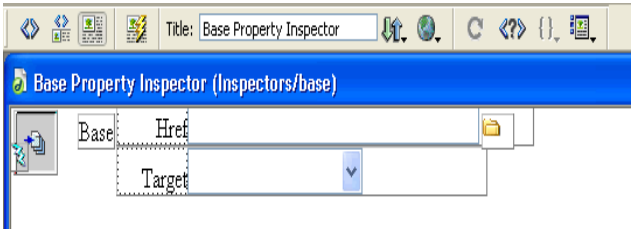
For versions through Dreamweaver 4, Dreamweaver rendered more space around form controls than do Microsoft Internet Explorer and Netscape Navigator. This means that form controls in extension UIs are rendered with extra space around them, because Dreamweaver uses its HTML rendering engine to display extension UIs.

Form control rendering has been improved in Macromedia Dreamweaver MX to more closely match the browsers. You can see the difference when you create documents that contain forms in Dreamweaver. However, to prevent extension developers from having to update existing extensions, form controls in extensions render the same way as they did in Dreamweaver 4. To take advantage of the rendering improvements, you must use one of three new `DOCTYPE` statements in your extension files, as shown in the following example:

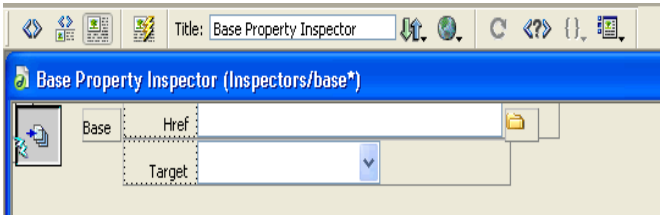
```
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine 5.0//dialog">

<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine5.0//floater">

<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine5.0//pi">
```

In most cases, `DOCTYPE` statements must go on the first line of a document. However, to avoid conflicts with extension-specific directives that, in previous versions, were required to be on the first line of a file (such as the comment at the top of a Property inspector file, or the `MENU-LOCATION=NONE` directive in a command), in Dreamweaver MX `DOCTYPE` statements and directives can be in any order as long as they appear before the opening `html` tag.

In addition to letting you make extension UIs more closely match the built-in dialog boxes and panels, the new DOCTYPE statements also let you view your extensions in the Dreamweaver Design view as they appear when viewed by users.



The Base Property inspector as it appears in Design view without the DOCTYPE statement.



The Base Property inspector as it appears in Design view with the DOCTYPE statement (and after a few adjustments to accommodate the new rendering).

Using custom UI controls in extensions

In addition to the standard HTML form elements, Dreamweaver supports custom controls to help you create flexible, professional-looking interfaces, as described in the following list:

- Editable select lists (also known as combo boxes) that let you combine the functionality of a select list with that of a text box
- Database controls that facilitate the display of data hierarchies and fields
- Tree controls that organize information into expandable and collapsible nodes
- Color button controls that let you add color picker interfaces to your extensions

Editable select lists

Extension UIs often contain pop-up lists that are defined using the <select> tag. In Dreamweaver, pop-up lists in extensions can be made editable by adding editable="true" to the <select> tag. To set a default value, set the editText attribute and the value that you want the select list to display.

The following example illustrates the settings for an editable select list:

```
<select name="travelOptions" style="width:250px" editable="true"
   editText="other (please specify)">
<option value="plane">plane</option>
<option value="car">car</option>
<option value=""bus">bus</option>
</select>
```

When you use select lists in your extensions, you can check for the presence and value of the editable attribute. If no value is present, the select list returns the default value of `false`, which indicates that the select list is not editable.

As with normal (noneditable) select lists, editable select lists have a `selectedIndex` property (see "Objects, properties, and methods of the Dreamweaver DOM" on page 42). This property returns -1 if the text box is selected.

To read the value of an active editable text box into an extension, read the value of the `editText` property. `editText` returns the string that the user entered into the editable text box, the value of the `editText` attribute, or an empty string if no text has been entered and no value has been specified for `editText`.

Dreamweaver adds the following custom attributes for `<select>` to control editable pop-up lists:

Attribute Name	Description	Accepted Values
`editable`	Declares that the pop-up list has an editable text area	Boolean value of `true` or `false`
`editText`	Holds or sets text within the editable text area	A string of any value

Note: Editable select lists are available in Dreamweaver MX.

The following example creates a command that contains an editable select list using common JavaScript functions:

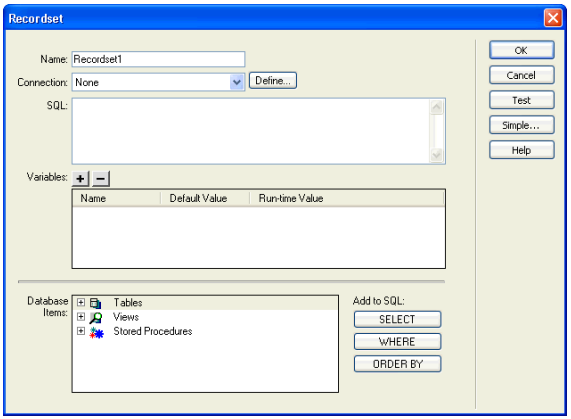
```
<html>
<head>
  <title>Editable Dropdown Test</title>
  <script language="javascript">
  function getAlert()
  {
    var i=document.myForm.mySelect.selectedIndex;
    alert ("selectedIndex: " + i);
    if (i>=0)
      alert("selected text " +document.myForm.mySelect.options[i].text);
    else
      alert("selected text " + document.myForm.mySelect.editText);
    else
      alert("nothing is selected");
  }
  function commandButtons()
  {
    return new Array("OK", "getAlert", "Cancel", "window.close()");
  }

  </script>
</head>

<body>
<div name="test">
<form name="myForm">
<table>
<tr> <td>button to click:</td><td>
<input type="button" value="button 1" onclick="getAlert();"></td>
</tr>
<tr>
<td>Editable DropDown with default text:</td>
<td><select name="mySelect" editable="true" style="width:150px"
  editText="Editable Text">
<option> opt 1 </option>
<option> opt 2 </option>
<option> opt 3 </option>
</select></td></tr>
<tr> <td>Editable DropDown without default text:</td>
<td><select name="mySelect_no" editable="true" style="width:150px">
<option value="1"> opt 1 </option>
<option value="2"> opt 2 </option>
<option value="3"> opt 3 </option>
</select></td></tr>
</table>

</form>
</div>

</body>
</html>
```

To use this sample, save it to the Dreamweaver Configuration/Commands folder as EditableSelectTest.htm. Restart Dreamweaver, and select EditableSelectTest from the Command menu.

Database controls

Using Dreamweaver, you can extend the HTML `<select>` tag to create a database tree control. You can also add a variable grid control. The database control is useful for displaying database schema. The variable grid control displays tabular information.

The following illustration shows an advanced Recordset dialog box that uses a database control and a variable grid control:



Adding a database tree control

The database control has the following attributes:

Attribute Name	Description
name	Name of the database control
control.style	Width and height, in pixels
type	Type of control
`connection`	Name of the database connection that is defined in the Connection Manager; if empty, the control is empty.
`noexpandbuttons`	When this attribute is specified, the tree control does not draw the plus (+) or collapse minus (-) indicators or the associated triangle arrows on the Macintosh. This attribute is useful for drawing multicolumn list controls.
`showheaders`	When this attribute is specified, the tree control displays a header at the top that lists the name of each column.

Any option tags that are placed inside the `<select>` tag are ignored.

To add a database tree control to a dialog box, you can use the following sample code with appropriate substitutions:

```
<select name="DBTree" style="width:400px;height:110px" ¬
type="mmdatabasetree" connection="connectionName" noexpandbuttons
  showHeaders></select>
```

You can change the `connection` attribute to retrieve selected data and display it in the tree. You can use `DBTreeControl` as a JavaScript wrapper object for the new tag. For more examples, see the DBTreeControlClass.js file in the Configuration\Shared\Scripts folder.

Adding a variable grid control

The variable grid control has the following attributes:

Attribute Name	Description
`name`	Name of the variable grid control
`style`	Width of the control, in pixels
`type`	Type of control
`columns`	Each column must have a name, separated by a comma
`columnWidth`	Width of each column, each separated by a comma. If no widths are specified, the columns are of equal width.

The following example adds a simple variable grid control to a dialog box:

```
<select name="ParamList" style="width:515px;" ¬
type="mmparameterlist columns"="Name,SQL Data ¬
Type,Direction,Default Value,Run-time Value" size=6></select>
```

The following example creates a variable grid control that is 500 pixels wide, with five columns of various widths:

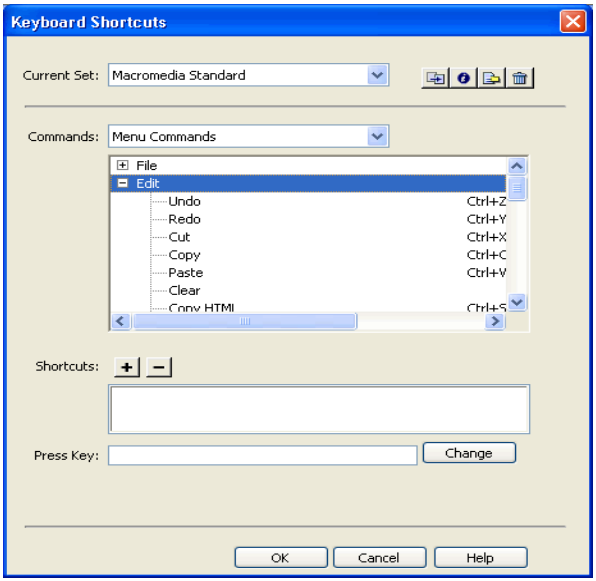
```
<select name="ParamList" style="width:500px;" ¬
type=mmparameterlist columns="Name,SQL Data Type,Direction, ¬
Default Value,Run-time Value" columnWidth="100,25,11," size=6>¬
</select>
```

This example creates two blank columns that are 182 pixels wide. (The specified columns total 136. The total width of the control is 500. The remaining space after the first three columns have been placed is 364. There are two columns left; 364 divided by 2 is 182.)

This grid control also has a JavaScript wrapper object that should be used to access and manipulate the grid control's data. You can find the implementation within the GridControlClass.js file in the Configuration\Shared\MM\Scripts\Class folder.

Adding tree controls

Tree controls display data in a hierarchical format and let users expand and collapse nodes in the tree. The `mm:treecontrol` tag lets you create tree controls for any type of information; unlike the database tree control described in “Adding a database tree control” on page 36, no association with a database is required.The Dreamweaver Keyboard Shortcuts editor uses the tree control, as shown in the following illustration:



Creating a tree control

The `MM:TREECONTROL` tag creates a tree control and can use one or more additional tags to add structure, as described in the following list:

- `MM:TREECOLUMN` is an empty, optional tag that defines a column in the tree control.
- `MM:TREENODE` is an optional tag that defines a node in the tree. It is a nonempty tag that can contain only other `MM:TREENODE` tags.

`MM:TREECONTROL` tags have the following attributes:

Attribute Name	Description
name	Name of the tree control
size	Optional. Number of rows that show in the control; default is 5 rows
theControl	Optional. If the number of nodes in theControl exceeds the value of the size attribute, scrollbars appear
multiple	Optional. Allows multiple selections; default is single-selection
style	Optional. Style definition for height and width of tree control; if specified, takes precedence over `SIZE` attribute
noheaders	Optional. Specifies that the column headers should not display

MM:TREECOLUMN tags have the following attributes:

Attribute Name	Description
name	Name of the column
value	String to appear in column header
width	Width of the column in pixels (percentage not supported); default is 100
align	Optional. Specifies whether the text in the column should be aligned left, right, or center; default is left
state	Specifies whether the column is visible or hidden

For readability, TREECOLUMN tags should follow immediately after the MM:TreeControl tag, as shown in the following example:

```
<MM:TREECONTROL name="tree1">
<MM:TREECOLUMN name="Column1" width="100" state="visible">
<MM:TREECOLUMN name="Column2" width="80" state="visible">
...
</MM:TREECONTROL>
```

The MM:TREENODE attributes are described in the following table:

Attribute Name	Description
name	Name of the node
value	Contains the content for the given node. For more than one column, this is a pipe-delimited string. To specify an empty column, place a single space character before the pipe (|).
expanded	An empty attribute that specifies the node is expanded by default
selected	You can select multiple nodes by setting this attribute on more than one tree node, if the tree has a MULTIPLE attribute.
icon	Optional. The index of built-in icon to use, starting with 0, as follows: 0 = no icon 1 = DW document icon 2 = Multidocument icon

The following example creates a tree control:

```
<mm:treecontrol name="CtrlName" [size=N] [style="[width:#px];[height:#px]"]>
<mm:treecolumn name="Column1" value="Items">
<mm:treenode value="Item1" selected></mm:treenode>
<mm:treenode value="Item2|Item3" expanded></mm:treenode>
<mm:treenode value="Item4|Item5"></mm:treenode>
</mm:treecolumn>
</mm:treenode>
</mm:treecontrol>
```

Manipulating content within a tree control

Tree controls and the nodes within them are implemented as HTML tags. They are parsed by Dreamweaver and stored in the document tree. These tags can be manipulated in the same way as any other document node. For more information on dom functions and methods, see "The Dreamweaver Document Object Model" on page 41.

Adding nodes To add a node to an existing tree control programmatically, set the `innerHTML` property of the `mm:treecontrol` tag or one of the existing `mm:treenode` tags. Setting the `innerHTML` property of a tree node creates a nested node.

The following example adds a node to the top level of a tree:

```
var tree = document.myTreeControl;
//add a top-level node to the bottom of the tree
tree.innerHTML = tree.innerHTML + '<mm:treenode name="node3"¬
  value="node3">';
```

Adding a child node To add a child node to the currently selected node set the `innerHTML` property of the selected node.

The following example adds a child node to the currently selected node:

```
var tree = document.myTreeControl;
var selNode = tree.selectedNodes[0];
//deselect the node, so we can select the new one
selnode.removeAttribute("selected");
//add the new node to the top of the selected node's children
selNode.innerHTML = '<mm:treenode name="item10" value="New item11" ¬
  expanded selected>' + selNode.innerHTML;
```

Deleting nodes To delete the currently selected node from the document structure, use the `innerHTML` or `outerHTML` properties.

The following example deletes the entire selected node and any children:

```
var tree = document.myTreeControl;
var selNode = tree.selectedNodes[0];
selNode.outerHTML = "";
```

A color button control for extensions

In addition to the standard input types such as text, checkbox, and button, Dreamweaver supports `mmcolorbutton`, an additional input type in extensions.

To cause a color picker to appear in the UI, specify `<input type="mmcolorbutton">` in your extension. You can set the default color for the color picker by setting a value attribute on the input tag. If no value is set, the color picker appears grey by default and the value property of the input object returns an empty string.

The following example shows a valid `mmcolorbutton` tag:

```
<input type="mmcolorbutton" name="colorbutton" value="#FF0000">
<input type="mmcolorbutton" name="colorbutton" value="teal">
```

A color button has one event, `onChange`, which is triggered when the color is changed.

You might want to keep a text box and a color picker synchronized. The following example creates a text box that synchronizes the color of the text box with the color of the color picker:

```
<input type = "mmcolorbutton" name="fgcolorPicker"
  onChange="document.fgcolorText.value=this.value">
<input type = "test" name="fgcolorText"
  onBlur="document.fgColorPicker.value=this.value">
```

In this example, when the user changes the value of the text box and then tabs or clicks elsewhere, the color picker updates to show the color that is specified in the text box. Whenever the user chooses a new color with the color picker, the text box updates to show the hex value for that color.

CHAPTER 4
The Dreamweaver Document Object Model

In Dreamweaver, the Document Object Model (DOM) is a critically important tool for extension builders. It is used to gain access to and manipulate elements within the user's document and within the extension file. For this reason, understanding the Dreamweaver DOM is important to extension developers.

A DOM defines the structure of documents that are created using a markup language. By representing tags and attributes as objects and properties, the DOM provides a way for documents and their components to be accessed and manipulated by programming languages.

The structure of an HTML document can be seen as a document tree. The root is the `HTML` tag, and the two largest trunks are `HEAD` and `BODY`. Offshoots of `HEAD` include `TITLE`, `STYLE`, `SCRIPT`, `ISINDEX`, `BASE`, `META`, and `LINK`, and offshoots of `BODY` include headings (`H1`, `H2`, and so on), block-level elements (`P`, `DIV`, `FORM`, and so on), text-level elements, (`FONT`, `BR`, `IMG`, etc.) and other element types. Leaves on these offshoots include attributes such as `WIDTH`, `HEIGHT`, `ALT`, and others.

In a DOM, the tree structure is preserved and presented as a hierarchy of parent nodes and child nodes. The root node has no parent, and leaf nodes have no children. At each level within the HTML structure, the HTML element can be exposed to JavaScript as a node. Using this structure, you can access the document or any element within it.

In JavaScript, you can call any document object by name or by index, as described in the following list:

- By name, as in `document.myForm.myButton`
- By index, as in `document.forms[0].elements[1]`

Objects with the same name are collapsed into an array. You can access a particular object in the array by incrementing the index with zero as the origin (for example, the first radio button with the name `myRadioGroup` in `myForm` is referenced as `document.myForm.myRadioGroup[0]`).

Which document DOM?

It is important to distinguish between the DOM of the user's document and the DOM of the extension. The information in this chapter applies to both types of Dreamweaver documents, but the way that you reference each DOM is different.

If you are familiar with JavaScript in browsers, you can reference objects in the active document by writing `document`. (for example, `document.forms[0]`), the same way that you reference objects in extension files. To reference objects in the user's document, however, you must call `dw.getDocumentDOM()`, `dw.createDocument()`, or another function that returns a user document object.

For example, to refer to the first image in the active document you can write `dw.getDocumentDOM().images[0]`. You can also store the document object in a variable and use that variable in future references, as shown in the following example:

```
var dom = dw.getDocumentDOM(); //get the dom of the current document
var firstImg = dom.images[0];
firstImg.src = "myImages.gif";
```

This kind of notation is common in files throughout the Configuration folder, especially in command files. For more information about `dw.getDocumentDOM()`, see "dreamweaver.getDocumentDOM()" on page 453.

The Dreamweaver DOM

The Dreamweaver DOM contains a subset of objects, properties, and methods from the World Wide Web Consortium (W3C) (http://www.w3.org/TR/REC-DOM-Level-1/) DOM Level 1, which are combined with some properties of the Microsoft Internet Explorer 4.0 DOM.

Objects, properties, and methods of the Dreamweaver DOM

The following table lists the objects, properties, methods, and events that the Dreamweaver DOM supports. Some properties are read-only when they are accessed as properties of a specific object. A bullet (•) indicates properties that are read-only when used in the listed context.

Object	Properties	Methods	Events

`window`	`navigator` •
	`document` •
	`innerWidth` •
	`innerHeight` •
	`screenX` •
	`screenY` •
		`alert()`
		`confirm()`
		`escape()`
		`unescape()`
		`close()`
		`setTimeout()`
		`clearTimeout()`
		`setInterval()`
		`clearInterval()`
		`resizeTo()`
			`onResize`

`navigator`	`platform` •	None	None
`document`	`forms` • (an array of form objects) `images` • (an array of image objects) `layers` • (an array of `LAYER`, `ILAYER`, and absolutely positioned `DIV` and `SPAN` objects) `child` objects by name • `nodeType` • `parentNode` • `childNodes` • `documentElement` • `body` • `URL` • `parentWindow` •	`getElementsBy TagName()` `hasChildNodes()`	`onLoad`

Object	Properties	Methods	Events
all tags/elements	nodeType • parentNode • childNodes • tagName • attributes by name innerHTML outerHTML	getAttribute() setAttribute() removeAttribute() getElementsByTagName() hasChildNodes()	
form	In addition to the properties available for all tags: tags:elements • (an array of button, checkbox, password, radio, reset, select, submit, text, file, hidden, image, and textarea objects) mmcolorbutton child objects by name •	Only those methods available to all tags.	None
layer	In addition to the properties available for all tags: visibility left top width height zIndex	Only those methods available to all tags.	None
image	In addition to the properties available for all tags: src	Only those methods available to all tags.	onMouseOver onMouseOut onMouseDown onMouseUp
button reset submit	In addition to the properties available for all tags: form •	In addition to the methods available for all tags: blur() focus()	onClick
checkbox radio	In addition to the properties available for all tags: checked form •	In addition to the methods available for all tags: blur() focus()	onClick
password text file hidden image (field) textarea	In addition to the properties available for all tags: form • value	In addition to the methods available for all tags: blur() focus() select()	onBlur onFocus
select	In addition to the properties available for all tags: form • options • (an array of option objects) selectedIndex	In addition to the methods available for all tags: blur() (Windows only) focus() (Windows only)	onBlur (Windows only) onChange onFocus (Windows only)
option	In addition to the properties available for all tags: text	Only those methods available to all tags.	None

Object	Properties	Methods	Events
mmcolorbutton	In addition to the properties available for all tags: name value	None	onChange
array boolean date function math number object string regexp	MatchesNetscape Navigator 4	Matches Netscape 4	None
text	nodeType • parentNode • childNodes • data	hasChildNodes()	None
comment	nodeType • parentNode • childNodes • data	hasChildNodes()	None
NodeList	length •	item()	None
NamedNodeMap	length •	item()	None

Properties and methods of the document object

The following table details the properties and methods of the document object that are taken from DOM Level 1 and used in Dreamweaver. A bullet (•) marks read-only properties.

Property or method	Return value
nodeType •	Node.DOCUMENT_NODE
parentNode •	null
parentWindow •	The JavaScript object that corresponds to the document's parent window. (This property is not included in DOM Level 1; however, it is supported by Microsoft Internet Explorer 4.0.)
childNodes •	A NodeList that contains all the immediate children of the document object. Typically the document has a single child: the HTML object.
documentElement •	The JavaScript object that corresponds to the HTML tag. This property is shorthand for getting the value of document.childNodes and extracting the HTML tag from the NodeList.
body •	The JavaScript object that corresponds to the BODY tag. This property is shorthand for calling document.documentElement.childNodes and extracting the BODY tag from the NodeList. For frameset documents, this property returns the node for the outermost frameset.
URL •	The *file*://URL for the document or, if the file has not been saved, an empty string.

Property or method	Return value
`getElementsByTagName(tagName)`	A `NodeList` that can be used to step through tags of type *`tagName`* (for example, `IMG`, `DIV`, and so on). If the *`tag`* argument is `LAYER`, the function returns all `LAYER` and `ILAYER` tags and all absolutely positioned `DIV` and `SPAN` tags. If the *`tag`* argument is `INPUT`, the function returns all form elements. (If a name attribute is specified for one or more `tagName` objects, it must begin with a letter as required by the HTML 4.01 specification, or the length of the array that this function returns is incorrect.)
`hasChildNodes()`	`true`

Properties and methods of HTML tag objects

Every HTML tag is represented by a JavaScript object. Tags are organized in a tree hierarchy, where tag `x` is a parent of tag `y`, if `y` falls completely within `x`'s opening and closing tags (`<x>x content <y>y content</y> more x content.</x>`). For this reason, your code should be well-formed.

The following table lists the properties and methods of tag objects in Dreamweaver, along with their return values or explanations. A bullet (•) marks read-only properties.

Property or method	Return value
`nodeType` •	`Node.ELEMENT_NODE`
`parentNode` •	The parent tag. If this is the `HTML` tag, the document object returns.
`childNodes` •	A `NodeList` that contains all the immediate children of the tag.
`tagName` •	The HTML name for the tag, such as `IMG`, `A`, or `BLINK`. This value always returns in uppercase letters.
`attrName`	A string that contains the value of the specified tag attribute. `tag.`*`attrName`* cannot be used if *`attrName`* is a reserved word in the JavaScript language (for example, `class`). In this case, use `getAttribute()` and `setAttribute()`.
`innerHTML`	The source code that is contained between the beginning tag and the end tag.For example, in the code `<p><b>Hello</b>, World!</p>`, `p.innerHTML` returns `<b>Hello</b>, World!`. If you write to this property, the DOM tree immediately updates to reflect the new structure of the document. (This property is not included in DOM Level 1; however, it is supported by Internet Explorer 4.0.)
`outerHTML`	The source code for this tag, including the tag. For the previous example code, `p.outerHTML` returns `<p><b>Hello</b>, World!</p>`. If you write to this property, the DOM tree immediately updates to reflect the new structure of the document. (This property is not included in DOM Level 1; however, it is supported by Internet Explorer 4.0.)
`getAttribute(attrName)`	The value of the specified attribute if it is explicitly specified; otherwise, `null`.
`getTranslatedAttribute(attrName)`	The translated value of the specified attribute, or the same value that `getAttribute()` returns if the attribute's value is not translated. (This property is not included in DOM Level 1; it was added to Dreamweaver 3 to support attribute translation.)
`setAttribute(attrName, attrValue)`	Does not return a value. Sets the specified attribute to the specified value: for example, `img.setAttribute("src", "image/roses.gif")`.

Property or method	Return value
`removeAttribute(attrName)`	Does not return a value. Removes the specified attribute and its value from the HTML for this tag.
`getElementsByTagName(tagName)`	A `NodeList` that can be used to step through child tags of type *`tagName`* (for example, `IMG`, `DIV`, and so on). If the *`tag`* argument is `LAYER`, the function returns all `LAYER` and `ILAYER` tags and all absolutely positioned `DIV` and `SPAN` tags. If the *`tag`* argument is `INPUT`, the function returns all form elements. (If a name attribute is specified for one or more `tagName` objects, it must begin with a letter as required by the HTML 4.01 specification, or the length of the array returned by this function is incorrect.)
`hasChildNodes()`	A Boolean value that indicates whether the tag has any children.
`hasTranslatedAttributes()`	A Boolean value that indicates whether the tag has any translated attributes. (This property is not included in DOM Level 1; it was added to Dreamweaver 3 to support attribute translation.)

Properties and methods of text objects

Each contiguous block of text in an HTML document (for example, the text within a `P` tag) is represented by a JavaScript object. Text objects never have children. The following table describes the properties and methods of text objects that are taken from DOM Level 1 and used in Dreamweaver. A bullet (•) marks read-only properties.

Property or method	Return value
`nodeType` •	`Node.TEXT_NODE`
`parentNode` •	The parent tag
`childNodes` •	An empty `NodeList`
`data`	The actual text string. Entities in the text are represented as a single character (for example, the text `Joseph &amp; I` is returned as `Joseph & I`).
`hasChildNodes()`	`false`

Properties and methods of comment objects

Each HTML comment is represented by a JavaScript object. The following table details the properties and methods of comment objects that are taken from DOM Level 1 and are used in Dreamweaver. A bullet (•) marks read-only properties.

Property or method	Return value
`nodeType` •	`Node.COMMENT_NODE`
`parentNode` •	The parent tag
`childNodes` •	An empty `NodeList`
`data`	The text string between the comment markers (`<!-- and -->`)
`hasChildNodes()`	`false`

The dreamweaver and site objects

Dreamweaver implements the standard objects that are accessible through the DOM and adds two custom objects: `dreamweaver` and `site`. Both of these custom objects are widely used within the APIs and in writing extensions. For additional information on the methods of the `dreamweaver` and `site` objects, see "The Dreamweaver JavaScript API" on page 371.

Properties of the dreamweaver object

The `dreamweaver` object has two read-only properties, as described in the following list:

- `appName` has the value `"Dreamweaver"`.
- `appVersion` has a value of the form "*`versionNumber.releaseNumber.buildNumber [languageCode] (platform)`*".

As an example, the value of the `appVersion` property for the Swedish Windows version of Dreamweaver MX would be `"6.0.XXXX [se] (Win32)"`; the value for the English Macintosh version would be `"6.0.XXXX [en] (MacPPC)"`.

Note: The build number for the version that comes as Dreamweaver MX was not known when this documentation was printed. You can find the build number under Help > About.

The `appName` and `appVersion` properties were implemented in Dreamweaver 3 and are not available in earlier versions of Dreamweaver. You might want to check whether the user of your extension has Dreamweaver version 3 or later. To do this, check for the existence of the `appVersion` or `appName` property.

To check for a specific version of Dreamweaver, check first for the existence of `appVersion` and then for the version number, as shown in the following example:

```
if (dreamweaver.appVersion && ¬
dreamweaver.appVersion.indexOf('3.01') != -1){
  // execute code
}
```

The `dreamweaver` object has a property called `systemScript` that lets you query the language of the user's operating system. Use this property if you need to include special cases in your extension code for localized operating systems, as shown in the following example:

```
if (dreamweaver,systemScript && (dreamweaver.systemScript.indexOf('ja')!=-1){
SpecialCase
}
```

`systemScript` returns the following values for localized operating systems:

Language	Value
Japanese	ja
Korean	ko
TChinese	zh_tw
SChinese	zh_cn

Operating systems for all European languages return `'en'`.

The site object

The `site` object has no properties. For information about the methods of the `dreamweaver` and `site` objects, see "The Dreamweaver JavaScript API" on page 371.

Part II
Extension APIs

Understand about functions that you need to write when you create new objects, toolbars, tag editors, floating panels, server behaviors, components, or server models.

- Chapter 5, “Objects”
- Chapter 6, “Commands”
- Chapter 7, “Menu Commands”
- Chapter 8, “Toolbars”
- Chapter 9, “Reports”
- Chapter 10, “Tag Libraries and Editors”
- Chapter 11, “Property Inspectors”
- Chapter 12, “Floating Panels”
- Chapter 13, “Behaviors”
- Chapter 14, “Server Behaviors”
- Chapter 15, “Data Sources”
- Chapter 16, “Server Formats”
- Chapter 17, “Components”
- Chapter 18, “Server Models”
- Chapter 19, “Data Translators”
- Chapter 21, “C-Level Extensibility”

CHAPTER 5
Objects

Objects are designed to insert a specific string of code into a user's document. An object appears in a tab in the Insert bar and in the Insert menu when its Object file is stored in a subfolder within the Configuration/Objects folder. If you add a new object to the Insert bar, you must add a new subfolder for it within the Configuration/Objects folder and also edit the insertbar.xml file.

Objects have three components: the Object file that defines what is inserted in your document, the 18 x 18 pixel image that appears on the Insert bar, and the insertbar.xml file that defines where the object appears on the Insert bar.

Objects are HTML files. The `BODY` of an Object file can contain an HTML form that accepts parameters for the object (for example, the number of rows and columns to insert in a table). The `HEAD` of an Object file contains JavaScript functions that process form input from the `BODY` and control what is added to the user's document.

Note: The simplest objects contain only the HTML to insert, without a `BODY` and `HEAD` tag. See "Customizing Dreamweaver" on the Macromedia Support Center for more information.

How object files work

When a user selects an object by clicking an icon in the Insert bar or by selecting an item in the Insert menu, the following events occur:

1. Dreamweaver calls the `canInsertObject()` function to determine whether to show a dialog box.
2. The Object file is scanned for a `FORM` tag. If a form exists and if the Show Dialog When Inserting Objects option is selected in the General preferences, Dreamweaver calls the `windowDimensions()` function, if defined, to determine the size of the dialog box in which to display the form. If no form exists in the Object file, Dreamweaver does not display a dialog box, and skips step 2.
3. If Dreamweaver displays a dialog box in step 1, the user enters parameters for the object (such as the number of rows and columns in a table) in the dialog box and clicks OK.
4. The `objectTag()` function is called, and its return value is inserted into the document after the current selection (it does not replace the current selection).
5. If Dreamweaver does not find the `objectTag()` function, it looks for an `insertObject()` function and calls that function instead.

Adding objects to the Insert bar

Each Object file has an associated 18 x 18 pixel image that appears in the Insert bar.

If you create a larger object image, Dreamweaver scales it to 18 x 18 pixels. If you do not create an image for your object, a default object icon appears in the Insert bar.

Note: Although Object files can be stored in separate folders, it's important that each filename be unique. The "dom.insertObject()" on page 471, for example, looks for a specified file anywhere within the Objects folder without regard to subfolders. If a file called Button.htm exists in the Forms folder and also in the MyObjects folder, Dreamweaver cannot distinguish between them.

Defining the Insert bar

The Insert bar is defined by the insertbar.xml file that is found in the Configurations/Objects folder.

The XML file contains definitions for each individual object, in the order that the objects appear.

The first time that the user launches Dreamweaver, the Insert bar appears horizontally above the document. After that, its visibility and position are saved in the registry.

The following example illustrates the format for the insertbar.xml file:

```
<?xml version="1.0"?>
<insertbar >
  <category id="DW_Insertbar_Common" folder="Common">
    <button id="DW_TagDialog"
      image="Objects/Common/tagDialog.gif"
      enabled="true"
      showIf="_VIEW_CODE"
      command="dw.getDocumentDOM().setView('code')"/>
    <separator showIf="_VIEW_CODE"/>

    <button id="DW_BR"
      image="Objects/Common/BR.gif"
      enabled="true"
      file="Objects/Common/br.htm"/>
    ...
    </category >
</insertbar>
```

Insert bar definition tags

The Insert bar has category, button, checkbutton, and separator items. The following sections describe the tags for these items.

<insertbar>

Description

Signals the beginning of the Insert bar definition file.

Attributes

None.

Contents

The `category` tag and its contents.

Container

None.

Example

```
<insertbar>
```

<category>

Description

Defines a tab on the Insert bar.

Attributes

`id, folder, {showif}`

Contents

Contains `button`, `checkbutton`, and `separator` tags.

Container

The `insertbar` tag.

Example

```
<category id="DW_Insertbar_Text" folder="Text">
```

<button>

Description

Defines a pushbutton. Executes the code that the `command` or `file` attributes specify.

Attributes

`id, image, {disabledImage}, {showif}, {enabled}, {command}, {file}, {tag}, {name}, {codeOnly}`

Contents

None.

Container

The `category` tag.

Example

```
<button id="DW_Anchor"
  image="Common\Anchor.gif"
  enabled="true"
  showIf=""
  file="Common\Anchor.htm"/>
```

<checkbutton>

Description

A button that has a checked or unchecked state. When you click it, a checkbutton displays as pressed in and highlighted. When it is unchecked, a checkbutton displays as flat. Dreamweaver has mouse-over, pressed, mouse-over-while-pressed, and disabled-while-pressed states. The command must ensure that clicking the checkbutton causes its state to change.

Attributes

id, image, {disabledImage}, {showIf}, {enabled}, {checked}, {command}, {file}, {tag}, {name}

Contents

None.

Container

The category tag.

Example

```
<checkbutton id="DW_StandardView"
  name = "Standard View"
  image="Tools\Standard View.gif"
  checked="_View_Standard"
  command="dw.getDocumentDOM().setShowLayoutView(false)"/>
```

<separator>

Description

Displays a vertical line on the Insert bar.

Attributes

(showIf}

Contents

None.

Container

The category tag.

Example

```
<separator showIf="_VIEW_CODE"/>
```

Insert bar tag attributes

The attributes for the Insert bar tags have the following meanings:

id="unique_id"

Required. The id is an identifier for tags in the insertbar.xml file. The id must be unique identifier for the element within the file.

Example

```
<category id="DW_Insertbar_Layout" . . .>
```

folder="category_folder"

Specifies a folder in the Dreamweaver Configuration/Objects folder. Dreamweaver takes the name of the category from the _folderinfo.txt file inside the folder, or from the folder name if the _folderinfo.txt file does not exist.

Example

```
folder="Tools"
```

image="image_path"

Required. Specifies the path, relative to the Dreamweaver Configuration folder, to the icon file that appears on the Insert bar. The icon can be in any format that Dreamweaver can render, but typically it is a GIF or JPEG file format.

Example

```
image="Common/Table.gif"
```

showIf="DW_enabler"

Optional. Specifies that this item should appear on the Insert bar only if the given Dreamweaver enabler is `true`. If you do not specify `showIf`, the item always appears. The possible enablers are `_SERVERMODEL_ASP`, `_SERVERMODEL_ASPNET`, `_SERVERMODEL_JSP`, `_SERVERMODEL_CFML` (for both new and old versions of ColdFusion), `_SERVERMODEL_CFML_UD4` (true only for UltraDev version 4 of ColdFusion), `_SERVERMODEL_PHP`, `_FILE_TEMPLATE`, `_VIEW_CODE`, `_VIEW_DESIGN`, `_VIEW_LAYOUT`, and `_VIEW_STANDARD`.

You can specify multiple enablers by placing a comma (which means AND) between the enablers. For example, if you want an object to appear only in Code view for an ASP page, specify the enablers as `showIf="_VIEW_CODE, _SERVERMODEL_ASP"`. You can also specify NOT with `"!"`.

Example

```
showIf="_VIEW_CODE, _SERVERMODEL_CFML"
```

enabled="DW_enabler"

Optional. Specifies that the item is enabled if *`DW_enabler`* is true. If you do not specify `enabled`, the item defaults to always enabled. The possible enablers are `_SERVERMODEL_ASP`, `_SERVERMODEL_ASPNET`, `_SERVERMODEL_JSP`, `_SERVERMODEL_CFML` (for both new and old versions of ColdFusion), `_SERVERMODEL_CFML_UD4` (true only for UltraDev version 4 of ColdFusion), `_SERVERMODEL_PHP`, `_FILE_TEMPLATE`, `_VIEW_CODE`, `_VIEW_DESIGN`, `_VIEW_LAYOUT`, and `_VIEW_STANDARD`.

You can specify multiple enablers by placing a comma (which means AND) between the enablers. You can also specify NOT with `"!"`.

Example

```
enabled="_View_Standard"
```

checked="DW_enabler"

Required for checkbuttons. The item is checked if *`DW_enabler`* is `true`. The possible enablers are `_SERVERMODEL_ASP`, `_SERVERMODEL_ASPNET`, `_SERVERMODEL_JSP`, `_SERVERMODEL_CFML` (for both new and old versions of ColdFusion), `_SERVERMODEL_CFML_UD4` (only for UltraDev version 4 of ColdFusion), `_SERVERMODEL_PHP`, `_FILE_TEMPLATE`, `_VIEW_CODE`, `_VIEW_DESIGN`, and `_VIEW_LAYOUT`.

You can specify multiple enablers by placing a comma (which means AND) between them.You can also specify NOT with `"!"`.

Example

```
checked="_View_Layout"
```

command="script "

Required unless the `command` attribute is specified. Do not specify both the `command` and the `file` attributes for an object. The `command` attribute specifies JavaScript code to execute when the user clicks the button.

Example

```
command="dw.getDocumentDOM().setShowLayoutView(true)"
```

file="object_file_path"

Required unless the `command` attribute is specified. The `file` attribute specifies the path, relative to the Dreamweaver Configuration folder, of an object file. Dreamweaver takes the tooltip for the object from the title of the object file.

Example

```
file="Templates/Editable.htm"
```

tag="tagStr"

Optional. Defines the tag for which to invoke a tag editor. In Code view, if the tag attribute is defined and the user clicks on the object, Dreamweaver invokes the Tag dialog box. In Code view, if both `tag` and `command` are specified, Dreamweaver invokes the tag editor. In Design view, if `codeOnly="TRUE"` and the `file` attribute is not specified, Dreamweaver MX invokes Split view, places focus in the code, and invokes the tag editor.

Example

```
tag = "form"
```

name="nameStr"

Optional. The `name` attribute specifies the tooltip that appears when the mouse cursor hovers over the object. If you specify an object file but do not specify the `name` attribute, Dreamweaver uses the name of the object file for the tooltip.

Example

```
name = "cfoutput"
```

codeOnly = "boolStr"

Optional. Specifies whether the object is only meant for Code view because it has no visual representation in Design view. The value of `boolStr` must be `"true"` or `"false"`.

Adding Objects to the Insert menu

Dreamweaver automatically adds any files that are inside one of the subfolders in the Configuration/Objects folder to the bottom of the Insert menu.

To control the position of an object in the Insert menu or any other menu, or to add an object to multiple menus, you can modify the menus.xml file. This file controls the entire menu structure for Dreamweaver. For more information about modifying the menus.xml file, see “Customizing Dreamweaver” on the Macromedia Support Center.

The Objects API

This section describes the functions in the Objects API. You must define either the `insertObject()` function or the `objectTag()` function. The remaining functions are optional.

canInsertObject()

Availability

Dreamweaver MX

Description

Determines whether to display the Object dialog box.

Arguments

None.

Returns

Dreamweaver expects a Boolean value.

Example

```
function canInsertObject(){
   var docStr = dw.getDocumentDOM().documentElement.outerHTML;
   var patt = /hava/;
   var found = ( docStr.search(patt) != -1 );
   var insertionIsValid = true;

   if (!found){
    insertionIsValid = false;
    alert("the document must contain a 'hava' string to use this object.\nHa
  Ha.");  }
   return insertionIsValid;}
```

displayHelp()

Description

If this function is defined, displays a Help button below the OK and Cancel buttons in the Parameters dialog box. This function is called when the user clicks the Help button.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

```
// the following instance of displayHelp() opens
// in a browser a file that explains how to use
// the extension.
function displayHelp(){
  var myHelpFile = dw.getConfigurationPath() +
    '/ExtensionsHelp/superDuperHelp.htm';
  dw.browseDocument(myHelpFile);
}
```

isDomRequired()

Description

Determines whether the object requires a valid DOM to operate. If this function returns `true` or if the function is not defined, Dreamweaver assumes that the command requires a valid DOM and synchronizes the Code and Design views for the document before executing. Synchronization causes all edits in the Code view to be updated in the Design view.

Arguments

None.

Returns

Dreamweaver expects `true` if a command requires a valid DOM to operate; `false` otherwise.

insertObject()

Availability

Dreamweaver MX

Description

Required if `objectTag()` is not defined. Called when the user clicks OK; either inserts code into the user's document and dismisses the dialog box, or displays an error message and leaves the dialog box open. This works as an alternate function to use in objects instead of `objectTag()`. It does not assume that the user is inserting text at the current insertion point and allows for data validation when the user clicks OK. You should use `insertObject()` if one of the following conditions exists:

- You need to insert code in more than one place.
- You need to insert code somewhere other than the insertion point.
- You need to validate input before inserting.

If none of these conditions apply, use `objectTag()`.

Arguments

None.

Returns

Dreamweaver expects a string that contains an error message or an empty string. If it returns an empty string, the Object dialog box closes when the user clicks OK. If it is not empty, Dreamweaver displays the error message and the dialog box remains.

Enabler

`canInsertObject()`

Example

```
function insertObject() {
  var theForm = document.forms[0];
  var nameVal = theForm.firstField.value;
  var passwordVal = theForm.secondField.value;
  var errMsg = "",
  var isValid = true;

  // ensure that field values are complete and valid
  if (nameVal == "" || passwordVal == "") {
    errMsg = "Complete all values or click Cancel."
  } else if (nameVal.length < 4 || passwordVal.length < 6) {
    errMsg = "Your name must be at least four characters, and your password at
least six";
  }

  if (!errMsg) {
  // do some document manipulation here. Exercise left to the reader
  }
  return errMsg;
  }
```

objectTag()

Description

The functions `objectTag()` and `insertObject()` are mutually exclusive; if both are defined in a document, then `objectTag()` is used. See the `insertObject()` function for more information.

Inserts a string of code into the user's document. In Dreamweaver 4, if the focus was in Code view and the selection was a range (meaning not an insertion point), the range was replaced by the string that `objectTag()` returns. This is true, even if `objectTag()` returned an empty string or returned nothing. Because the main reason for returning an empty string, or `null`, from `objectTag()` was because edits to the document have already been made manually, having the selection be replaced by "" often deleted the edit. In Dreamweaver MX, returning an empty string, or `null` (also known as "`Return;`"), is a signal to Dreamweaver to do nothing.

Note: The assumption is that edits have been made manually prior to the `return` statement, so doing nothing in this case is ***not*** equivalent to clicking Cancel.

Arguments

None.

Returns

Dreamweaver expects the string to be inserted in the user's document.

Example

The following instance of `objectTag()` inserts an `OBJECT/EMBED` combination for a specific ActiveX control and plug-in:

```
function objectTag() {
return '\n' +
'<OBJECT CLASSID="clsid:166F100B-3A9R-11FB-8075444553540000" \n'¬
+ 'CODEBASE="http://www.mysite.com/product/cabs/¬
myproduct.cab#version=1,0,0,0" \n' + 'NAME="MyProductName"> \n' ¬
+ '<PARAM NAME="SRC" VALUE=""> \n' + '<EMBED SRC="" HEIGHT="" ¬
WIDTH="" NAME="MyProductName"> \n' + '</OBJECT>'
}
```

windowDimensions()

Description

Sets specific dimensions for the Options dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an Options dialog box that is larger than 640 x 480 pixels.

Arguments

platform

The value of the argument is either `"macintosh"` or `"windows"`, depending on the user's platform.

Returns

Dreamweaver expects a string of the form `"widthInPixels,heightInPixels"`.

The returned dimensions are smaller than the size of the entire dialog box because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example

The following instance of `windowDimensions()` sets the dimensions of the Parameters dialog box to 648 x 520 pixels for Windows and 660 x 580 pixels for the Macintosh:

```
function windowDimensions(platform){
  var retval = ""
  if (platform = = "windows"){
  retval = "648, 520";
  }else{
  retval = "660, 580";
  }
  return retval;
}
```

CHAPTER 6
Commands

Commands can be used to perform almost any kind of edit to a user's current document, other open documents, or to any HTML document on a local drive. Commands can insert, remove, or rearrange HTML tags and attributes, comments, and text.

Commands are HTML files. The `BODY` of a Command file can contain an HTML form that accepts options for the command (for example, how a table should be sorted and by which column). The `HEAD` of a Command file contains JavaScript functions that process form input from the `BODY` and control what edits are made to the user's document.

How commands work

When a user clicks a menu that contains a command, the following events occur:

1. Dreamweaver calls the `canAcceptCommand()` function to determine whether the menu item should be disabled. If `canAcceptCommand()` returns `false`, the command is dimmed in the menu, and the procedure stops. If `canAcceptCommand()` returns `true`, the procedure can continue.
2. The user selects a command from the menu.
3. Dreamweaver calls the `receiveArguments()` function, if defined, in the selected Command file to let the command process any arguments that are passed from the menu item or from the function "dreamweaver.runCommand()" on page 400.
4. Dreamweaver calls the `commandButtons()` function, if defined, to determine which buttons appear on the right side of the Options dialog box and what code should execute when the user clicks the buttons.
5. Dreamweaver scans the Command file for a `FORM` tag. If a form exists, Dreamweaver calls the `windowDimensions()` function, which sizes the Options dialog box that contains the `BODY` elements of the file. If `windowDimensions()` is not defined, Dreamweaver automatically sizes the dialog box.
6. If the Command file's `BODY` tag contains an `onLoad` handler, Dreamweaver executes it (whether or not a dialog box appears). If no dialog box appears, the remaining steps do not occur.
7. The user selects options for the command. Dreamweaver executes event handlers that are associated with the fields as the user encounters them.
8. The user clicks one of the buttons that is defined by `commandButtons()`.
9. Dreamweaver executes the associated code. The dialog box remains visible until one of the scripts in the command calls `window.close()`.

The Command API

The custom functions in the Command API are not required.

canAcceptCommand()

Description

Determines whether the command is appropriate for the current selection.

Note: Do not define `canAcceptCommand()` unless it returns `false` in at least one case. If the function is not defined, the command is assumed to be appropriate; making this assumption saves time and improves performance.

Arguments

None.

Returns

Dreamweaver expects `true` if the command is allowed; `false` otherwise, dimming the command in the menu.

Example

The following instance of `canAcceptCommand()` makes the command available only when the selection is a table:

```
function canAcceptCommand(){
  var retval=false;
  var selObj=dw.getDocumentDOM.getSelectedNode();
  return (selObj.nodeType == Node.ELEMENT_NODE && ¬
  selObj.tagName=="TABLE");{
    retval=true;
}
return retval;
}
```

commandButtons()

Description

Defines the buttons that should appear on the right side of the Options dialog box and their behavior when they are clicked. If this function is not defined, no buttons appear, and the `BODY` of the Command file expands to fill the entire dialog box.

Arguments

None.

Returns

Dreamweaver expects an array that contains an even number of elements. The first element is a string that contains the label for the topmost button. The second element is a string of JavaScript code that defines the behavior of the topmost button when it is clicked. Remaining elements define additional buttons in the same way.

Example

The following instance of `commandButtons()` defines three buttons: OK, Cancel, and Help.

```
function commandButtons(){
  return new Array("OK" , "doCommand()" , "Cancel" , ¬
  "window.close()" , "Help" , "showHelp()");
}
```

isDomRequired()

Description

Determines whether the command requires a valid DOM to operate. If this function returns `true` or if the function is not defined, Dreamweaver assumes that the command requires a valid DOM and synchronizes the Design and Code views of the document before executing. Synchronization causes all edits in the Code view to be updated in the Design view.

Arguments

None.

Returns

Dreamweaver expects `true` if a command requires a valid DOM to operate; `false` otherwise.

receiveArguments()

Description

Processes any arguments that are passed from a menu item or from `dw.runCommand()`, if any arguments are passed via the `dw.runCommand()` function.

Arguments

`{arg1}`, *`{arg2}`*,...*`{argN}`*

If the `arguments` attribute is defined for a `menuitem` tag, the value of that attribute passes to the `receiveArguments()` function as one or more arguments. Arguments can also be passed to a command by the `dw.runCommand()` function.

Returns

Dreamweaver expects nothing.

windowDimensions()

Description

Sets specific dimensions for the Parameters dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an Options dialog box that is larger than 640 x 480 pixels.

Arguments

platform

The value of the argument is either `"macintosh"` or `"windows"`, depending on the user's platform.

Returns

Dreamweaver expects a string of the form `"widthInPixels,heightInPixels"`.

The returned dimensions are smaller than the size of the entire dialog box because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example

The following example of `windowDimensions()` sets the dimensions of the Parameters dialog box to 648 x 520 pixels:

```
function windowDimensions(){
  return "648,520";
}
```

A simple command example

The following command converts the selected text to all lowercase characters. The command is very simple. It does not display a dialog box, so the `commandButtons()` function is not defined.

```
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine 5.0//dialog">
<HTML>
<HEAD>
<TITLE>Make Lower Case</TITLE>
<SCRIPT LANGUAGE="javascript">

function canAcceptCommand(){
  // Get the DOM of the current document
  var theDOM = dw.getDocumentDOM();
  // Get the offsets of the selection
  var theSel = theDOM.getSelection();
  // Get the selected node
  var theSelNode = theDOM.getSelectedNode();
  // Get the children of the selected node
  var theChildren = theSelNode.childNodes;

  // If the selection is not an insertion point, and
  // either the selection or its first child is a
  // text node, return true.
  return (theSel[0] != theSel[1] && (theSelNode.nodeType == ¬
  Node.TEXT_NODE || theChildren[0].nodeType == Node.TEXT_NODE));
}

function changeToLowerCase() {
  // Get the DOM again
  var theDOM = dw.getDocumentDOM();
  // Get the offsets of the selection
  var theSel = theDOM.getSelection();

  // Get the outerHTML of the HTML tag (the
  // entire contents of the document)
  var theDocEl = theDOM.documentElement;
  var theWholeDoc = theDocEl.outerHTML;

  // Extract the selection
  var selText = theWholeDoc.substring(theSel[0],theSel[1]);

  // Re-insert the modified selection into the document
  theDocEl.outerHTML = theWholeDoc.substring(0,theSel[0]) + ¬
  selText.toLowerCase() + theWholeDoc.substring(theSel[1]);

  // Set the selection back to where it was when you
  // started
  theDOM.setSelection(theSel[0],theSel[1]);
}

</SCRIPT>
</HEAD>

<BODY onLoad="changeToLowerCase()">

<!-- The function that does all the work in this command is
called from the onLoad handler on the BODY tag. There is no form
in the BODY, so no dialog box appears. -->

</BODY>
</HTML>
```

Adding commands to the Commands menu

Dreamweaver automatically adds any files that are inside the Configuration/Commands folder to the bottom of the Commands menu. To prevent a command from appearing in the Commands menu, put the following comment on the first line of the file:

```
<!-- MENU-LOCATION=NONE -->
```

CHAPTER 7
Menu Commands

Menu commands make menus more flexible and dynamic. As with regular commands, menu commands can be used to perform almost any kind of edit to the current document, other open documents, or any HTML document on a local drive. The Menu Commands API expands the regular command API to accomplish several tasks that are related to displaying and calling the command from the menu system.

Note: Because menu commands are directly related to the menu system in Dreamweaver, you should read "Customizing Dreamweaver," in *Using Dreamweaver* before continuing in this chapter.

Menu commands are HTML files that are referenced in the `file` attribute of a `menuitem` tag in the `menus.xml` file. The `BODY` of a Menu Commands file can contain an HTML form that accepts options for the command (for example, how a table should be sorted and by which column). The `HEAD` of a Menu Commands file contains JavaScript functions that process form input from the `BODY` and control the edits that are made to the user's document.

Menu commands are stored in the Configuration/Menus folder inside the Dreamweaver application folder.

Note: If you add custom menu commands to Dreamweaver, add them at the top level of the Menus folder or create a subfolder. The MM folder is reserved for the menu commands that come with Dreamweaver.

How menu commands work

When the user clicks a menu with a menu item that contains a menu command, the following events occur:

1. If any `menuitem` tag in the menu contains the `dynamic` attribute, Dreamweaver calls the `getDynamicContent()` function in the associated Menu Commands file to populate the menu.
2. Dreamweaver calls the `canAcceptCommand()` function in each Menu Commands file that is referenced in the menu to check whether the command is appropriate for the selection.

- If `canAcceptCommand()` returns `false`, the menu item is dimmed.
- If `canAcceptCommand()` returns `true` or is not defined, Dreamweaver calls the `isCommandChecked()` function to determine whether to display a check mark next to the menu item. If `isCommandChecked()` is not defined, no check mark appears.

3. Dreamweaver calls the `setMenuText()` function to determine the text that should appear in the menu.

   If `setMenuText()` is not defined, Dreamweaver uses the text that is specified in the `menuitem` tag.
4. The user selects an item from the menu.

5 Dreamweaver calls the `receiveArguments()` function, if defined, in the selected Menu Commands file to let the command process any arguments that are passed from the menu item.

Note: If it is a dynamic menu item, the ID of the menu item is passed as the only argument.

6 Dreamweaver calls the `commandButtons()` function, if defined, to determine which buttons appear on the right side of the Options dialog box and what code should execute when the user clicks the buttons.

7 Dreamweaver scans the Menu Commands file for a `FORM` tag.

If a form exists, Dreamweaver calls the `windowDimensions()` function to determine the size of the Options dialog box that contains the `BODY` elements of the file.

If `windowDimensions()` is not defined, Dreamweaver automatically sizes the dialog box.

8 If the Menu Commands file's `BODY` tag contains an `onLoad` handler, Dreamweaver executes the associated code (whether or not a dialog box appears). If no dialog box appears, the remaining steps do not occur.

9 The user selects options in the dialog box. Dreamweaver executes event handlers that are associated with the fields as the user encounters them.

10 The user clicks one of the buttons that are defined by `commandButtons()`.

11 Dreamweaver executes the code that is associated with the clicked button.

12 The dialog box remains visible until one of the scripts in the Menu Commands calls `window.close()`.

The Menu Commands API

The custom functions in the Menu Commands API are not required.

canAcceptCommand()

Description

Determines whether the menu item should be active or dimmed.

Arguments

`{arg1}`, *`{arg2}`*,...*`{argN}`*

If it is a dynamic menu item, the unique ID given in `getDynamicContents()` is the only argument. Otherwise, if the `arguments` attribute is defined for a `menuitem` tag, the value of that attribute passes to the `canAcceptCommand()` function (and to the "`isCommandChecked()`" on page 70, "`receiveArguments()`" on page 70, and "`setMenuText()`" on page 71 functions) as one or more arguments. The `arguments` attribute is useful for distinguishing between two menu items that call the same menu command.

Note: The `arguments` attribute is ignored for dynamic menu items.

Returns

Dreamweaver expects a Boolean value that indicates whether the item should be enabled.

commandButtons()

Description

Defines the buttons that should appear on the right side of the Options dialog box and their behavior when they are clicked. If this function is not defined, no buttons appear, and the `BODY` of the Menu Commands file expands to fill the entire dialog box.

Arguments

None.

Returns

Dreamweaver expects an array that contains an even number of elements. The first element is a string that contains the label for the topmost button. The second element is a string of JavaScript code that defines the behavior of the topmost button when it is clicked. The remaining elements define additional buttons in the same manner.

Example

The following example of `commandButtons()` defines three buttons: OK, Cancel, and Help.

```
function commandButtons(){
  return new Array("OK" , "doCommand()" , "Cancel" , ¬
  "window.close()" , "Help" , "showHelp()");
}
```

getDynamicContent()

Description

Retrieves the content for the dynamic portion of the menu.

Arguments

menuID

The argument is the value of the `id` attribute in the `menuitem` tag that is associated with the item.

Returns

Dreamweaver expects an array of strings where each string contains the name of a menu item and its unique ID, separated by a semicolon. If the function returns `null`, the menu does not change.

Example

The following example of `getDynamicContent()` returns an array of four menu items (My Menu Item 1, My Menu Item 2, and so on):

```
function getDynamicContent(){
  var stringArray= new Array();
  var i=0;
  var numItems = 4;

  for (i=0; i<numItems;i++)
    stringArray[i] = new String("My Menu Item " + i + ";¬
    id='My-MenuItem" + i + "'");

  return stringArray;
}
```

isCommandChecked()

Description

Determines whether to display a check mark next to the menu item

Arguments

{arg1}, *{arg2}*,...*{argN}*

If it is a dynamic menu item, the unique ID given in `getDynamicContents()` is the only argument. Otherwise, if the `arguments` attribute is defined for a `menuitem` tag, the value of that attribute passes to the `isCommandChecked()` function (and to the "`canAcceptCommand()`" on page 68, "`receiveArguments()`" on page 70, and "`setMenuText()`" on page 71 functions) as one or more arguments. The `arguments` attribute is useful for distinguishing between two menu items that call the same menu command.

Note: The `arguments` attribute is ignored for dynamic menu items.

Returns

Dreamweaver expects a Boolean value that indicates whether a check mark should appear next to the menu item.

Example

```
function isCommandChecked()
{
  var bChecked = false;
  var cssStyle = arguments[0];

  if (dw.getDocumentDOM() == null)
    return false;

  if (cssStyle == "(None)")
  {
    return dw.cssStylePalette.getSelectedStyle() == '';
  }
  else
  {
    return dw.cssStylePalette.getSelectedStyle() == cssStyle;
  }
    return bChecked;
}
```

receiveArguments()

Description

Processes any arguments that are passed from a menu item or from `dw.runCommand()`. If it is a dynamic menu item, it processes the dynamic menu item ID.

Arguments

{arg1}, *{arg2}*,...*{argN}*

If it is a dynamic menu item, the unique ID that is given in `getDynamicContents()` is the only argument. Otherwise, if the `arguments` attribute is defined for a `menuitem` tag, the value of that attribute passes to the `receiveArguments()` function (and to the "`canAcceptCommand()`" on page 68, "`isCommandChecked()`" on page 70, and "`setMenuText()`" on page 71 functions) as one or more arguments. The `arguments` attribute is useful for distinguishing between two menu items that call the same menu command.

Note: The `arguments` attribute is ignored for dynamic menu items.

Returns

Dreamweaver expects nothing.

Example

```
function receiveArguments()
{
  var styleName = arguments[0];
  if (styleName == "(None)")
    dw.getDocumentDOM('document').applyCSSStyle('','');
  else
    dw.getDocumentDOM('document').applyCSSStyle('',styleName);
}
```

setMenuText()

Description

Specifies the text that should appear in the menu.

Note: Do not use this function if you are using "getDynamicContent()" on page 69.

Arguments

{arg1}, *{arg2}*,...*{argN}*

If the `arguments` attribute is defined for a `menuitem` tag, the value of that attribute passes to the `setMenuText()` function (and to the "canAcceptCommand()" on page 68, "isCommandChecked()" on page 70, and "receiveArguments()" on page 70 functions) as one or more arguments. The `arguments` attribute is useful for distinguishing between two menu items that call the same menu command.

Returns

Dreamweaver expects the string that should appear in the menu.

Example

```
function setMenuText()
{
  if (arguments.length != 1) return "";

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
    return dw.getUndoText();
  else if (whatToDo == "redo")
    return dw.getRedoText();
  else return "";
}
```

windowDimensions()

Description

Sets specific dimensions for the Parameters dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want a dialog box larger than 640 x 480 pixels.

Arguments

platform

The value of the argument is either `"macintosh"` or `"windows"`, depending on the user's platform.

Returns

Dreamweaver expects a string of the form `"widthInPixels,heightInPixels"`.

The returned dimensions are smaller than the size of the entire dialog box because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example

The following example of `windowDimensions()` sets the dimensions of the Parameters dialog box to 648 x 520 pixels:

```
function windowDimensions(){
  return "648,520";
}
```

A simple menu command

The following menu command is associated with two menu items: Undo and Redo. It checks the `arguments` attribute of the `menuitem` tag and performs a `dw.undo()` or a `dw.redo()` operation, depending on the value of the first (and only) argument.

```
<HTML>
<HEAD>
<!-- Copyright 1999 Macromedia, Inc. All rights reserved. -->
<TITLE>Edit Clipboard</TITLE>
<SCRIPT LANGUAGE="javascript">
function receiveArguments()
{
  if (arguments.length != 1) return;

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
  {
    dw.undo();
  }
  else if (whatToDo == "redo")
  {
    dw.redo();
  }
}

function canAcceptCommand()
{
  var selarray;
  if (arguments.length != 1) return false;
  var bResult = false;

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
  {
    bResult = dw.canUndo();
  }
  else if (whatToDo == "redo")
  {
    bResult = dw.canRedo();
  }
  return bResult;
}

function setMenuText()
{
  if (arguments.length != 1) return "";

  var whatToDo = arguments[0];
  if (whatToDo == "undo")
    return dw.getUndoText();
  else if (whatToDo == "redo")
    return dw.getRedoText();
  else return "";
}

</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

In this command, the `receiveArguments()` function processes the arguments and executes the command. More complex menu commands might call different functions to execute the command. For example, the following code checks whether the first argument is `"foo"`; if it is, it calls the `doOperationX()` function and passes it the second argument. If the first argument is `"bar"`, it calls the `doOperationY()` function and passes it the second argument. `doOperationX()` or `doOperationY()` is responsible for executing the command.

```
function receiveArguments(){
  if (arguments.length != 2) return;

  var whatToDo = arguments[0];

  if (whatToDo == "foo"){
    doOperationX(arguments[1]);
  }else if (whatToDo == "bar"){
    doOperationX(arguments[1]);
  }
}
```

A simple dynamic menu

The following menu command generates the Preview in Browser submenu, and it launches the current file (or the selected files in the Site panel) in the browser that the user chooses from the submenu.

```
<HTML>
<HEAD>
<!-- Copyright 1999 Macromedia, Inc. All rights reserved. -->
<TITLE>Preview Browsers</TITLE>
<SCRIPT LANGUAGE="javascript">
<!--
  // getDynamicContent returns the contents of a dynamically
  // generated menu.
  // returns an array of strings to be placed in the menu, with a unique
  // identifier for each item separated from the menu string by a
  // semicolon.
  //
  // return null from this routine to indicate that you are not
  // adding any
  // items to the menu
  function getDynamicContent(itemID)
  {
    var browsers = null;
    var PIB = null;
    var i;
    var j=0;
    var bUpdate = dw.getMenuNeedsUpdating(itemID);

    if (bUpdate)
    {
      browsers = new Array();
      PIB = dw.getBrowserList();
      // each browser pair has the name of the browser and the path
      // that leads to the application on disk. We only put the
      // names in the menus.
      for (i=0; i<PIB.length; i=i+2)
      {
        browsers[j] = new String(PIB[i]);

        if (dw.getPrimaryBrowser() == PIB[i+1])
           browsers[j] += "\tF12";
        if (navigator.platform == "MacPPC")
        {
           if (dw.getSecondaryBrowser() == PIB[i+1])
            browsers[j] += "\t ?F12";
        }
        else
        {
          if (dw.getSecondaryBrowser() == PIB[i+1])
            browsers[j] += "\t Ctrl+F12";
        }

        browsers[j] += ";id='"+PIB[i]+"'";
        j = j+1;
      }
      dw.notifyMenuUpdated(itemID, "dw.getBrowserList()");
    }
    return browsers;
  }

function canAcceptCommand()
```

```
  {
    var bHaveDocument;

    if (dw.getFocus() == 'site')
      bHaveDocument = site.getSelection().length > 0;
    else
      bHaveDocument = dw.getDocumentDOM('document') != null;

    return bHaveDocument;
  }

function receiveArguments()
  {
    var theBrowser = arguments[0];
    if (dw.getFocus() == 'site')
      dw.browseDocument(site.getSelection(),theBrowser);
    else
      dw.browseDocument(dw.getDocumentPath('document'),theBrowser);
  }

// -->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

CHAPTER 8
Toolbars

You can create a toolbar for Macromedia Dreamweaver MX simply by creating a file that defines the toolbar and placing that file in the Configuration/Toolbars folder. Within a toolbar file, you can define items such as check buttons, radio buttons, text boxes, and pop-up menus using a few custom XML tags. You can assign attributes and commands to toolbar items to specify how they look and behave, include other toolbar files, and reference toolbar items that are defined in other toolbars.

How toolbars work

Toolbars are defined by XML and image files that are stored in the Toolbars folder of the main Dreamweaver Configuration folder. The default Dreamweaver toolbars are stored in the Configuration/Toolbars/toolbars.xml file. At start up, Dreamweaver loads all the toolbar files in the Toolbars folder. You can add new toolbars simply by copying a file into the Toolbars folder, rather than modifying the main toolbars.xml file.

Toolbar XML files define one or more toolbars and their toolbar items. A toolbar is a list of items such as buttons, text boxes, pop-up menus, and so on. A toolbar item represents a single control that a user can access in a toolbar.

Some types of toolbar controls, such as push buttons and pop-up menus, have icon images associated with them. Icon images are stored in an images folder below the Toolbars folder. Images can be in any format that Dreamweaver can render but are typically GIF or JPEG file formats. Images for Macromedia-authored toolbars are stored in the Toolbars/images/MM folder.

As with menus, you can specify the functionality of individual toolbar items either through the item attributes or through a command file. Macromedia-authored toolbar command files are stored in the Toolbars/MM folder.

Tip: The Toolbar API is compatible with the Menu Commands API, so toolbar controls can reuse menu command files.

Unlike menus, you can define toolbar items independently from the toolbars that use them. This flexibility lets you use toolbar items in multiple toolbars by using the `itemref` tag.

The first time Dreamweaver loads a toolbar, its visibility and position are set by the toolbar definition. After that, its visibility and position are saved in and restored from the registry (Windows) or the Dreamweaver MX Preferences file (Macintosh).

How toolbars behave

In Windows, Dreamweaver MX toolbars generally act the same as standard Windows toolbars. Dreamweaver MX toolbars have the following characteristics:

- You can drag and drop toolbars to dock them, undock them, and reposition them relative to other toolbars.
- You can horizontally dock toolbars to the top or bottom of the frame window.

  In the Dreamweaver 4 workspace, which refers to the traditional or classic look of the Dreamweaver interface, where the user manages separate, floating windows, toolbars dock inside the document window. In classic mode, each window has its own set of toolbars. If you undock a toolbar, it is visible only when its document is in front.

  In the Dreamweaver MX workspace (also known as multiple document interface [MDI] mode*),* which integrates all the Dreamweaver document windows within a single parent frame, you can specify whether toolbars dock to the Dreamweaver MX workspace frame or to the document window.

  For toolbars that dock to the Dreamweaver MX workspace frame, there is only one instance of each toolbar. In this case, the toolbars always operate on the document in front. In the Dreamweaver MX workspace, you can dock toolbars above, below, or to the left or right of the Insert toolbar. Toolbars that are attached to the Dreamweaver MX workspace frame do not automatically disable when there is no document window. The toolbar items determine whether they are enabled when no document is open.

  Toolbars that stay docked to the document window work the same as toolbars in the Dreamweaver 4 workspace. There is one instance for each window. Toolbars that are attached to a document window, in either the Dreamweaver 4 workspace or the Dreamweaver MX workspace, completely disable themselves when their window is not the front document, and re-run all their update handlers when their window comes to the front.

  You cannot drag and drop toolbars between the document window and the Dreamweaver MX workspace frame.
- If you switch between the Dreamweaver 4 workspace and the Dreamweaver MX workspace, all toolbars revert to their default positions.
- Toolbars remain a fixed size. A toolbar does not shrink if the container shrinks or if other toolbars are placed next to it.
- You can show or hide toolbars from the View >Toolbars menu.
- Toolbars cannot overlap.
- Only the outline of the toolbar appears while you are dragging it.

On the Macintosh, toolbars are always attached to the document window. They can be shown or hidden from the menu, but you cannot drag and drop them, rearrange them, or undock them.

How toolbar commands work

When Dreamweaver draws a toolbar, the following events occur:

1. For each toolbar control item, Dreamweaver determines whether the `file` attribute exists.
2. If the `file` attribute exists, Dreamweaver calls `canAcceptCommand()` to determine whether it should enable the control in the current context of the document.

   For the Document Title text box in the Dreamweaver toolbar, for example, `canAcceptCommand()` checks to see if there is a current DOM and if the current document is an HTML file. If both these conditions are true, the function returns `true` and Dreamweaver enables the text box on the toolbar.
3. If the `file` attribute exists, Dreamweaver ignores the following attributes, if they are specified: `checked`, `command`, `DOMRequired`, `enabled`, `script`, `showif`, `update`, and `value`.
4. If the `file` attribute does not exist, Dreamweaver processes the attributes that are set for the toolbar control item: `checked`, `command`, `DomRequired`, and so on.

   For more information on specific item tag attributes, see "Item Tag Attributes" on page 88.
5. Dreamweaver calls the `getCurrentValue()` function on every update cycle, as specified by the `update` attribute, to determine what value to display for the control.
6. The user selects an item on the toolbar.
7. Dreamweaver calls the `receiveArguments()` function to process any arguments that are specified by the `arguments` attribute of the toolbar item.

For more information on the purpose of specific functions in the Toolbar Command API, see "The Toolbar Command API" on page 93.

The toolbar definition file

A toolbar is simply a list of toolbar items, optionally separated by separators. Each toolbar item can be either a reference to an item using the `itemref` tag, a separator using the `separator` tag, or a complete toolbar item definition, as described in "Toolbar item tags" on page 83.

Each toolbar definition file starts with the following declarations:

```
<?xml version="1.0" encoding="optional_encoding"?>
<!DOCTYPE toolbarset SYSTEM "-//Macromedia//DWExtension toolbar 5.0">
```

If the encoding is omitted, Dreamweaver defaults to the default encoding of the operating system.

After the declarations, the file consists of a single `toolbarset` tag, which contains any number of the following tags: `toolbar`, `itemref`, `separator`, `include`, and *`itemtype`* tags, where *`itemtype`* is a `button`, `checkbutton`, `radiobutton`, `menubutton`, `dropdown`, `combobox`, `editcontrol`, or `colorpicker`. The following example, which is an abbreviated excerpt from the toolbars.xml file, illustrates the hierarchy of tags in the toolbar file. The example substitutes ellipses (. . .) for the toolbar item attributes that are described in the following sections.

```
<?xml version="1.0"?>
<!DOCTYPE toolbarset SYSTEM "-//Macromedia//DWExtension toolbar 5.0">
<toolbarset>

<!-- main toolbar -->
  <toolbar id="DW_Toolbar_Main" label="Document">
    <radiobutton id="DW_CodeView" . . ./>
    <radiobutton id="DW_SplitView" . . ./>
    <radiobutton id="DW_DesignView" . . ./>
    <separator/>
    <checkbutton id="DW_LiveDebug" . . ./>
    <checkbutton id="DW_LiveDataView" . . ./>
    <separator/>
    <editcontrol id="DW_SetTitle" . . ./>
    <menubutton id="DW_FileTransfer" . . ./>
    <menubutton id="DW_Preview" , , ,/>
    <separator/>
    <button id="DW_DocRefresh" . . ./>
    <button id="DW_Reference" . . ./>
    <menubutton id="DW_CodeNav" . . ./>
    <menubutton id="DW_ViewOptions" . . ./>
  </toolbar>
</toolbarset>
```

The following section describes each of the toolbar tags.

‹toolbar›

Description

Defines a toolbar. Dreamweaver displays the items and separators from left to right in the specified order, laying out items automatically. The toolbar file does not specify control over the spacing between the items, but you can specify the widths of certain kinds of items.

Attributes

`id, label, {container}, {initiallyVisible}, {initialPosition}, {relativeTo}`

`id="`*`unique_id`*`"` Required. An identifier string must be unique within a given file; this also applies to all files that are included by that file. The JavaScript API functions that manipulate a toolbar refer to it by its ID. For more information on these functions, see “Toolbar functions” on page 637. If two toolbars that are included in the same file have the same ID, Dreamweaver displays an error.

`label="string"` Required. The name of the toolbar that Dreamweaver displays to the user. The label appears in the View >Toolbars menu and in the title bar of the toolbar when it’s floating.

`container="mainframe"` or `"document"` Defaults to `"mainframe"`. Specifies where the toolbar should dock in the Dreamweaver MX workspace on Windows. If set to `"mainframe"`, the toolbar appears in the outer Dreamweaver MX workspace frame and operates on the front document. If it is set to `"document"`, the toolbar appears in each document window. In the Dreamweaver 4 workspace and on the Macintosh, all toolbars appear in each document window.

`initiallyVisible="true"` or `"false"`. Specifies whether the toolbar should be visible the first time Dreamweaver loads it from the Toolbars folder. After the first time, the user controls visibility. Dreamweaver saves the current state to the system registry (Windows) or the Dreamweaver MX Preferences file (Macintosh) when the user quits Dreamweaver. Dreamweaver restores the setting from the registry or the Preferences file when it restarts. You can manipulate toolbar visibility using the `dom.getToolbarVisibility()` and `dom.setToolbarVisibility()` functions, as described in "Toolbar functions" on page 637. If you do not set the `initiallyVisible` attribute, it defaults to `true`.

`initialPosition="top"`, `"below"`, or `"floating"`. Specifies where Dreamweaver initially positions the toolbar, relative to other toolbars, the first time that Dreamweaver loads it. The possible values for `intialPosition` are described in the following list:

`top` This is the default. The toolbar appears at the top of the document window. If multiple toolbars specify `top` for a given window type, the toolbars appear in the order that Dreamweaver encounters them during loading, which might not be predictable, if the toolbars reside in separate files.

`below` The toolbar appears at the beginning of the row immediately below the toolbar that is specified in the `relativeTo` attribute. Dreamweaver reports an error if the `relativeTo` toolbar isn't found. If multiple toolbars specify `below` relative to the same toolbar, they appear in the order that Dreamweaver encounters them during loading, which might not be predictable if the toolbars reside in separate files.

`floating` Toolbar is not be initially docked to the window; it floats above the document. Dreamweaver automatically places the toolbar so it is offset from other floating toolbars. On the Macintosh, `floating` is treated the same as `top`.

As with `initiallyVisible`, this attribute applies only the first time that Dreamweaver loads the toolbar. After that, the toolbar's position is saved to the registry or the Dreamweaver MX Preferences file. You can reset the position of the toolbar by using the `dom.setToolbarPosition()` function. For more information on `dom.setToolbarPosition()`, see "dom.setToolbarPosition()" on page 638.

If you do not specify `initialPosition`, Dreamweaver positions the toolbar in the order that it is encountered during loading.

`relativeTo="`*`toolbar_id`*`"` Required if `initialPosition` specifies `below`. Otherwise, it is ignored. Specifies the ID of the toolbar below which this toolbar should be positioned.

Contents

Contains `include` tags, `itemref` tags, `separator` tags, and individual item definitions such as `button`, `combobox`, `dropdown`, and so on. For descriptions of the item definitions that you can specify, see "Toolbar item tags" on page 83.

Container

The `toolbarset` tag.

Example

```
<toolbar id="MyDWedit_toolbar" label="Edit">
```

<include/>

Description

Loads toolbar items from the specified file before continuing to load the current file. Toolbar items that are defined in the included file can be referenced in the current file. If a file attempts to recursively include another file, Dreamweaver displays an error message and ignores the recursive include. Any `toolbar` tags in the included file are skipped, although toolbar items in those toolbars are available for reference in the current file.

Attributes

`file` The pathname, relative to the Toolbars folder, of the toolbar XML file to include.

Contents

None.

Container

The `toolbar` tag or the `toolbarset` tag.

Example

```
<include file="mine/editbar.xml"/>
```

<itemtype/>

Description

Defines a single toolbar item. Toolbar items include buttons, radio buttons, check buttons, combo boxes, pop-up menus, and so on. For a list of the types of toolbar items that you can define, see “Toolbar item tags” on page 83.

Attributes

The attributes vary, depending on the item you are defining. For a complete list of the attributes that you can specify for toolbar items, see “Item Tag Attributes” on page 88.

Contents

None.

Container

The `toolbar` tag or the `toolbarset` tag.

Example

```
<button id="strikeout_button" .../>
```

<itemref/>

Description

The `itemref` tag refers to (and includes in the current toolbar) a toolbar item that was defined either inside a previous toolbar or outside of all toolbars.

Attributes

`id, {showIf}`

`id="`*`id_reference`*`"` Required. Must be the ID of an item that was previously defined or included in the file. Dreamweaver does not allow forward references. If a toolbar item tag references an ID that hasn't been defined, Dreamweaver reports an error and ignores the reference.

`showIf="script"` Specifies that this item appears only on the toolbar if the specified script returns `true`. For example, you can use `showIf` to show certain buttons only in a given application or only when a page is written in a server-side language such as ColdFusion, ASP, or JSP. If you do not specify `showIf`, the item always appears. Dreamweaver checks this property whenever the item's enabler runs; that is, according to the value of the `update` attribute. You should use this attribute sparingly. The `showIf` attribute can be used either in the item definition or in a reference to the item from a toolbar. If both the definition and the reference specify the `showIf` attribute, Dreamweaver shows the item only if both conditions are true. The `showIf` attribute is equivalent to the `showIf()` function in a command file.

Contents

None.

Container

The `toolbar` tag or the `toolbarset` tag.

Example

```
<itemref id="strikeout_button">
```

<separator/>

Description

Inserts a separator at the current location in the toolbar.

Attributes

`{showIf}`

`showif` Specifies that the separator should appear only on the toolbar if the given script returns `true`. For example, you can use `showIf` to show the separator only in a given application or only when the page has a certain document type. If unspecified, the separator always appears.

Contents

None.

Container

The `toolbar` tag.

Example

```
<separator/>
```

Toolbar item tags

Each type of toolbar item has its own tag and its own set of required and optional attributes. You can define `toolbar` items either inside or outside of toolbars. In general, it is better to define them outside of toolbars and refer to them within toolbars using the `itemref` tag.

You can define the following types of items in a toolbar.

<button>

Description

A pushbutton that executes a specific command when pressed. Looks and acts the same as the Reference button on the Dreamweaver toolbar.

Attributes

```
id, image, tooltip, command, {showif}, {disabledImage}, {overimage}, {label},
  {file}, {domRequired}, {enabled}, {update}, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<BUTTON ID="DW_DocRefresh"
  image="Toolbars/images/MM/refresh.gif"
  disabledImage="Toolbars/images/MM/refresh_dis.gif"
  tooltip="Refresh Design View (F5)"
  enabled="((dw.getDocumentDOM() != null) && (dw.getDocumentDOM().getView() !=
  'browse') && (!dw.getDocumentDOM().isDesignViewUpdated()))"
  command="dw.getDocumentDOM().synchronizeDocument()"
  update="onViewChange,onCodeViewSyncChange"/>
```

<checkbutton>

Description

A button that has a checked or unchecked state and that executes a specific command when pressed. When it is checked, it appears pressed in and highlighted. When it is not checked, it appears flat. Dreamweaver implements the following states for the check button: mouse-over, pressed, mouse-over-while-pressed, and disabled-while-pressed. The handler that is specified by the `checked` attribute or the `isCommandChecked()` function must ensure that clicking the check button causes the button's state to toggle.

Attributes

```
id, {showif}, image, {disabledImage}, {overimage}, tooltip, {label}, {file},
  {domRequired}, {enabled}, checked, {update}, command, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<CHECKBUTTON ID="DW_LiveDebug"
  image="Toolbars/images/MM/debugview.gif"
  disabledImage="Toolbars/images/MM/globe_dis.gif"
  tooltip="Live Debug"
  enabled="dw.canLiveDebug()"
  checked="dw.getDocumentDOM() != null && dw.getDocumentDOM().getView() ==
  'browse'"
  command="dw.toggleLiveDebug()"
  showIf="dw.canLiveDebug()"
  update="onViewChange"/>
```

<radiobutton>

Description

A radio button is exactly the same as a check button, except that when it is off, it appears as a raised button. Dreamweaver implements the following states for the radio button: mouse-over, pressed, mouse-over-while-pressed, and disabled-while-pressed. Dreamweaver does not enforce mutual exclusion between radio buttons. The handler that is specified by the `checked` attribute or the `isCommandChecked()` function must ensure that the checked and unchecked states of radio buttons are consistent with each other.

Radio buttons act the same as the Code view, Design view, and Split view buttons on the Dreamweaver document toolbar.

Attributes

```
id, image, tooltip, checked, command, {showif}, {disabledImage}, {overimage},
  {label}, {file}, {domRequired}, {enabled}, {update}, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<RADIOBUTTON ID="DW_CodeView"
  image="Toolbars/images/MM/codeView.gif"
  disabledImage="Toolbars/images/MM/codeView_dis.gif"
  tooltip="Show Code View"
  domRequired="false"
  enabled="dw.getDocumentDOM() != null"
  checked="dw.getDocumentDOM() != null && dw.getDocumentDOM().getView() ==
  'code'"
  command="dw.getDocumentDOM().setView('code')"
  update="onViewChange"/>
```

<menubutton>

Description

A button that pops up the context menu that is specified by the `menuid` attribute. Dreamweaver implements mouse-over and pressed states for menu buttons. Dreamweaver does not draw the menu arrow; you must include it in your icon.

Attributes

```
id, image, tooltip, menuID, domRequired, enabled, {showif}, {disabledImage},
  {overimage}, {label}, {file}, {update}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<MENUBUTTON ID="DW_CodeNav"
  image="Toolbars/images/MM/codenav.gif"
  disabledImage="Toolbars/images/MM/codenav_dis.gif"
  tooltip="Code Navigation"
  enabled="dw.getFocus() == 'textView' || dw.getFocus() == 'html'"
  menuID="DWCodeNavPopup"
  update="onViewChange"/>
```

<dropdown>

Description

A noneditable pop-up menu that executes a specific command when you choose an entry and it updates itself, based on an attached JavaScript function. It looks and acts the same as the Format control in the Text Property inspector, except it's a standard size instead of the small Property inspector size.

Attributes

```
id, tooltip, file, enabled, checked, value, command, {showif}, {label},
  {width}, {domRequired}, {update}, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<dropdown id="Font_Example"
  width="115"
  tooltip="Font"
  domRequired="false"
  file="Toolbars/mine/fontExample.htm"
  update="onSelChange"/>
```

<combobox>

Description

An editable pop-up menu that executes its command when you choose an entry or when the user makes an edit in the text box and switches focus. It looks and acts the same as the Font control on the Text Property inspector, except it's a standard size instead of the small Property inspector size.

Attributes

```
id, file, tooltip, enabled, value, command, {showif}, {label}, {width},
  {domRequired}, {update}, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<COMBOBOX ID="Address_URL"
  width="300"
  tooltip="Address"
  label="Address: "
  file="Toolbars/MM/AddressURL.htm"
  update="onBrowserPageBusyChange"/>
```

<editcontrol>

Description

A text editing box that executes its command when the user makes a change in the text box and switches focus.

Attributes

```
id, tooltip, file, value, command, {showif}, {label}, {width}, {domRequired},
  {enabled}, {update}, {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<EDITCONTROL ID="DW_SetTitle"
  label="Title: "
  tooltip="Document Title"
  width="150"
  file="Toolbars/MM/EditTitle.htm"/>
```

<colorpicker>

Description

A panel of colors, without an associated text box that executes its command when the user selects a new color. It looks and acts the same as the color picker on the Dreamweaver Property inspector. You can specify a different icon to replace the default icon.

Attributes

```
id, tooltip, value, command, {showif}, {image}, {disabledImage}, {overimage},
  {label}, {colorRect}, {file}, {domRequired}, {enabled}, {update},
  {arguments}
```

For a description of each attribute, see "Item Tag Attributes" on page 88.

Contents

None.

Container

Either the `toolbar` tag or the `toolbarset` tag.

Example

```
<colorpicker id="Color_Example"
  image="Toolbars/images/colorpickerIcon.gif"
  disabledImage="Toolbars/images/colorpickerIconD.gif"
  colorRect="0 12 16 16"
  tooltip="Text Color"
  domRequired="false"
  file="Toolbars/mine/colorExample.htm"
  update="onSelChange"/>
```

Item Tag Attributes

The attributes for toolbar item tags have the following meanings:

id="unique_id"

Required. The `id` is an identifier for the toolbar item. The `id` must be unique within the current file and all files that are included within the current file. The `itemref` tag uses the item `id` to refer to and include an item within a toolbar.

Example

```
<button id=”DW_DocRerefresh” . . . >
```

showIf="script"

Optional. Specifies that the item appears on the toolbar only if the script returns `true`. For example, you can use `showIf` to show certain buttons only when a page is written in a certain server-side language such as ColdFusion, ASP, or JSP. If you do not specify `showIf`, the item always appears.

The `showIf` attribute is checked whenever the item's enabler runs; that is, according to the value of the `update` attribute. You should use `showIf` sparingly.

You can specify the `showIf` attribute in the item definition and in a reference to the item on an `itemref` tag. If the definition and the reference specify the `showIf` attribute, the item shows only if both conditions are true. The `showIf` attribute is the same as the `showIf()` function in a toolbar command file. If you specify both the `showIf` attribute and the `showif()` function, `showIf()` overrides the attribute.

Example

```
showIf="dw.canLiveDebug()"
```

image="image_path"

Required for buttons, check buttons, radio buttons, menu buttons, and combo buttons. The `image` attribute is optional for color pickers and is ignored for other item types. The `image` attribute specifies the path, relative to the Configuration folder, of the icon file that displays on the button. The icon can be in any format that Dreamweaver can render, but typically it is a GIF or JPEG file format.

If an icon is specified for a color picker, the icon replaces the color picker entirely. If the `colorRect` attribute is also set, the current color appears on top of the icon in the specified rectangle.

Example

```
image="Toolbars/images/MM/codenav.gif"
```

disabledImage="image_path"

Optional. Dreamweaver ignores the `disabledImage` attribute for items other than buttons, check buttons, radio buttons, menu buttons, color pickers, and combo buttons. This attribute specifies the path, relative to the Configuration folder, of the icon file that Dreamweaver displays if the button is disabled. If you do not specify `disabledImage`, Dreamweaver displays the image that is specified in the `image` attribute when the button is disabled.

Example

```
disabledImage="Toolbars/images/MM/codenav_dis.gif"
```

overImage="image_path"

Optional. Dreamweaver ignores the `overImage` attribute for items other than buttons, check buttons, radio buttons, menu buttons, color pickers, and combo buttons. This attribute specifies the path, relative to the Configuration folder, of the icon file that Dreamweaver displays when the user moves the mouse over the button. If you do not specify `overImage`, the button does not change when the user moves the mouse over it, except for a ring that Dreamweaver draws around the button.

Example

```
overImage="Toolbars/images/MM/codenav_ovr.gif"
```

tooltip="tooltip string"

Required. Specifies the identifying text, or tooltip, that appears when the mouse cursor hovers over the toolbar item.

Example

```
tooltip="Code Navigation"
```

label="label string"

Optional. The `label` attribute specifies a label that displays next to the item. Dreamweaver does not automatically add a colon to labels. Labels for nonbutton items are always positioned on the left of the item. Dreamweaver places labels for buttons, check buttons, radio buttons, menu buttons, and combo buttons inside the button and to the right of the icon. Dreamweaver shows labels for buttons only if `Show Icon Labels` is checked on the View >Toolbars menu. Labels for other types of controls are always visible, regardless of whether this menu item is checked.

Example

```
label="Title: "
```

width="number"

Optional. The `width` attribute applies only to text box, pop-up menu, and combo box items. This attribute specifies the width of the item in pixels. If you do not specify the `width` attribute, Dreamweaver uses a reasonable default width.

Example

```
width="150"
```

menuID="menu_id"

Required for menu buttons and combo buttons, unless you specify `getMenuID()` in an associated command file. Dreamweaver ignores the `menuID` attribute for other types of items. This attribute specifies the ID of the menu bar that contains the context menu to pop up when the user presses the button, menu button, or combo button. The ID comes from the ID attribute of a `menubar` tag in menus.xml.

Example

```
menuID="DWCodeNavPopup"
```

colorRect="left top right bottom"

Optional for color pickers that have an image attribute. The `colorRect` attribute is ignored for other types of items and for color pickers that do not specify an image. If you specify the `colorRect` attribute, Dreamweaver displays the color that is currently selected in the color picker in the rectangle, relative to the left or top of the icon. If you do not specify the `colorRect` attribute, Dreamweaver does not display the current color on the image.

Example

```
colorRect=”0 12 16 16”
```

file="command_file_path"

Required for pop-up menus and combo boxes. The `file` attribute is optional for other types of items. The `file` attribute specifies the path, relative to the Configuration folder, of a command file that contains JavaScript functions to populate, update, and execute the item. The `file` attribute overrides the `enabled`, `checked`, `value`, `update`, `domRequired`, `menuID`, `showIf`, and `command` attributes. In general, if you specify a command file with the `file` attribute, Dreamweaver ignores all the equivalent attributes that are specified in the tag. For more information about command files, see “The Toolbar Command API” on page 93.

Example

```
file="Toolbars/MM/EditTitle.htm"
```

domRequired="true" or "false"

Optional. As with menus, the `domRequired` attribute specifies whether the Design view should be synchronized with the Code view before Dreamweaver runs the associated command. If you do not specify this attribute, it defaults to `true`. This attribute is equivalent to `isDOMRequired()` in a toolbar command file.

Example

```
domRequired="false"
```

enabled="script"

Optional. As with menus, the script returns a value that specifies whether the item is enabled. If you do not specify this attribute, it defaults to enabled. The `enabled` attribute is equivalent to `canAcceptCommand()` in a toolbar command file.

Example

```
enabled="dw.getFocus() == 'textView' || dw.getFocus() == 'html'"
```

checked="script"

Required for check buttons and radio buttons. Dreamweaver ignores the `checked` attribute for other types of items. As with menus, the script returns a value that specifies whether the item is checked or unchecked. The `checked` attribute is equivalent to `isCommandChecked()` in a toolbar command file. If you do not specify this attribute, it defaults to unchecked.

Example

```
checked="dw.getDocumentDOM() != null && dw.getDocumentDOM().getView() ==
  'code'"
```

value="script"

Required for pop-up menus, combo boxes, text boxes, and color pickers. Dreamweaver ignores the `value` attribute for other types of items.

To determine what value to display for pop-up menus and combo boxes, Dreamweaver first calls `isCommandchecked()` for each item in the menu. If `isCommandchecked()` returns `true` for any items, Dreamweaver displays the value for the first one. If no items return `true`, or `isCommandChecked()` is not defined, Dreamweaver calls `getCurrentValue()` or executes the script that the `value` attribute specifies. If the control is a combo box, Dreamweaver displays the returned value. If the control is a pop-up menu, Dreamweaver temporarily adds the returned value to the list and displays it.

In all other cases, the script returns the current value to display. For pop-up menus or combo boxes, this value should be one of the items in the menu list. For combo boxes and text boxes, the value can be any string that the script returns. For color pickers, the value should be a valid color but Dreamweaver does not enforce this.

The value attribute is equivalent to `getCurrentValue()` in a toolbar command file.

update="update_frequency_list"

Optional. Specifies how often the `enabled`, `checked`, `showif`, and `value` handlers should run to update the visible state of the item. The `update` attribute is equivalent to `getUpdateFrequency()` in a toolbar command file.

You must specify the update frequency for toolbar items because these items are always visible, unlike menu items. For this reason, you should always choose the lowest frequency possible and make sure your handlers for `enabled`, `checked`, and `value` are as simple as possible.

The following table lists the possible values for *update_frequency_list*, from least to most frequent. If you do not specify the update attribute, the update frequency defaults to onEdit frequency. You can specify multiple update frequencies, separated by commas. The handlers run on any of the specified events.

onServerModelChange executes when the server model of the current page changes.

onCodeViewSyncChange executes when the Code view becomes in or out of sync with the Design view.

onViewChange executes whenever the user switches focus between Code view and Design view or when the user changes between Code view, Design view, or Split view.

onEdit executes whenever the document is edited in Design view. Changes that you make in Code view do not trigger this event.

onSelChange executes whenever the selection changes in Design view. Changes that you make in Code view do not trigger this event.

onEveryIdle executes regularly when the application is idle. This can be very expensive, because this means the enabler/checked/showif/value handlers are running often. It should only be used for buttons that need to have their enable state changed at special times, and handlers should be quick.

Note: In all these cases, Dreamweaver actually executes the handlers after the specified event occurs, when the application is in a quiescent state. You are not guaranteed that your handlers will run after every edit or selection change; your handlers run "soon after" a batch of edits or selection changes occur. The handlers are guaranteed to run when the user clicks on a toolbar item.

Example

```
update="onViewChange"
```

command="script"

Required for all items except menu buttons. Dreamweaver ignores the command attribute for menu buttons. Specifies the JavaScript function to execute when the user performs one of the following actions:

- Clicks a button
- Selects an item from a pop-up menu or combo box
- Tabs out of, presses Return in, or clicks away from a text box or combo box
- Selects a color from a color picker

The command attribute is equivalent to the receiveArguments() function in a toolbar command file.

Example

```
command="dw.toggleLiveDebug()"
```

arguments="argument_list"

Optional. The `arguments` attribute specifies the comma-separated list of arguments to pass to the `receiveArguments()` function in a toolbar command file. If you do not specify the `arguments` attribute, Dreamweaver passes the ID of the toolbar item. In addition, pop-up menus, combo boxes, text boxes, and color pickers pass their current value as the first argument, before any arguments that the `arguments` attribute specifies, and before the item ID if no arguments are specified.

Example

On a toolbar with Undo and Redo buttons on it, each button calls the menu command file, Edit_Clipboard.htm, and passes an argument that specifies the action.

```
<button id="DW_Undo"
 image="Toolbars/images/MM/undo.gif"
 disabledImage="Toolbars/images/MM/undo_dis.gif"
 tooltip="Undo"
 file="Menus/MM/Edit_Clipboard.htm"
 arguments="'undo'"
 update="onEveryIdle"/>

<button id="DW_Redo"
 image="Toolbars/images/MM/redo.gif"
 disabledImage="Toolbars/images/MM/redo_dis.gif"
 tooltip="Redo"
 file="Menus/MM/Edit_Clipboard.htm"
 arguments="'redo'"
 update="onEveryIdle"/>
```

The Toolbar Command API

In many cases where you specify a script for an attribute, you can also implement the attribute through a JavaScript function in a command file. This is necessary when the functions need to take arguments, as in the command handler for a text box. It is required for pop-up menus and combo boxes.

The command file API for toolbar items is an extension of the menu command file API, so you can reuse menu command files directly as toolbar command files, perhaps with some additional functions that are specific to toolbars.

canAcceptCommand()

Description

Determines whether the toolbar item is enabled. The enabled state is the default condition for an item, so you should not define this function unless it returns `false` in at least one case.

Arguments

For pop-up menus, combo boxes, text boxes, and color pickers, the first argument is the current value within the control. The `getDynamicContent()` function can optionally attach individual IDs to items within a pop-up menu. If the selected item in the pop-up menu has an ID attached, Dreamweaver passes that ID to `canAcceptCommand()` instead of the value. For combo boxes, if the current contents of the text box do not match an entry in the pop-up menu, Dreamweaver passes the contents of the text box. Dreamweaver compares against the pop-up menu without case-sensitivity to determine whether the contents of the text box match an entry in the list.

If you specified the `arguments` attribute for this toolbar item in the toolbars.xml file, those arguments are passed next. If you did not specify the `arguments` attribute, Dreamweaver passes the ID of the item.

Returns

Dreamweaver expects a Boolean value that indicates whether the item is enabled.

Example

```
function canAcceptCommand()
{
  return (dw.getDocumentDOM() != null);
}
```

getCurrentValue()

Description

Returns the current value to display in the item. Dreamweaver calls `getCurrentValue()` for pop-up menus, combo boxes, text boxes, and color pickers. For pop-up menus, the current value should be one of the items in the menu. If the value is not in the pop-up menu, Dreamweaver selects the first item. For combo boxes and text boxes, this value can be any string that the function returns. For color pickers, the value should be a valid color, but Dreamweaver does not enforce this. This function is equivalent to the `value` attribute.

Arguments

None.

Returns

Dreamweaver expects a string that contains the current value to display. For the color picker, the string contains the RGB form of the selected color, for example "#FFFFFF" for the color white.

Example

```
function getCurrentValue()
{
  var title = "";
  var dom = dw.getDocumentDOM();
  if (dom)
    title = dom.getTitle();
  return title;
}
```

getDynamicContent()

Description

Required for pop-up menus and combo boxes. As with menus, this function returns an array of strings that populate the pop-up menu. Each string can optionally end with `";id=id"`. If an ID is specified, Dreamweaver passes the ID to the `receiveArguments()` function instead of the actual string to appear in the menu.

The name `getDynamicContent()` is a misnomer because this function should be used even if the list of entries in the menu is fixed. For example, the Menus/MM/Text_Size.htm file is not a dynamic menu; it is designed to be called from each one of a set of static menu items. By adding a `getDynamicContent()` function that simply returns the list of possible font sizes, however, the same command file can also be used for a toolbar pop-up menu. Toolbar items ignore underscores in the strings in a returned array so you can reuse menu command files. In the menu command file, Dreamweaver ignores the `getDynamicContent()` function because the menu item is not marked as dynamic.

Arguments

None.

Returns

Dreamweaver expects an array of strings with which to populate the menu.

Example

```
function getDynamicContent()
{
  var items = new Array;
  var filename = dw.getConfigurationPath() + "/Toolbars/MM/AddressList.xml";
  var location = MMNotes.localURLToFilePath(filename);
  if (DWfile.exists(location))
  {
    var addressData = DWfile.read(location);
    var addressDOM = dw.getDocumentDOM(dw.getConfigurationPath() +
      '/Shared/MM/Cache/empty.htm');
    addressDOM.documentElement.outerHTML = addressData;
    var addressNodes = addressDOM.getElementsByTagName("url");
    if (addressNodes.length)
    {
      for (var i=0; i < addressNodes.length ; i++ )
      {
        items[i] = addressNodes[i].address + ";id='" +
          addressNodes[i].address + "'";
      }
    }
  }
  return items;
```

getMenuID()

Description

Only valid for menu buttons. Dreamweaver calls `getMenuID()` to get the ID of the menu that should appear when the user clicks the button.

Arguments

None.

Returns

Dreamweaver expects a string that contains a menu ID, which is defined in `menus.xml`.

Example

```
function getMenuID()
{
  var dom = dw.getDocumentDOM();
  var menuID = '';
  if (dom)
  {
    var view = dom.getView();
    var focus = dw.getFocus();
    if (view == 'design')
    {
      menuID = 'DWDesignOnlyOptionsPopup';
    }
    else if (view == 'split')
    {
      if (focus == 'textView')
      {
        menuID = 'DWSplitCodeOptionsPopup';
      }
      else
      {
        menuID = 'DWSplitDesignOptionsPopup';
      }
    }
    else if (view == 'code')
    {
      menuID = 'DWCodeOnlyOptionsPopup';
    }
    else
    {
      menuID = 'DWBrowseOptionsPopup';
    }
  }
  return menuID;
}
```

getUpdateFrequency()

Description

Specifies how often to run the handlers for the `enabled`, `checked`, `showIf`, and `value` attributes to update the visible state of the item.

You must specify the update frequency for toolbar items because they are always visible, unlike menus. For this reason, you should always choose the lowest frequency possible and make sure your handlers for `enabled`, `checked`, and `value` are as simple as possible.

This function is equivalent to the `update` attribute in a toolbar item.

Arguments

None.

Returns

Dreamweaver expects a string that contains a comma-separated list of update handlers. For a complete list of the possible update handlers, see “update="update_frequency_list"” on page 91.

Example

```
function getUpdateFrequency()
{
  return onSelChange";
}
```

isCommandChecked()

Description

Returns a value that specifies whether the item is selected. For a button, checked means that the button appears on or depressed. The `isCommandChecked()` function is equivalent to the `checked` attribute in a toolbar item tag.

Arguments

For pop-up menus, combo boxes, text boxes, and color pickers, the first argument is the current value within the control. The `getDynamicContent()` command can optionally attach individual IDs to items within a pop-up menu. If the selected item in the menu has an ID attached, Dreamweaver passes that ID to `isCommandChecked()` instead of the value. For combo boxes, if the current contents of the text box do not match an entry in the pop-up menu, Dreamweaver passes the contents of the text box. For determining whether the text box matches, Dreamweaver compares against the menu without case-sensitivity.

If you specified the `arguments` attribute, those arguments are passed next. If you do not specify the `arguments` attribute, Dreamweaver passes the ID of the item.

Returns

Dreamweaver expects a Boolean value that indicates whether the item is checked.

Example

The following example determines which item, if any, should be checked in a pop-up menu of paragraph formats and CSS styles.

```
function isCommandChecked()
{
  var bChecked = false;
  var style = arguments[0];
  var textFormat = dw.getDocumentDOM().getTextFormat();

  if (dw.getDocumentDOM() == null)
    bChecked = false;

  if (style == "(None)")
    bChecked = (dw.cssStylePalette.getSelectedStyle() == '' || textFormat ==
"" || textFormat == "P" || textFormat == "PRE");
  else if (style == "Heading 1")
    bChecked =  (textFormat == "h1");
  else if (style == "Heading 2")
    bChecked =  (textFormat == "h2");
  else if (style == "Heading 3")
    bChecked =  (textFormat == "h3");
  else if (style == "Heading 4")
    bChecked =  (textFormat == "h4");
  else if (style == "Heading 5")
    bChecked =  (textFormat == "h5");
  else if (style == "Heading 6")
    bChecked =  (textFormat == "h6");
  else
    bChecked = (dw.cssStylePalette.getSelectedStyle() == style);

  return bChecked;
}
```

isDOMRequired()

Description

The `isDOMRequired()` function specifies whether the toolbar command requires a valid DOM to operate. If this function returns `true` or if the function is not defined, Dreamweaver assumes that the command requires a valid DOM and synchronizes the Code view and Design view for the document before executing the associated command. This function is equivalent to the `domRequired` attribute in a toolbar item tag.

Arguments

None.

Returns

Dreamweaver expects `true` if the DOM is required; `false` if the DOM is not required.

Example

```
function isDOMRequired()
{
  return false;
}
```

receiveArguments()

Description

Processes any arguments that are passed from a toolbar item. The `receiveArguments()` function is equivalent to the `command` attribute in a toolbar item tag.

Arguments

For pop-up menus, combo boxes, text boxes, and color pickers, the first argument is the current value within the control. The `getDynamicContent()` command can optionally attach individual IDs to items within a pop-up menu. If the selected item in the pop-up menu has an ID attached, Dreamweaver passes that ID to `receiveArguments()` instead of the value. For combo boxes, if the current contents of the text box do not match an entry in the pop-up menu, Dreamweaver passes the contents of the text box. To determine whether the text box matches, Dreamweaver compares against the pop-up menu without case-sensitivity.

If you specified the `arguments` attribute, those arguments are passed next. If you did not specify the `arguments` attribute, Dreamweaver passes the ID of the item.

Returns

Dreamweaver expects nothing.

Example

```
function receiveArguments(newTitle)
{
  var dom = dw.getDocumentDOM();
  if (dom)
    dom.setTitle(newTitle);
}
```

showIf()

Description

Specifies that an item appears on the toolbar only if the function returns `true`. For example, you could use `showIf()` to show certain buttons only when the page has a certain server model. If `showif()` is not defined, the item always appears. The `showIf()` function is the same as the `showIf` attribute in a toolbar item tag.

The `showIf()` function is called whenever the item's enabler runs; that is, according to the value that `getUpdateFrequency()` returns.

Arguments

None.

Returns

Dreamweaver expects a Boolean value that indicates whether the item will show.

Example

```
function showif()
{
     var retval = false;
     var dom = dw.getDocumentDOM();

     if(dom)
     {
       var view = dom.getView();
       if(view == 'design')
       {
         retval = true;
       }
     }
     return retval;
}
```

A simple toolbar command file

The following text box item lets the user enter a name for the current Dreamweaver document.

```
<EDITCONTROL ID="DW_SetTitle"
  label="Title: "
  tooltip="Document Title"
  width="150"
  file="Toolbars/MM/EditTitle.htm"/>
```

The `file` attribute in this text box item causes Dreamweaver to invoke the Toolbars/MM/EditTitle.htm command file when the user interacts with the text box.

```
<html>
<head>
<title>Edit Title</title>

<script language="JavaScript">
function receiveArguments(newTitle)
{
  var dom = dw.getDocumentDOM();
  if (dom)
    dom.setTitle(newTitle);
}

function canAcceptCommand()
{
  return (dw.getDocumentDOM() != null && dw.getDocumentDOM().getParseMode() ==
  'html');
}

function getCurrentValue()
{
  var title = "";
  var dom = dw.getDocumentDOM();
  if (dom)
    title = dom.getTitle();
  return title;
}
</script>
</head>

<body>
</body>
</html>
```

For the Document Title text box, the `getCurrentValue()` function calls the JavaScript API function `dom.getTitle()` to obtain and return the current title. Until the user enters a title for the document, the `getCurrentValue()` function returns "Untitled Document," which Dreamweaver displays in the text box. After the user enters a title, Dreamweaver displays the new document title.

Dreamweaver invokes the `receiveArguments()` function when the user enters a value in the Document Title text box and presses the Enter key or moves the focus away from the control. Dreamweaver passes `receiveArguments() newTitle`, which is the value that the user enters. The `receiveArguments()` function first checks to see if a current DOM exists. If it does, `receiveArguments()` sets the new document title by passing it to the `dom.setTitle()` function and then returning it to Dreamweaver.

CHAPTER 9
Reports

You can use the Reports API functions to create custom site reports or modify the set of prewritten reports that come with Dreamweaver. You can access site reports only through the Site Reports dialog box.

You can use the Results Window API to create a stand-alone report. Stand-alone reports are regular commands that directly use the Results Window API rather than the Reports API. You can access a stand-alone report the same way as any other command, through the menus or through another command.

Site reports reside in the Dreamweaver Configuration/Reports folder. The Reports folder has subfolders that represent report categories. Each report can belong to only one category. The category name cannot exceed 31 characters. Each subfolder can have a file in it named _foldername.txt. If this file is present, Dreamweaver uses its contents as the category name. If _foldername.txt is not present, Dreamweaver uses the folder name as the category name.

Stand-alone reports reside in the Dreamweaver Configuration/Commands folder.

When the user chooses multiple site reports from the Site Reports dialog box, Dreamweaver places all the results in the same results window under the Site Reports tab. Dreamweaver replaces these results the next time the user runs any site report.

In contrast, Dreamweaver creates a new Results window each time the user runs a new stand-alone report.

How site reports work

1. Reports are accessible through the Site > Reports... menu. When it is selected, this menu item displays a dialog box from which the user selects reports to run on a choice of targets.
2. The user selects which files to run the selected reports on using the Report On: menu. This menu contains Current Document, All Files in Current Local Site, Selected Files In Local Site, and Folder. When the user selects the Folder option, a Browse button and text field appear, so the user can select a folder.
3. The user can customize reports that have parameters by selecting the Settings button and entering values for the parameters. Each report is responsible for displaying its own Settings dialog box. This dialog box is optional; not every report requires the user to set the report's parameters. If a report does not have a Settings dialog box, then the Report Settings... button is dimmed when the report is selected in the list.

4 After the reports are selected and their settings are set, the user clicks the Run button.

   At this point, Dreamweaver clears all items from the Site Reports tab of the Results panel. Dreamweaver calls the `beginReporting()` function in each report before the reporting process begins. If a report returns `false` from this function, it is removed from the report run.

5 Each file is passed to each report that was selected in the Reports dialog box using the `processFile()` function. If the report needs to include information about this file in the results list, it should call the `dw.resultsPalette.siteReports.addResultItem()` function. This process continues until all files that pertain to the user's selection are processed, or the user clicks the Stop button in the bottom of the window. Dreamweaver displays the name of each file being processed and the number of files that remain to be processed.

   Dreamweaver calls the `endReporting()` function in each report after all the files have been processed and the reporting process completes.

How stand-alone reports work

1 The custom command opens a new results window by calling `dw.createResultsWindow` and storing the returned results object in a window variable. The remaining functions in this process should be called as methods of this object.

2 The custom command initializes the title and format of the Results window by calling `setTitle()` and `SetColumnWidths()` as methods of the results window object.

3 The command can either start adding items to the Results window immediately by calling `addItem()`, or it can begin iterating through a list of files by calling `setFileList()` and `startProcessing()` as methods of the Results window object.

4 When the command calls `resWin.startProcessing()`, Dreamweaver calls the function `processFile()` for each file URL in the list. Define the `processFile()` function in the stand-alone command. It receives the file URL as its only argument. Use the `setCallbackCommands()` function of the Results window object if you want Dreamweaver to call `processFile()` in some other command.

5 To call `addItem()`, the `processFile()` function needs to have access to the Results window that was created by the stand-alone command. The `processFile()` function can also call the `stopProcessing()` function of the Results window object to stop processing the list of files.

The Reports API

The only required function for the Reports API is the `processFile()` function. All other functions are optional.

processFile()

Description

Called when there is a file to process. The Report command should process the file without modifying it and use the `dw.ResultsPalette.SiteReports()` function, the `addResultItem()` function, or the `resWin.addItem()` function to return information about the file. Dreamweaver automatically releases each file's DOM when it is finished.

Arguments

`strFilePath`

`strFilePath` is the full path and filename of the file to process.

Returns

Dreamweaver expects nothing.

beginReporting()

Description

Called at the start of the reporting process, before any reports are run. If the Report command returns `false` from this function, the Report command is excluded from the report run.

Arguments

`target`

`target` is a string that indicates the target of the report session. It can be one of the following values: `"CurrentDoc"`, `"CurrentSite"`, `"CurrentSiteSelection"` (for the selected files in a site), or `"Folder:+ the path to the folder the user selected"` (for example, `"Folder:c:temp"`).

Returns

Dreamweaver expects `true` if the report runs successfully; `false` if *`target`* is excluded from the report run.

endReporting()

Description

Called at the end of the Report process.

Arguments

None.

Returns

Dreamweaver expects nothing.

commandButtons()

Description

Defines the buttons that should appear on the right side of the Options dialog box and their behavior when they are clicked. If this function is not defined, no buttons appear, and the `BODY` of the report file expands to fill the entire dialog box.

Arguments

None.

Returns

Dreamweaver expects an array that contains an even number of elements. The first element is a string that contains the label for the topmost button. The second element is a string of JavaScript code that defines the behavior of the topmost button when it is clicked. Remaining elements define additional buttons in the same manner.

Example

The following instance of `commandButtons()` defines three buttons: OK, Cancel, and Help.

```
function commandButtons(){
  return new Array("OK" , "doCommand()" , "Cancel" , ¬
  "window.close()" , "Help" , "showHelp()");
}
```

configureSettings()

Description

Determines whether the Report Settings button should be enabled in the Reports dialog box when this report is selected.

Arguments

None.

Returns

Dreamweaver expects `true` if the Report Settings button should be enabled; `false` otherwise.

windowDimensions()

Description

Sets specific dimensions for the Parameters dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an Options dialog box larger than 640 x 480 pixels.

Arguments

`platform`

The value of the argument is either `"macintosh"` or `"windows"`, depending on the user's platform.

Returns

Dreamweaver expects a string of the form `"widthInPixels,heightInPixels"`.

The returned dimensions are smaller than the size of the entire dialog box because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example

The following instance of `windowDimensions()` sets the dimensions of the Parameters dialog box to 648 x 520 pixels:

```
function windowDimensions(){
  return "648,520";
}
```

CHAPTER 10
Tag Libraries and Editors

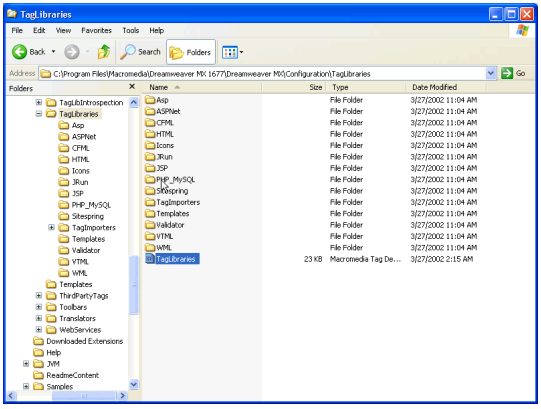
Dreamweaver MX users can use tag editors to insert new tags, edit existing tags, and access reference information about tags. Dreamweaver comes with editors for the following languages: HTML, ASP.Net, CFML, JRun, and JSP. You can customize tag editors that come with Dreamweaver, and you can create new tag editors. You can also add new tags to the Tag Libraries.

The Tag Chooser uses information that is stored in the Tag Libraries to let Dreamweaver users view available tags and select them to use in the active document.

Dreamweaver stores information about each tag, including all tag attributes, in a set of subfolders that reside in the Configuration/TagLibraries folder. The tag editor and Tag Chooser functions use the information that is stored in this folder when manipulating and editing tags. Before you can create custom tag editors, you should understand the Tag Library structure.

Tag Library file format

A Tag Library consists of a single root file, the TagLibraries.vtm file, that lists every installed tag, plus a .vtm (VTML) file for each tag in the Tag Library. The TagLibraries.vtm file functions as a table of contents and contains pointers to each individual tag's .vtm file. The following illustration shows how Dreamweaver organizes the .vtm files by markup language:



HomeSite users will recognize the .vtm file structure, but they should be aware that Dreamweaver does not use .vtm files in exactly the same way as HomeSite. The most important difference is that Dreamweaver contains its own HTML renderer that displays extension UIs, so the .vtm files are not used in the GUI rendering process.

The following example illustrates the structure of the TagLibraries.vtm file:

```
<taglibraries>
<taglibrary name="Name of tag library" doctypes="HTML,ASP-JS,ASP-VB"
  tagchooser="relative path to TagChooser.xml file" id="DWTagLibrary_html">
    <tagref name="tag name" file="relative path to tag .vtm file"/>
</taglibrary>

<taglibrary name="CFML Tags" doctypes="ColdFusion" servermodel="Cold Fusion"
  tagchooser="cfml/TagChooser.xml" id="DWTagLibrary_cfml">
    <tagref name="cfabort" file="cfml/cfabort.vtm"/>
</taglibrary>

<taglibrary name="ASP.NET Tags" doctypes="ASP.NET_CSharp,ASP.NET_VB"¬
  servermodel="ASPNet" prefix="<asp:" tagchooser="ASPNet/TagChooser.xml"¬
  id="DWTagLibrary_aspnet">
    <tagref name="dataset" file="aspnet/dataset.vtm" prefix="<mm:dataset"/>
</taglibrary>
</taglibraries>
```

The `taglibrary` tag groups one or more tags into a Tag Library. When you import tags or create a new set of tags, you can group them into Tag Libraries. Typically, a `taglibrary` grouping corresponds to a set of tags that are defined in a JavaServer Pages (JSP) TLD file, an XML document type definition (DTD) file, an ASP.Net name space, or some other logical grouping.

The following table lists `taglibrary` attributes:

Attribute	Description	Mandatory/ optional
`taglibary.name`	Used to refer to the Tag Library in the user interface.	Mandatory
`taglibrary.doctypes`	Indicates the document types for which this library is active. When active, library tags appear in the Code Hints pop-up menu. Not all Tag Libraries can be active at the same time because name conflicts can occur (for example, HTML and WML files are incompatible).	Mandatory
`taglibrary.prefix`	When specified, tags within the Tag Library have the form `taglibrary.prefix + tagref.name` For example, if the `taglibrary.prefix` is `"<jrun:"` and the `tagref.name` is `"if" then the tag is of the form "<jrun:if"`. This can be overridden for a particular tag. See the information on "`taglibrary.prefix`" on page 108 below.	Optional
`taglibrary.servermodel`	If the tags in the Tag Library execute on an application server, `servermodel` identifies the server model of the tag. If the tags are client-side tags (not server-side tags), the `servermodel` attribute is omitted. `servermodel` is also used for Check Target Browsers.	Optional
`taglibrary.id`	This can be any string that is different from the `taglibrary.ID` attributes of other Tag Libraries in the file. The ID attribute is used by the Extension Manager, so the MXP files can insert new `<taglibrary>` and the `tags` files into the TagLibraries.vtm file.	Optional
`taglibrary.tagchooser`	A relative file path to the TagChooser.xml file that is associated with this Tag Library.	Optional

The following table lists `tagref` attributes:

Attribute	Description	Mandatory/ optional
`tagref.name`	Used to refer to the tag in the user interface.	Mandatory
`tagref.prefix`	Specifies how the tag appears in Source view. When used, `tagref.prefix` determines the prefix of the current tag. When the attribute is defined, it overrides the value specified for `taglibrary.prefix`.	Optional
`tagref.file`	References the `VTML` file for the tag.	Optional

Because the `tagref.prefix` attribute can override `taglibrary.prefix`, the relationship between the two attributes can be confusing. The following table shows the relationship between the `taglibrary.prefix` and `tagref.prefix` attributes:

Is the `taglibrary.prefix` defined?	Is the `tagref.prefix` defined?	Resulting tag prefix
No	No	`'<' + tagref.name`
Yes	No	`taglibrary.prefix + tagref.name`
No	Yes	`tagref.prefix`
Yes	Yes	`tagref.prefix`

To define tags, Dreamweaver MX uses a modified version of Macromedia's `VTML` file format. The following example demonstrates all the elements that Dreamweaver MX must use to define an individual tag:

```
<tag name="input" bind="value" casesensitive="no" endtag="no">
  <tagformat indentcontents="yes" formatcontents="yes" nlbeforetag ¬
  nlbeforecontents=0 nlaftercontents=0 nlaftertag=1 />
  <tagdialog file = "input.HTM"/>
  <attributes>
    <attrib name="name"/>
    <attrib name="wrap" type="Enumerated">
       <attriboption value="off"/>
       <attriboption value="soft"/>
       <attriboption value="hard"/>
    </attrib>
    <attrib name="onFocus" casesensitive="yes"/>
    <event name="onFocus"/>
  </attributes>
</tag>
```

The following table lists the attributes that define tags:

Attribute	Description	Mandatory/ optional
`tag.bind`	Used by the Data Binding panel. When you select a tag of this type, the `BIND` attribute indicates the default attribute for data binding.	Optional
`tag.casesensitive`	Specifies whether the tag name is case-sensitive. If the tag is case-sensitive, it is inserted into the user's document using exactly the case that is specified in the Tag Library. If the tag is not case-sensitive, it is inserted using the default case that is specified in the Code Format tab of the Preferences dialog box. If `casesensitive` is omitted, the tag is assumed to be case-insensitive.	Optional
`tag.endtag`	Specifies whether the tag has both a beginning and an end tag. For example, `<input>` has no end tag; there is no matching `</input>` tag. If the end tag is optional, the `ENDTAG` attribute should be set to `Yes`.	Optional
`tagformat`	Specifies the tag's formatting rules. In Dreamweaver versions before Dreamweaver MX, these rules were stored in SourceFormat.txt.	Optional
`tagformat.indentcontents`	Specifies whether the contents of this tag should be indented.	Optional
`tagformat.formatcontents`	Specifies whether the contents of this tag should be parsed. This attribute is set to `No` for tags such as `<SCRIPT>` and `<STYLE>`, for which content is something other than HTML.	Optional
`tagformat.nlbeforetag`	The number of newline characters to insert before this tag.	Optional
`tagformat.nlbeforecontents`	The number of newline characters to insert before the contents of this tag.	Optional
`tagformat.nlaftercontents`	The number of newline characters to insert after the contents of this tag.	Optional
`tagformat.nlaftertag`	The number of newline characters to insert after this tag.	Optional
`attrib.name`	The name of the attribute, as it appears in the source code.	Mandatory

Attribute	Description	Mandatory/ optional
attrib.type	If omitted, attrib.type is "text". It can have the following values: TEXT–free text content ENUMERATED–a list of enumerated values COLOR–a color value (name or hex) FONT–font name or font family STYLE–CSS styles attribute FILEPATH –a full file path DIRECTORY–a directory path FILENAME–filename only RELATIVEPATH –a relative representation of the path FLAG –an ON/OFF attribute that contains no value	Optional
attrib.casesensitive	Specifies whether the attribute name is case-sensitive. If the CASESENSITIVE attribute is missing, the attribute name is case-insensitive.	Optional

Note: In versions before Dreamweaver MX, tag information is stored in the Configuration/TagAttributeList.txt file.

The Tag Chooser

The Tag Chooser lets the user view tags in functional groups so that they can easily access frequently used tags. In order to add a tag or a set of tags to the Tag Chooser, a user must add them to the Tag Library. This can be done using the tag library editor dialog box or by installing a Dreamweaver extension (an MXP file).

tagchooser.xml files

The tagchooser.xml file provides the metadata for organizing tag groupings that appear in the Tag Chooser. Each tag that comes with Dreamweaver is stored in a functional grouping and is available in the Tag Chooser. By editing the TagChooser.xml file, you can regroup existing tags and group new tags. You can customize how tags are organized for your users by creating subcategories so they can easily access their most important tags.

The TagLibraries.vtm file supports the use of the TAGLIBRARY.TAGCHOOSER attribute, which points to the tagchooser.xml file. If you change existing tagchooser.xml files or create new ones, the TAGLIBRARY.TAGCHOOSER attribute must point to the correct location for the Tag Chooser to be fully functional.

If there is no TAGLIBRARY.TAGCHOOSER attribute, the Tag Chooser displays the tree structure that is in the TagLibraries.vtm file.

TagChooser.xml files are stored in Configuration/TagLibraries/*TagLibraryName* folder. The following example illustrates the structure of TagChooser.xml files:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes" ?>
<tclibrary name="Friendly name for library node" desc='Description for
  incorporated reference' reference="Language[,Topic[,Subtopic]]">
  <category name="Friendly name for category node" desc='Description for
  incorporated reference' reference="Language[,Topic[,Subtopic]]"
  id="Unique id">
    <category name="Friendly name for subcategory node" ICON="Relative path"
  desc='Description for incorporated reference'
  reference="Language,Topic[,Subtopic]" id="Unique id">
      <element name="Friendly name for list item" value='Value to pass to
  visual dialog editors' desc='Description for incorporated reference'
  reference="Language[,Topic[,Subtopic]]" id="Unique id"/>
      ... more elements to display in the list view ...
    </category>
    ... more subcategories ...
  </category>
  ... more categories ...
</tclibrary>
```

The following table lists the tags that are available for use in the TagChooser.xml files:

Tag	Description	Mandatory/ Optional
`tclibrary`	The tag is the outermost tag, which encapsulates this Tag Library's Tag Chooser structure.	Mandatory
`tclibrary.name`	Value appears in the Tree view node.	Mandatory
`tclibrary.desc`	Value is an HTML string and is displayed in the Tag Info section of the Tag Chooser dialog box. If there is no `DESC` attribute, the information for Tag Info comes from the Reference panel. Interchangeable with `tclibrary.reference`.	Optional (`desc` and `reference` are mutually exclusive)
`tclibrary.reference`	Value describes the language, topic, and subtopic to display in the Tag Info section of the Tag Chooser dialog box. Interchangeable with `tclibrary.desc`.	Optional (`desc` and `reference` are mutually exclusive)

The `CATEGORY` tag represents all other nodes in the Tree view under the `TCLIBRARY`'s node, as shown in the following table:

Tag	Description	Mandatory/ Optional
`category.name`	Value appears in the tree view node.	Mandatory
`category.desc`	Value is an HTML string that appears in the tag info section of the Tag Chooser dialog box. If neither `desc` nor `reference attr` is specified, nothing appears in the Tag info section.	Optional (`desc` and `reference` are mutually exclusive)
`category.reference`	Value describes the language, topic, and subtopic to display in the Tag info section.	Optional (`desc` and `reference` are mutually exclusive)

Tag	Description	Mandatory/ Optional
category.icon	Value is a relative path to an icon GIF.	Optional
category.id	Any string that is different from the category.id attributes of other categories in this file.	Mandatory

The ELEMENT tag represents the tag to insert, with attributes as described in the following table:

Attribute	Description	Mandatory/ Optional
element.name	Value appears as a List view item.	Mandatory
element.value	Value that is either placed directly into the code or a parameter that passes into visual dialog editors.	Mandatory
element.desc	Value is an HTML string and appears in the incorporated Reference panel. If not specified, the REFERENCE attribute displays reference content in the incorporated Reference panel.	Optional (desc and reference are mutually exclusive)
element.reference	As many as three strings separated by commas that describes the language, topic, and subtopic respectively. This information appears in the Reference panel. The first string is mandatory. The second string is mandatory for the ELEMENT tag only; optional for CATEGORY and TCLIBRARY tags. The third string is optional.	Optional (desc and reference are mutually exclusive)
element.id	Any string that is different from the element.id attributes of other elements in this file.	Optional

Creating a new tag editor

The examples in this section use CFWEATHER, a hypothetical ColdFusion tag that was written to extract the current temperature from a weather database, to illustrate the steps necessary to create a new tag editor.

The attributes for CFWEATHER are as described in the following table:

Attribute	Description
zip	A five-digit ZIP code
tempaturescale	The temperature scale (Celsius or Farhenheit)

Registering the tag in the tag library

For Dreamweaver to recognize the new tag, it must be identified in the TagLibraries.vtm file, which is located in the Configuration/TagLibraries folder. However, if the user is on a system that supports multiple users (such as Windows XP, Windows 2000, or Mac OS X), the user has another TagLibraries.vtm file in their Configuration folder. This file is the one that needs to be updated because this file is the instance that Dreamweaver looks for and parses.

The location of the user's Configuration folder depends on the user's platform.

For Windows 2000 and Windows XP platforms:

```
<drive>:\Documents and Settings\<username>\ ¬
  Application Data\Macromedia\Dreamweaver MX\Configuration
```

For Windows NT platforms:

```
<drive>:\WinNT\profiles\<username>\ ¬
  Application Data\Macromedia\Dreamweaver MX\Configuration
```

For Mac OS X platforms:

```
<drive>:Users:<username>:Library:Application Support: ¬
  Macromedia:Dreamweaver MX:Configuration
```

If Dreamweaver MX cannot find TagLibraries.vtm in the user's Configuration folder, Dreamweaver looks for it in the Dreamweaver Configuration folder.

Note: On multiuser platforms, if you edit the copy of TagLibraries.vtm that resides in the Dreamweaver Configuration folder and not the one located in the user's Configuration folder, Dreamweaver is not aware of the changes because Dreamweaver parses the copy of TagLibraries.vtm in the user's Configuration folder, not in the Dreamweaver Configuration folder.

`cfweather` is a ColdFusion tag, so an appropriate Tag Library group already exists that you can use to register the `<cfweather>` tag.

To register the tag:

1. Open the TagLibraries.vtm file in a text editor.
2. Scroll through the existing Tag Libraries to find the CFML tags `<taglibrary>` group.
3. Add a new tag reference element, as shown in the following example:

   ```
   <tagref name="cfweather" file="cfml/cfweather.vtm"/>
   ```

4. Save the file.

   The tag is now registered in the tag library. It has a file pointer to the cfweather.vtm tag definition file.

Creating a tag definition (.vtm) file

When a user selects a registered tag using the Tag Chooser or a tag editor, Dreamweaver looks for a corresponding .vtm file for the tag definition.

To create a tag definition file:

1. In a text editor, create a file with the following contents:

   ```
   <TAG NAME="cfweather" endtag="no">
            <TAGFORMAT NLBEFORETAG="1" NLAFTERTAG="1"/>
            <TAGDIALOG FILE="cfweather.htm"/>

            <ATTRIBUTES>
              <ATTRIB NAME="zip" TYPE="TEXT"/>
              <ATTRIB NAME="tempaturescale" TYPE="ENUMERATED">
                <ATTRIBOPTION VALUE="Celsius"/>
                <ATTRIBOPTION VALUE="Fahrenheit"/>
              </ATTRIB>
            </ATTRIBUTES>
   </TAG>
   ```

2. Save the file as Configuration/Taglibraries/CFML/cfweather.vtm.

   Using the tag definition file, Dreamweaver can perform code hinting, code completion, and tag formatting functionality for the `<cfweather>` tag.

Creating a tag editor UI

To create the CFWEATHER tag editor user interface:

1 Save the following file as Configuration/Taglibraries/CFML/cfweather.htm:

```
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine 5.0//dialog">
<html>
<head>
<title>CFWEATHER</title>
<script src="../../Shared/Common/Scripts/dwscripts.js"></script>
<script src="../../Shared/Common/Scripts/ListControlClass.js"></script>
<script src="../../Shared/Common/Scripts/tagDialogsCmn.js"></script>
<script>

/************************ GLOBAL VARS ************************/
var TEMPATURESCALELIST;   // tempaurelist control (initialized in
  initializeUI())
var theUIObjects;         // array of UI objects used by common API functions

/*************************************************************/

// inspectTag() API function defined (required by all tag editors)
function inspectTag(tagNodeObj)
{
  // call into a common library version of inspectTagCommon defined
  // in tagDialogCmns.js (note that it's been included)
  // For more information about this function, look at the comments
  // for inspectTagCommon in tagDialogCmn.js
  tagDialog.inspectTagCommon(tagNodeObj, theUIObjects);
}

function applyTag(tagNodeObj)
{
  // call into a common library version of applyTagCommon defined
  // in tagDialogCmns.js (note that it's been included)
  // For more information about this function, look at the comments
  // for applyTagCommon in tagDialogCmn.js
  tagDialog.applyTagCommon(tagNodeObj, theUIObjects);
}

function initializeUI()
{
  // define two arrays for the values and display captions for the list control
  var theTempatureScaleCap = new Array("celsius","fahrenheit");
  var theTempatureScaleVal = new Array("celsius","fahrenheit");

  // instantiate a new list control
  TEMPATURESCALELIST = new ListControl("thetempaturescale");

  // add the tempaturescalelist dropdown list control to the uiobjects
  theUIObjects  = new Array(TEMPATURESCALELIST);

  // call common populateDropDownList function defined in tagDialogCmn.js to
  // populate the tempaturescale list control
  tagDialog.populateDropDownList(TEMPATURESCALELIST, theTempatureScaleCap,
  theTempatureScaleVal, 1);
}
</script>

</head>
<body onLoad="initializeUI()">
```

```
<div name="General">
  <table border="0" cellspacing="4">
    <tr>
      <td valign="baseline" align="right" nowrap="nowrap">Zip Code: </td>
      <td nowrap="nowrap">
       <input type="text" id="attr:cfargument:zip" name="thezip" attname="zip"
  style="width:100px"  /> 
      </td>
    </tr>
    <tr>
      <td valign="baseline" align="right" nowrap="nowrap">Type: </td>
      <td nowrap="nowrap">
          <select name="thetempaturescale" id="attr:cfargument:tempaturescale"
  attname="tempaturescale" editable="false" style="width:200px">
          </select>
      </td>
    </tr>
  </table>
</div>
</body>
</html>
```

2 Verify that the tag editor is working by performing the following steps:

- Launch Dreamweaver MX.
- Type `<cfweather>` in Code view.
- Right click on the tag.
- Select Edit Tag `<cfweather>` from the Context menu.

  If the tag editor launches, it has been created successfully.

Adding a tag to Tag Chooser

To add the `CFWEATHER` tag to the Tag Chooser:

1 Modify the Configuration/Taglibraries/CFML/tagchooser.xml file by adding a new category called Third Party Tags, which features the `<cfweather>` tag, as shown in the following example:

```
<category name="Third Party Tags" icon="icons/Elements.gif"
  reference='CFML'>
  <element name="cfweather" value='cfweather zip=""
  temperaturescale="fahrenheit">' />
</category>
```

Note: On multiuser platforms, the tagchooser.xml file also exists in the user's Configuration folder. For more information regarding multiuser platforms, see the discussion in "Registering the tag in the tag library" on page 113.

2 Verify the `<cfweather>` tag now appears in the Tag Chooser by performing the following steps:

- Select Insert > Tag.
- Expand the CFML Tags group.
- Select the Third Party Tags group that appears at the bottom of the Tag Chooser.

- The `<cfweather>` tag appears in the list box on the right.
- Select cfweather, and click the Insert button.

  The tag editor should appear.

Tag editor APIs

In order to create a new tag editor, you must provide an implementation for the three functions `inspectTag()`, `validateTag()`, and `applyTag()`. For an example of an implementation, see "Creating a tag editor UI" on page 115.

inspectTag()

Availability

Dreamweaver MX

Description

When the tag editor first pops up, the function is called. The function is passed the tag that the user is editing, which is expressed as a `dom` object. The function extracts attribute values from the tag that is being edited and uses these values to initialize form elements in the tag editor.

Arguments

Accepts `dom` node of the edited tag.

Returns

Dreamweaver expects nothing.

Example

Suppose the user is editing the following tag:

```
<crfweather zip = "94065"/>
```

If the editor contains a text field for editing the `zip` attribute, the function needs to initialize the form element so that the user sees the actual ZIP code in the text field, rather than an empty field.

The following code performs the initialization:

```
function inspectTag(tag)
{
  document.forms[0].zip.value = tag.zip
}
```

validateTag()

Availability

Dreamweaver MX

Description

When user clicks on a node in the tree control or clicks OK, the function performs input validation on the currently displayed HTML form elements.

Arguments

None.

Returns

Dreamweaver expects a Boolean value: `true` if the input for HTML form elements is valid; `false` if input values are not valid.

Example

While the user creates a table, a negative integer is entered for the number of table rows. `validateTag()` detects the invalid input, pops up an alert message, and returns `false`.

applyTag()

Availability

Dreamweaver MX

Description

When the user clicks OK, Dreamweaver calls `validateTag()`. If `validateTag()` returns `true`, Dreamweaver calls this function and passes the `dom` object that represents the current tag (the tag that is being edited). The function reads the values out of the form elements and writes them into the `dom` object.

Arguments

Accepts the `dom` node of the tag being edited.

Returns

Dreamweaver expects nothing.

Example

Continuing the `cfweather` example, if the user changes the zip from 94065 to 53402, in order to update the user's document to use the new ZIP code, the `dom` object must be updated:

```
function applyTag(tag)
{
  tag.zip = document.forms[0].zip.value
}
```

CHAPTER 11
Property Inspectors

The Property inspector is perhaps the most familiar floating panel in the Dreamweaver interface. It is indispensable for defining, reviewing, and changing the name, size, appearance, andother attributes of the selection as well as for launching internal and external editors for the selected element.

Dreamweaver has several built-in interfaces for the Property inspector that let you set properties for many standard HTML tags. These built-in inspectors are part of the core Dreamweaver code; for this reason, you cannot find corresponding Property inspector files for them in the Configuration folder. With custom Property inspector files, you can override these built-in interfaces or create new ones to inspect custom tags.

Custom Property inspector files are HTML files that reside in the Configuration/Inspectors folder inside the Dreamweaver application folder. Property inspector files must contain a comment (in addition to the doctype comment for Dreamweaver MX) immediately preceding the opening HTML tag in the following format:

```
<!-- tag:tagNameOrKeyword,priority:1to10,selection:¬
exactOrWithin,hline,vline, serverModel-->
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine5.0//pi">
```

where:

- *tagNameOrKeyword* is the tag to be inspected or one of the following keywords: `*COMMENT*` (for comments), `*LOCKED*` (for locked regions), or `*ASP*` (for `ASP` tags).
- *1to10* is the priority of the Property inspector file: `1` indicates that this inspector should be used only when no others can inspect the selection; `10` indicates that this inspector takes precedence over all others that can inspect the selection.
- *exactOrWithin* indicates whether the selection can be within the tag (`within`) or must exactly contain the tag (`exact`).
- `hline` (optional) indicates that a horizontal gray line should appear between the upper and lower halves of the inspector in expanded mode.
- `vline` (optional) indicates that a vertical gray line should appear between the tag name field and the rest of the properties in the inspector (see an HTML file in the Configuration/ Inspectors folder for an example).
- `serverModel` (optional) indicates the server model of the Property inspector. If the server model of the Property inspector is not the same as the server model for the document, the Property inspector is not used to display the properties of the current selection.

The following comment is appropriate for an inspector that is designed to inspect the `HAPPY` tag:

```
<!-- tag:HAPPY, priority:8,selection:exact,hline,vline, ¬
serverModel:ASP -->
```

In some cases, you might want to specify that your extension use only Dreamweaver MX extension rendering (and not the previous rendering engine) by inserting the following line immediately before the Tag comment, as shown in the following example:

```
<!--DOCTYPE HTML SYSTEM "-//Macromedia//DWEtension layout-engine 5.0//pi"-->
```

The `BODY` of a Property inspector file contains an HTML form. Instead of displaying the form contents in a dialog box, however, Dreamweaver uses the form to define the input areas and layout of the inspector.

How Property inspector files work

At start up, Dreamweaver reads the first line of each .htm and .html file in the Configuration/Inspectors folder, looking for the comment string that defines the type, priority, and selection type of a Property inspector. Files that do not have this comment as their first line are ignored.

When the user makes a selection in Dreamweaver or moves the insertion point to a different location, the following events occur:

1. Dreamweaver looks for any inspectors that have a `within` selection type.
2. If there are any `within` inspectors, Dreamweaver searches up the document tree from the currently selected tag to check whether there are inspectors for any tags that surround the selection. If—and only if—there are no `within` inspectors, Dreamweaver looks for any inspectors that have a selection type of `exact`.
3. For the first tag found that has one or more inspectors, Dreamweaver calls each inspector's `canInspectSelection()` function. If this function returns `false`, Dreamweaver no longer considers the inspector a candidate for inspecting the selection.
4. If more than one potential inspector remains after calling `canInspectSelection()`, Dreamweaver sorts the remaining inspectors by priority.
5. If more than one potential inspector shares the same priority, Dreamweaver selects an inspector alphabetically by name.
6. The selected inspector appears in the Property inspector floating panel. If the Property inspector file defines the `displayHelp()` function, a small question mark (?) icon appears in the upper-right corner of the inspector.
7. Dreamweaver calls the `inspectSelection()` function to gather information about the current selection and populate the inspector's fields.
8. Event handlers attached to the fields in the Property inspector interface execute as the user encounters them. (For example, you might have an `onBlur` event that calls `setAttribute()` to set an attribute to the value that the user entered.)

The Property inspector API

Two of the Property inspector API functions (`canInspectSelection()` and `inspectSelection()`) are required.

canInspectSelection()

Description

Determines whether the Property inspector is appropriate for the current selection.

Arguments

None.

Use "dom.getSelectedNode()" on page 546 to get the current selection as a JavaScript object.

Returns

Dreamweaver expects `true` if the inspector can inspect the current selection; `false` otherwise.

Example

The following instance of `canInspectSelection()` returns `true` if the selection contains the `CLASSID` attribute, and the value of that attribute is `"clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"` (the class ID for Flash Player):

```
function canInspectSelection(){
  var theDOM = dw.getDocumentDOM();
  var theObj = theDOM.getSelectedNode();
  return (theObj.nodeType == Node.ELEMENT_NODE && ¬
  theObj.hasAttribute("classid") && ¬
  theObj.getAttribute("classid").toLowerCase()== ¬
  "clsid:D27CDB6E-AE6D-11cf-96B8-444553540000");
}
```

displayHelp()

Description

If this function is defined, a question mark (?) icon appears in the upper-right corner of the Property inspector. This function is called when the user clicks the icon.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

The following example of `displayHelp()` opens a file in a browser window that explains the fields of the Property inspector:

```
function displayHelp(){
  dw.browseDocument('http://www.hooha.com/dw/inspectors/inspHelp.html');
}
```

inspectSelection()

Description

Refreshes the contents of the text fields based on the attributes of the current selection.

Arguments

maxOrMin

The argument is either `max` or `min`, depending on whether the Property inspector is in its expanded or contracted state.

Returns

Dreamweaver expects nothing.

Example

The following example of `inspectSelection()` gets the value of the `CONTENT` attribute and uses it to populate a form field called `keywords`:

```
function inspectSelection(){
  var dom = dreamweaver.getDocumentDOM();
  var theObj = dom.getSelectedNode();
  document.forms[0].keywords.value = ¬
  theObj.getAttribute("content");
}
```

A simple Property inspector example

The following Property inspector inspects a fictional tag called INTJ. The INTJ tag is empty (it has no closing tag), so its selection type is exact. As long as the selection is an INTJ tag, the Property inspector appears—so the canInspectSelection() function returns true every time. To have a different inspector appear, depending on the value of the INTJ tag's TYPE attribute, for example, the canInspectSelection() function must check the value of the TYPE attribute to determine which Property inspector is the right one. (This is how the keywords and description Property inspectors work, because "keywords" and "description" are not tags but values of the META tag's NAME attribute.)

```
<!-- tag:INTJ,priority:5,selection:exact,vline,hline -->
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine5.0//pi">
<HTML>
<HEAD>
<TITLE>Interjection Inspector</TITLE>
<SCRIPT LANGUAGE="JavaScript">

function canInspectSelection(){
  return true;
}

function inspectSelection(){
  // Get the DOM of the current document var
  // theDOM = dw.getDocumentDOM();
  // Get the selected node var theObj = theDOM.getSelectedNode();

  // Get the value of the TYPE attribute on the INTJ tag var
  // theType = theObj.getAttribute('type');
  // Initialize a variable called typeIndex to -1. This will be
  // used to store the menu index that corresponds to
  // the value of the TYPE attribute
  var typeIndex = -1;

  // If there was a TYPE attribute
  if (theType){
    // If the value of TYPE is "jeepers", set typeIndex to 0
    if (theType.toLowerCase() == "jeepers"){
      typeIndex = 0;
    // If the value of TYPE is "jinkies", set typeIndex to 1
    }else if (theType.toLowerCase() == "jinkies"){
      typeIndex = 1;
    // If the value of TYPE is "zoinks", set typeIndex to 2
    }else if (theType.toLowerCase() == "zoinks"){
      typeIndex = 2;
    }
  }

  // If the value of the TYPE attribute was "jeepers",
  // "jinkies", or "zoinks", choose the corresponding
  // option from the pop-up menu in the interface
  if (typeIndex != -1){
    document.topLayer.document.topLayerForm.intType.¬
    selectedIndex = typeIndex;
  }
}

function setInterjectionTag(){
  // Get the DOM of the current document
  var theDOM = dw.getDocumentDOM();
  // Get the selected node
```

```
  var theObj = theDOM.getSelectedNode();

  // Get the index of the selected option in the pop-up menu
  // in the interface
  var typeIndex = document.topLayer.document.¬
  topLayerForm.intType.selectedIndex;
  // Get the value of the selected option in the pop-up menu
  // in the interface
  var theType = document.topLayer.document.¬
  topLayerForm.intType.options[typeIndex].value;

  // Set the value of the TYPE attribute to theType
  theObj.setAttribute('type',theType);
}

</SCRIPT>
</HEAD>

<BODY>
<SPAN ID="image" STYLE="position:absolute; width:23px; ¬
height:17px; z-index:16; left: 3px; top: 2px">
<IMG SRC="interjection.gif" WIDTH="36" HEIGHT="36" ¬
NAME="interjectionImage">
</SPAN>
<SPAN ID="label" STYLE="position:absolute; width:23px; ¬
height:17px; z-index:16; left: 44px; top: 5px">Interjection</SPAN>

<!-- If your form fields are in different layers, you must ¬
create a separate form inside each layer and reference it as ¬
shown in the inspectSelection() and setInterjectionTag() ¬
functions above. -->

<SPAN ID="topLayer" STYLE="position:absolute; z-index:1; ¬
left: 125px; top: 3px; width: 431px; height: 32px">
<FORM NAME="topLayerForm">
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0">
<TR>
<TD VALIGN="baseline" ALIGN="right">Type:</TD>
<TD VALIGN="baseline" ALIGN="right">
<SELECT NAME="intType" STYLE="width:86" ¬
onChange="setInterjectionTag()">
<OPTION VALUE="jeepers">Jeepers</OPTION>
<OPTION VALUE="jinkies">Jinkies</OPTION>
<OPTION VALUE="zoinks">Zoinks</OPTION>
</SELECT>
</TR>
</TABLE>
</FORM>
</SPAN>

</BODY>
</HTML>
```

CHAPTER 12
Floating Panels

You can create any kind of floating panel or inspector without the size and layout limitations of Property inspectors.

Although a custom Property inspector should be your first choice for setting the properties of the current selection, custom floating panels offer more room and flexibility for displaying information about the entire document or multiple selections.

Custom Floating Panel files are HTML files that reside in the Configuration/Floaters folder inside the Dreamweaver application folder. The `BODY` of a Floating Panel file contains an HTML form; event handlers that are attached to form elements can call JavaScript code that performs arbitrary edits to the current document.

Dreamweaver has several built-in floating panels that are accessible from the Window menu. (These built-in panels are part of the core Dreamweaver code and do not have corresponding Floating Panel files for them in the Configuration/Floaters folder.)

You can create custom panels and add them to the Window menu. For more information on adding items to the menu system, see "Customizing Dreamweaver," in the Dreamweaver MX Support Center.

How floating panel files work

Custom floating panels can be moved, resized, and tabbed together the same way that the floating panels that are built into Dreamweaver. Custom floating panels differ from built-in floating panels in the following ways:

- It is not possible to display an icon in the tab of a custom floating panel; the tab always shows the contents of the floating panel's `TITLE` tag.
- Custom floating panels display in the default gray. Setting the `BGCOLOR` attribute in the `BODY` tag has no effect.
- All custom floating panels either appear always on top of the Document window or float behind it when inactive, depending on the setting for All Other Floaters in the Floating panels preferences.

Floating panel files also differ somewhat from other extensions. Unlike other extension files, Dreamweaver does not load floating panel files into memory at startup unless the floating panels were visible when Dreamweaver last shut down. If the floating panels were not visible when Dreamweaver last shut down, the files that define them are loaded only when referenced from one of the following functions: "dreamweaver.getFloaterVisibility()" on page 644, "dreamweaver.setFloaterVisibility()" on page 647, or "dreamweaver.toggleFloater()" on page 650.

When one of the files inside the Configuration folder calls `dw.getFloaterVisibility(floaterName)`, `dw.setFloaterVisibility(floaterName)`, or `dw.toggleFloater(floaterName)`, the following events occur:

1. If *floaterName* is not one of the reserved floating panel names, Dreamweaver searches the Configuration/Floaters folder for a file called *floaterName*.htm. (For a complete list of reserved floating panel names, see "dreamweaver.getFloaterVisibility()" on page 644.) If *floaterName*.htm is not found, Dreamweaver searches for *floaterName*.html. If no file is found, nothing happens.
2. If the Floating Panel file is being loaded for the first time, the `initialPosition()` function is called, if defined, to determine the floating panel's default position on the screen, and the `initialTabs()` function is called, if defined, to determine the floating panel's default tab grouping.
3. The `selectionChanged()` and `documentEdited()` functions are called on the assumption that changes probably occurred while the floating panel was hidden.
4. When the floating panel is visible, the following actions occur:

- When the selection changes, the `selectionChanged()` function is called, if it is defined.
- When the user makes changes to the document, the `documentEdited()` function is called, if it is defined.
- Event handlers that are attached to the fields in the floating panel interface execute as the user encounters them. (For example, a button with an `onClick` event handler that calls `dw.getDocumentDOM().body.innerHTML=''` removes everything between the opening and closing `BODY` tags in the document when it is clicked.)

5. When the user quits Dreamweaver, the current visibility, position, and tab grouping of the floating panel are saved. The next time Dreamweaver starts up, it loads the floating panel files for any floating panels that were visible at the last shutdown and displays the floating panels in their last position and tab grouping.

The Floating panel API

All the custom functions in the Floating panel API are optional.

Dreamweaver MX introduces a new user interface in Windows, known as the Dreamweaver MX workspace or multiple document interface (MDI). This interface, or type of workspace, is optional, but it is also the default workspace. In the Dreamweaver MX workspace, Dreamweaver MX integrates all documents into one parent container in which you can dock all objects and panels.

If you prefer, in Windows you can choose to work in the Dreamweaver 4 workspace, in which you manage separate, floating windows. The Dreamweaver 4 workspace is called the classic workspace.

In Windows, you can switch from one type of workspace to the other through the Preferences item on the Edit menu.

Some of the functions in this section operate only in the Dreamweaver MX workspace and only on the Windows operating system. The description of the function indicates whether this is the case.

displayHelp()

Description

If this function is defined, a Help button appears below the OK and Cancel buttons in your dialog box. This function is called when the user clicks the Help button.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

```
// the following instance of displayHelp() opens
// in a browser a file that explains how to use
// the extension.
function displayHelp(){
  var myHelpFile = dw.getConfigurationPath() +
    '/ExtensionsHelp/superDuperHelp.htm';
  dw.browseDocument(myHelpFile);
}
```

documentEdited()

Description

Called when the floating panel becomes visible and after the current series of edits is complete; that is, multiple edits might occur before this function is called. This function should be defined only if the floating panel must track edits to the document.

Note: Define `documentEdited()` only if you absolutely require it because its existence impacts performance.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

The following example of `documentEdited()` scans the document for layers and updates a text field that displays the number of layers in the document:

```
function documentEdited(){
  /* create a list of all the layers in the document */
  var theDOM = dw.getDocumentDOM();
  var layersInDoc = theDOM.getElementsByTagName("layer");
  var layerCount = layersInDoc.length;

  /* update the numOfLayers field with the new layer count */
  document.theForm.numOfLayers.value = layerCount;
}
```

getDockingSide()

Availability

Dreamweaver MX (Windows only)

Description

Specifies the locations at which a floating panel can dock. The function returns a string that contains some combination of the words `"left"`, `"right"`, `"top"`, and `"bottom"`. If the label is in the string, you can dock to that side. If the function is missing, you cannot dock to any side.

You can use this function to prevent certain panels from docking on a certain side of the Dreamweaver MX workspace or to each other.

Arguments

None.

Returns

Dreamweaver expects a string containing the words `"left"`, `"right"`, `"top"`, and `"bottom"`, or a combination of them, that specifies where Dreamweaver can dock the floating panel.

Example

```
getDockingSide()
{
  return dock_side = “left top”;
}
```

initialPosition()

Description

Determines the initial position of the floating panel the first time it is called. If this function is not defined, the default position is the center of the screen.

Arguments

`platform`

Possible values for *`platform`* are `"Mac"` and `"Win"`.

Returns

Dreamweaver expects a string of the form `"leftPosInPixels,topPosInPixels"`.

Example

The following example of `initialPosition()` specifies that the first time the floating panel appears, it should be 420 pixels from the left and 20 pixels from the top in Windows, and 390 pixels from the left side of the screen and 20 pixels from the top of the screen on the Macintosh:

```
function initialPosition(platform){
  var initPos = "420,20";
  if (platform == "macintosh"){
    initPos = "390,20";
  }
  return initPos;
}
```

initialTabs()

Description

Determines which other floating panels are tabbed together the first time that this floating panel appears. If any listed floating panel has appeared previously, it is not included in the tab group.To ensure that two custom floating panels are tabbed together, each panel should reference the other with the `initialTabs()` function.

Arguments

None.

Returns

Dreamweaver expects a string of the form `"floaterName1,floaterName2,...floaterNameN"`.

Example

The following example of `initialTabs()` specifies that the first time the floating panel appears, it should be tabbed together with the scriptEditor floating panel:

```
function initialTabs(){
  return "scriptEditor";
}
```

isATarget()

Availability

Dreamweaver MX (Windows only)

Description

Specifies whether other panels can dock to this panel. If `isATarget()` is not specified, the default is `false`, which prevents other panels from trying to dock to this one.

Arguments

None.

Returns

Dreamweaver expects a Boolean value that indicates whether other panels can dock to this panel.

Example

```
IsATarget()
{
  return true;
}
```

isAvailableInCodeView()

Description

Defined by a floating panel to determine whether the floating panel should be enabled when Code view is selected. If this function is not defined, the floating panel is disabled in the Code view.

Arguments

None.

Returns

Dreamweaver expects a Boolean value that indicates whether the floating panel should be enabled in Code view.

isResizable()

Availability

Dreamweaver 4

Description

Determines whether a user can resize a floating panel. If the function is not defined or returns a value of `true`, the user can resize the floating panel. If the function returns `false`, the user is unable to resize the floating panel.

Arguments

None.

Returns

`true` if the user can resize the floating panel, otherwise returns `false`.

Example

The following example prevents the user from resizing the floating panel.

```
function isResizable()
{
  return false;
}
```

selectionChanged()

Description

Called when the floating panel becomes visible and when the selection changes (when focus switches to a new document or when the insertion pointer moves to a new location in the current document). This function should be defined only if the floating panel must track the selection.

Note: Define `selectionChanged()` only if you absolutely require it because its existence impacts performance.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

The following example of `selectionChanged()` shows a different layer in the floating panel, depending on whether the selection is a script marker. If the selection is a script marker, Dreamweaver makes the script layer visible. Otherwise, Dreamweaver makes the blank layer visible:

```
function selectionChanged(){
  /* get the selected node */
  var theDOM = dw.getDocumentDOM();
  var theNode = dw.getSelectedNode();

  /* check to see if the node is a script marker */
  if (theNode.nodeType == Node.ELEMENT_NODE && ¬
  theNode.tagName == "SCRIPT"){
  document.layers['blanklayer'].visibility = 'hidden';
  document.layers['scriptlayer'].visibility = 'visible';
  }else{
    document.layers['scriptlayer'].visibility = 'hidden';
    document.layers['blanklayer'].visibility = 'visible';
  }
}
```

About performance

Declaring the `selectionChanged()` or `documentEdited()` function in your custom floating panels risks impacting Dreamweaver performance adversely. Consider that `documentEdited()` and `selectionChanged()` are called after every keystroke and mouse click when Dreamweaver is idle for more than one-tenth of a second. It's important use different scenarios to test your floating panel, using large documents (100K or more of HTML) whenever possible, to test performance impact.

To help avoid performance penalties, `setTimeout()` was implemented as a global method in Dreamweaver 3. As in the browsers, `setTimeout()` takes two arguments: the JavaScript to be called and the amount of time in milliseconds to wait before calling it.

The `setTimeout()` method lets you build pauses into your processing. These pauses let the user continue interacting with the application. You must build in these pauses explicitly because the screen freezes while scripts process, which prevents the user from performing further edits. The pauses also prevent you from updating the interface or the floating panel.

The following example is from a floating panel that displays information about every layer in the document. It uses `setTimeout()` to pause for half a second after processing each layer:

```
/* create a flag that specifies whether an edit is being processed, and set it
   to false. */
document.running = false;

/* this function called when document is edited */
function documentEdited(){
  /* create a list of all the layers to be processed */
  var dom = dw.getDocumentDOM();
  document.layers = dom.getElementsByTagName("layer");
  document.numLayers = document.layers.length;
  document.numProcessed = 0;

  /* set a timer to call processLayer(); if we didn't get
   * to finish processing the previous edit, then the timer
   * is already set. */
  if (document.running = false){
    setTimeout("processLayer()", 500);
  }

  /* set the processing flag to true */
  document.running = true;
}

/* process one layer */
function processLayer(){
  /* display information for the next unprocessed layer.
      displayLayer() is a function you would write to
      perform the "magic".  */
   displayLayer(document.layers[document.numProcessed]);

  /* if there's more work to do, set a timeout to process
   * the next layer.  If we're finished, set the document.running
   * flag to false. */
  document.numProcessed = document.numProcessed + 1;
  if (document.numProcessed < document.numLayers){
    setTimeout("processLayer()", 500);
  }else{
    document.running = false;
  }
}
```

A simple floating panel example

The following floating panel example contains a text field that shows the contents of the selected Script marker (the yellow icon that appears in the Document window to mark the location of a script). If no Script marker is selected, a layer that contains the text `(no script selected)` appears.

```
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine5.0//floater">
<html>
<head>
<title>Script Editor</title>
<script language="JavaScript">

function selectionChanged(){
  /* get the selected node */
  var theDOM = dw.getDocumentDOM();
  var theNode = theDOM.getSelectedNode();

  /* check to see if the node is a script marker */
  if (theNode.nodeType == Node.ELEMENT_NODE && ¬
  theNode.tagName == "SCRIPT"){
    document.layers['scriptlayer'].visibility = 'visible';
    document.layers['scriptlayer'].document.theForm.¬
    scriptCode.value = theNode.innerHTML;
    document.layers['blanklayer'].visibility = 'hidden';
  }else{
    document.layers['scriptlayer'].visibility = 'hidden';
    document.layers['blanklayer'].visibility = 'visible';
  }
}

/* update the document with any changes made by
   the user in the textarea */
function updateScript(){
  var theDOM = dw.getDocumentDOM();
  var theNode = dw.getSelectedNode();
  theNode.innerHTML = document.layers['scriptlayer'].document.¬
  theForm.scriptCode.value;
}

</script>
</head>

<body>
<div id="blanklayer" style="position:absolute; width:422px; ¬
height:181px; z-index:1; left: 8px; top: 11px; ¬
visibility: hidden">
<center>
<br>
<br>
<br>
<br>
<br>
(no script selected)
</center>
</div>

<div id="scriptlayer" style="position:absolute; width:422px; ¬
height:181px; z-index:1; left: 8px; top: 11px; ¬
visibility: visible">
<form name="theForm">
<textarea name="scriptCode" cols="80" rows="20" wrap="VIRTUAL" ¬
onBlur="updateScript()"></textarea>
```

```
</form>
</div>

</body>
</html>
```

Remember that it is not sufficient to save this code in a file called scriptEditor.htm in the Configuration/Floaters folder; you must also call `dw.setFloaterVisibility('scriptEditor',true)` or `dw.toggleFloater('scriptEditor')` to load the floating panel and make it visible. The most obvious place from which to do this is the Window menu in the menus.xml file. The `menuitem` tag to toggle the script editor panel might look like this:

```
<menuitem name="Script Editor" enabled="true" ¬
command="dw.toggleFloater('scriptEditor')"¬
checked="dw.getFloaterVisibility('scriptEditor')" />
```

CHAPTER 13
Behaviors

Behaviors let users make their HTML pages interactive. They offer web designers an easy way to assign actions to page elements by filling in an HTML form.

You should write behavior actions when you want to share functions with users or when you want to insert the same JavaScript function repeatedly but change the parameters each time.

Note: You cannot use behaviors to insert VBScript functions directly; however, you can add a VBScript function indirectly by editing the DOM in the `applyBehavior()` function.

The term *behavior* refers to the combination of an event (such as `onClick`, `onLoad`, or `onSubmit`) and an action (such as Check Plugin, Go to URL, Swap Image). The browser determines which HTML elements accept which events. Files that list events that each browser supports are stored in the Configuration/Behaviors/Events folder within the Dreamweaver application folder.

Actions are the part of a behavior that you can control; so when you write a behavior, you're really writing an Action file. Actions are HTML files. The `BODY` of an Action file generally contains an HTML form that accepts parameters for the action (for example, parameters that indicate which layers are to be shown or hidden). The `HEAD` of an Action file contains JavaScript functions that process form input from the `BODY` and control the functions, arguments, and event handlers that are inserted into a user's document.

Note: For information about server behaviors that provide web application functionality, see "Server Behaviors" on page 145.

How Behaviors work

When a user selects an HTML element in a Dreamweaver document and clicks the plus (+) button, the following events occur:

1. Dreamweaver calls the `canAcceptBehavior()` function in each Action file to see whether this action is appropriate for the document or the selected element.

   If the return value of this function is `false`, Dreamweaver dims the action in the Actions pop-up menu. (For example, the Control Shockwave action is dimmed when the user's document has no Shockwave movies.) If the return value is a list of events, Dreamweaver compares each event with the valid events for the currently selected HTML element and target browser until it finds a match. Dreamweaver populates the Events pop-up menu with the matched event from `canAcceptBehavior()` at the top of the list; if no match exists, the default event for the HTML element (marked in the Event file with an asterisk [*]) becomes the top item. The remaining events in the menu are assembled from the Event file.

2. The user selects an action from the Actions pop-up menu.

3 Dreamweaver calls the `windowDimensions()` function, if defined, to determine the size of the Parameters dialog box. If `windowDimensions()` is not defined, the size is determined automatically.

   A dialog box always appears, with OK and Cancel buttons appearing at the right edge, regardless of the contents of the Body element.

4 Dreamweaver displays a dialog box that contains the `BODY` elements of the Action file. If the Action file's `BODY` tag contains an `onLoad` handler, Dreamweaver executes it.

5 The user fills in the parameters for the action. Dreamweaver executes event handlers that are associated with the form fields as the user encounters them.

6 The user clicks OK.

7 Dreamweaver calls the `behaviorFunction()` and `applyBehavior()` functions in the selected Action file. These functions return strings that are inserted into the user's document.

8 If the user later double-clicks the action in the Actions column, Dreamweaver reopens the Parameters dialog box and executes the `onLoad` handler. Dreamweaver then calls the `inspectBehavior()` function in the selected Action file, which fills in the fields with the data that the user previously entered.

Inserting multiple functions in the user's file

Actions can insert multiple functions—the main behavior function plus any number of helper functions—into the `HEAD`. Two or more behaviors can even share helper functions, as long as the function definition is exactly the same in each Action file. One way of ensuring that shared functions are identical is to store each helper function in an external JavaScript file and insert it into the appropriate Action files using `<SCRIPT SRC="externalFile.js">`.

When the user deletes a behavior, Dreamweaver attempts to remove any unused helper functions that are associated with the behavior. If other behaviors are using a helper function, it is not deleted. Because the algorithm for deleting helper functions errs on the side of caution, Dreamweaver might occasionally leave an unused function in the user's document.

The Behaviors API

Two Behaviors API functions are required (`applyBehavior()` and `behaviorFunction()`); the rest are optional.

applyBehavior()

Description

Inserts into the user's document an event handler that calls the function that `behaviorFunction()` inserts. This function can also perform other edits on the user's document, but it must not delete the object to which the behavior is being applied or the object that receives the action.

Arguments

`uniqueName`

The argument is a unique identifier among all instances of all behaviors in the user's document. Its format is *`functionNameInteger`*, where *`functionName`* is the name of the function that `behaviorFunction()` inserts. This argument is useful if you insert a tag into the user's document and you want to assign a unique value to its `NAME` attribute.

Returns

Dreamweaver expects a string that contains the function call to be inserted in the user's document, usually after accepting parameters from the user. If `applyBehavior()` determines that the user made an invalid entry, the function can return an error string instead of the function call. If the string is empty (`return "";`), Dreamweaver does not report an error; if the string is not empty and not a function call, Dreamweaver displays a dialog box with the text: `Invalid input supplied for this behavior: [the string returned from applyBehavior()]`. If the return value is `null` (`return;`), Dreamweaver indicates that an error occurred but gives no specific information.

Note: Quotation marks within the returned string must be preceded by a backslash (\) to avoid errors that the JavaScript interpreter reports.

Example

The following instance of `applyBehavior()` returns a call to the function `MM_openBrWindow()` and passes parameters that are given by the user (the height and width of the window; whether the window should have scroll bars, a toolbar, a location bar, and other features; and the URL that should open in the window):

```
function applyBehavior() {
  var i,theURL,theName,arrayIndex = 0;
  var argArray = new Array(); //use array to produce correct ¬
  number of commas w/o spaces
  var checkBoxNames = new Array("toolbar","location",¬
  "status","menubar","scrollbars","resizable");

  for (i=0; i<checkBoxNames.length; i++) {
    theCheckBox = eval("document.theForm." + checkBoxNames[i]);
    if (theCheckBox.checked) argArray[arrayIndex++] = ¬
    (checkBoxNames[i] + "=yes");
  }
  if (document.theForm.width.value)
    argArray[arrayIndex++] = ("width=" + ¬
    document.theForm.width.value);
  if (document.theForm.height.value)
    argArray[arrayIndex++] = ("height=" + ¬
    document.theForm.height.value);
  theURL = escape(document.theForm.URL.value);
  theName = document.theForm.winName.value;
  return "MM_openBrWindow('"+theURL+"',¬
  '"+theName+"','"+argArray.join()+"')";
}
```

behaviorFunction()

Description

Inserts one or more functions—surrounded by `<SCRIPT LANGUAGE="JavaScript"></SCRIPT>` tags, if none yet exist—into the `HEAD` of the user's document.

Arguments

None.

Returns

Dreamweaver expects either a string that contains the JavaScript functions or a string that contains the names of the functions to be inserted in the user's document. This value must be exactly the same every time (it cannot depend on input from the user). The functions are inserted only once, regardless of how many times the action is applied to elements in the document.

Note: Quotation marks within the returned string must be preceded by a backslash (\) to avoid errors that the JavaScript interpreter reports.

Example

The following instance of `behaviorFunction()` returns a function called `MM_popupMsg()`:

```
function behaviorFunction(){
  return ""+
  "function MM_popupMsg(theMsg) { //v1.0\n"+
  "  alert(theMsg);\n"+
  "}";
}
```

The following example is equivalent to the preceding `behaviorFunction()` declaration and is the method used to declare `behaviorFunction()` in all behaviors that come with Dreamweaver:

```
function MM_popupMsg(theMsg){ //v1.0
  alert(theMsg);
}

function behaviorFunction(){
  return "MM_popupMsg";
}
```

canAcceptBehavior()

Description

Determines whether the action is allowed for the selected HTML element and specifies the default event that should trigger the action. Can also check for the existence of certain objects (such as Shockwave movies) in the user's document and not allow the action if these objects do not appear.

Arguments

`HTMLElement`

The argument is the selected HTML element.

Returns

Dreamweaver expects one of the following values:

- `true` if the action is allowed but has no preferred events.
- A list of preferred events (in descending order of preference) for this action. Specifying preferred events overrides the default event (as denoted with an asterisk [*] in the Event file) for the selected object. See step 1 in "How Behaviors work" on page 135.
- `false` if the action is not allowed.

If `canAcceptBehavior()` returns `false`, the action is dimmed in the Actions pop-up menu in the Behaviors panel.

Example

The following instance of `canAcceptBehavior()` returns a list of preferred events for the behavior if the document has any named images:

```
function canAcceptBehavior(){
  var theDOM = dreamweaver.getDocumentDOM();
  // Get an array of all images in the document
  var allImages = theDOM.getElementsByTagName('IMG');
  if (allImages.length > 0){
    return "onMouseOver, onClick, onMouseDown";
  }else{
    return false;
  }
}
```

displayHelp()

Description

If this function is defined, a Help button appears below the OK and Cancel buttons in the Parameters dialog box. This function is called when the user clicks the Help button.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

```
// the following instance of displayHelp() opens
// in a browser a file that explains how to use
// the extension.
function displayHelp(){
  var myHelpFile = dw.getConfigurationPath() +
    '/ExtensionsHelp/superDuperHelp.htm';
  dw.browseDocument(myHelpFile);
}
```

deleteBehavior()

Description

Undoes any edits that `applyBehavior()` performed.

Note: Dreamweaver automatically deletes the function declaration and the event handler that are associated with a behavior when the user deletes the behavior in the Behaviors panel. It is necessary to define `deleteBehavior()` only if the `applyBehavior()` function performs additional edits on the user's document (for example, if it inserts a tag).

Arguments

`applyBehaviorString`

This argument is the string that the `applyBehavior()` function returns.

Returns

Dreamweaver expects nothing.

identifyBehaviorArguments()

Description

Identifies arguments from a behavior function call as navigation links, dependent files, URLs, Netscape Navigator 4.0-style references, or object names so that URLs in behaviors can update if the user saves the document to another location and so the referenced files can appear in the site map and be considered dependent files for the purposes of uploading to and downloading from a server.

Arguments

theFunctionCall

This argument is the string that the `applyBehavior()` function returns.

Returns

Dreamweaver expects a string that contains a comma-separated list of the types of arguments in the function call. The length of the list must equal the number of arguments in the function call. Argument types must be one of the following types:

- `nav` specifies that the argument is a navigational URL, and therefore, it should appear in the site map.
- `dep` specifies that the argument is a dependent file URL, and therefore, it should be included with all other dependent files when a document that contains this behavior is downloaded from or uploaded to a server.
- `URL` specifies that the argument is both a navigational URL and a dependent URL or that it is a URL of an unknown type, and therefore, that it should appear in the site map and be considered a dependent file when uploading to or downloading from a server.
- `NS4.0ref` specifies that the argument is a Netscape Navigator 4.0-style object reference.
- `IE4.0ref` specifies that the argument is an Internet Explorer DOM 4.0-style object reference.
- `objName` specifies that the argument is a simple object name, as specified in the `NAME` attribute for the object. This type was added in Dreamweaver 3.
- `other` specifies that the argument is none of the above types.

Example

This simple example of `identifyBehaviorArguments()` works for the Open Browser Window behavior action, which returns a function that always has three arguments (the URL to open, the name of the new window, and the list of window properties):

```
function identifyBehaviorArguments(fnCallStr) {
  return "URL,other,other";
}
```

A more complex version of `identifyBehaviorArguments()` is necessary for behavior functions that have a variable number of arguments (such as Show/Hide Layer). For this version of `identifyBehaviorArguments()`, there is a minimum number of arguments, and additional arguments always come in multiples of the minimum number. In other words, a function with a minimum number of arguments of 4 may have 4, 8, or 12 arguments, but it cannot have 10 arguments.

```
function identifyBehaviorArguments(fnCallStr) {
  var listOfArgTypes;
  var itemArray = dreamweaver.getTokens(fnCallStr, '(),');

  // The array of items returned by getTokens() includes the
  // function name, so the number of *arguments* in the array
  // is the length of the array minus one. Divide by 4 to get the
  // number of groups of arguments.
  var numArgGroups = ((itemArray.length - 1)/4);
  // For each group of arguments
  for (i=0; i < numArgGroups; i++){

  // Add a comma and "NS4.0ref,IE4.0ref,other,dep" (because this
  // hypothetical behavior function has a minimum of four
  // arguments the Netscape object reference, the IE object
  // reference, a dependent URL, and perhaps a property value
  // such as "show" or "hide") to the existing list of argument
  // types, or if no list yet exists, add only
  // "NS4.0ref,IE4.0ref,other,dep"
  var listOfArgTypes += ((listOfArgTypes)?",":"") + ¬
  "NS4.0ref,IE4.0ref,other,dep";
  }
}
```

inspectBehavior()

Description

Inspects the function call for a previously applied behavior in the user's document and sets the values of the options in the Parameters dialog box accordingly. If `inspectBehavior()` is not defined, the default option values appear.

Note: `inspectBehavior()` must rely solely on information that the *`applyBehaviorString`* argument passes to it. Do not attempt to obtain other information about the user's document (for example, using `dreamweaver.getDocumentDOM()`) within this function.

Arguments

`applyBehaviorString`

This argument is the string that the `applyBehavior()` function returns.

Returns

Dreamweaver expects nothing.

Example

The following instance of `inspectBehavior()`, taken from Display Status Message.htm, fills in the Message field in the parameters form with the message that the user selected when the behavior was originally applied:

```
function inspectBehavior(msgStr){
  var startStr = msgStr.indexOf("'") + 1;
  var endStr = msgStr.lastIndexOf("'");
  if (startStr > 0 && endStr > startStr) {
    document.theForm.message.value = ¬
    unescQuotes(msgStr.substring(startStr,endStr));
  }
}
```

Note: For more information about the `unescQuotes()` function, see the dwscripts.js file in the Configuration/Shared/Common/Scripts/CMN folder.

windowDimensions()

Description

Sets specific dimensions for the Parameters dialog box. If this function is not defined, the window dimensions are computed automatically.

Note: Do not define this function unless you want an Parameters dialog box that is larger than 640 x 480 pixels.

Arguments

`platform`

The value of the argument is either `"macintosh"` or `"windows"`, depending on the user's platform.

Returns

Dreamweaver expects a string of the form `"widthInPixels,heightInPixels"`.

The returned dimensions are smaller than the size of the entire dialog box because they do not include the area for the OK and Cancel buttons. If the returned dimensions do not accommodate all options, scroll bars appear.

Example

The following instance of `windowDimensions()` sets the dimensions of the Parameters dialog box to 648 x 520 pixels:

```
function windowDimensions(){
  return "648,520";
}
```

What to do when an action requires a return value

Sometimes an event handler must have a return value (for example, `onMouseOver="window.status='This is a link'; return true"`). But if Dreamweaver inserts `"return behaviorName(args)"` into the event handler, behaviors later in the list are skipped.

To get around this limitation, set a variable called `document.MM_returnValue` to the desired return value within the string that `behaviorFunction()` returns. This setting causes Dreamweaver to insert `return document.MM_returnValue` at the end of the list of actions in the event handler. See the Validate Form.js file in the Configuration/Behaviors/Actions folder within the Dreamweaver application folder for an example of the use of `MM_returnValue`.

A simple behavior example

To understand how behaviors work and how you can create one, it's helpful to look at an example. The Configuration/Behaviors/Actions folder inside the Dreamweaver application folder contains many examples; however, many are likely to be too complex a starting point for all but the most advanced developers. The simplest Action file to start with is Call JavaScript.htm (along with its counterpart, Call JavaScript.js, which contains all the JavaScript functions).

The following code presents a relatively simple example. It checks the brand of the browser and goes to one page if the brand is Netscape Navigator and another if the brand is Microsoft Internet Explorer. This code can easily be expanded to check for other brands such as Opera and WebTV and modified to perform other actions than going to URLs.

```
<!DOCTYPE HTML SYSTEM "-//Macromedia//DWExtension layout-engine 5.0//dialog">
<html>
<head>
<title>behavior "Check Browser Brand"</title>
<meta http-equiv="Content-Type" content="text/html">
<script language="JavaScript">

// The function that will be inserted into the
// HEAD of the user's document
function checkBrowserBrand(netscapeURL,explorerURL) {
  if (navigator.appName == "Netscape") {
    if (netscapeURL) location.href = netscapeURL;
  }else if (navigator.appName == "Microsoft Internet Explorer") {
    if (explorerURL) location.href = explorerURL;
  }
}

//****************** API **********************

function canAcceptBehavior(){
  return true;
}

// Return the name of the function to be inserted into
// the HEAD of the user's document
function behaviorFunction(){
    return "checkBrowserBrand";
}

// Create the function call that will be inserted
// with the event handler
function applyBehavior() {
  var nsURL = escape(document.theForm.nsURL.value);
  var ieURL = escape(document.theForm.ieURL.value);
  if (nsURL && ieURL) {
    return "checkBrowserBrand(\'" + nsURL + "\',\'" + ieURL + ¬
    "\')";
  }else{
    return "Please enter URLs in both fields."
  }
}

// Extract the arguments from the function call
// in the event handler and repopulate the
// parameters form
function inspectBehavior(fnCall){
  var argArray = getTokens(fnCall, "()',");
  var nsURL = unescape(argArray[1]);
```

```
  var ieURL = unescape(argArray[2]);
  document.theForm.nsURL.value = nsURL;
  document.theForm.ieURL.value = ieURL;
}

//***************** LOCAL FUNCTIONS  ******************

// Put the cursor in the first text field
// and select the contents, if any
function initializeUI(){
  document.theForm.nsURL.focus();
  document.theForm.nsURL.select();
}

// Let the user browse to the Navigator and
// IE URLs
function browseForURLs(whichButton){
  var theURL = dreamweaver.browseForFileURL();
  if (whichButton == "nsURL"){
    document.theForm.nsURL.value = theURL;
  }else{
    document.theForm.ieURL.value = theURL;
  }
}

//*************** END OF JAVASCRIPT *****************
</script>
</head>
<body>
<form method="post" action="" name="theForm">
<table border="0" cellpadding="8">
<tr>
<td nowrap="nowrap">  Go to this URL if the browser is ¬
Netscape Navigator:<br>
<input type="text" name="nsURL" size="50" value="">  
<input type="button" name="nsBrowse" value="Browse..." ¬
onClick="browseForURLs('nsURL')"></td>
</tr>
<tr>
<td nowrap="nowrap">  Go to this URL is the browser is ¬
Microsoft Internet Explorer:<br>
<input type="text" name="ieURL" size="50" value="">  
<input type="button" name="ieBrowse" value="Browse..." ¬
onClick="browseForURLs('ieURL')"></td>
</tr>
</table>
</form>
</body>
</html>
```

CHAPTER 14
Server Behaviors

Dreamweaver MX provides users with an interface for adding server behaviors into their documents to perform server-side tasks such as filtering records based on user criteria, paging through records, linking result lists to details pages, and inserting records into a result set. If a Dreamweaver user repeatedly inserts the same runtime code into documents, you can create a new extension to automate updating a document with these frequently used code blocks. See "Adding Custom Server Behaviors" in *Getting Started with Dreamweaver MX* for details about working with the Server Behavior Builder interface to implement a custom server behavior. Then, refer to this chapter for details about working with the supporting server behavior files and the functions that are available for interacting with established server behaviors. Dreamweaver currently supports server behavior extensions that add runtime code for the following server models: ASP.Net/C#, ASP.Net/VisualBasic, ASP/JavaScript, ASP/VBScript, ColdFusion, JSP, and PHP/MySQL.

The following terms are used throughout this chapter:

- **Server Behavior extension:** The server behavior extension is the interface between server-side code and Dreamweaver. A server behavior extension consists of JavaScript, HTML, and Extension Data Markup Language (EDML), which is XML that is created specifically for extension data. Examples of these files reside in your installation directory in the Configuration/ServerBehaviors folder, arranged according to server model. When you script an extension, use `dwscripts.applySB()` to instruct Dreamweaver to read the EDML files, retrieve the components of your extension, and add the appropriate code blocks to the user's document.
- **Server behavior instance:** When Dreamweaver adds code blocks to a user's document, the inserted code constitutes an instance of the server behavior. The user can apply most server behaviors more than once, which results in multiple server behavior instances. Each server behavior instance is listed in the Server Behaviors panel of the Dreamweaver interface.
- **Runtime code:** Runtime code is the set of code blocks that are added to a document when a server behavior is applied. These code blocks usually include some server-side code, such as ASP script that is enclosed in `<% ... %>` tags.
- **Participants:** Your server behavior extension inserts code blocks into the user's document. A code block is a single, continuous block of script, such as a server-side tag, an HTML tag, or an attribute that adds server-side functionality to a web page. An EDML file defines each code block as a participant. All the participants (code blocks) for a given server behavior comprise one participant group.

Note: For information about participants, participant groups, and how Dreamweaver EDML files are structured, see "Extension Data Markup Language" on page 146.

Dreamweaver architecture

When you use the Server Behavior Builder to create a Dreamweaver-specific extension, Dreamweaver creates several files (EDML and HTML script files) that support inserting the Server Behavior code into a Dreamweaver document (some behaviors also reference JavaScript files for additional functionality). The architecture simplifies your implementation of the API and also separates your runtime code from how Dreamweaver deploys it. This chapter discusses ways of modifying these files.

Server behavior folders and files

The user interface for each server behavior resides in the Configuration/ServerBehaviors/ServerModelName folder, where `ServerModelName` is one of the following server types: ASP.NET_Csharp, ASP.NET_VB (Visual Basic), ASP_Js (JavaScript), ASP_Vbs (VBScript), ColdFusion (Dreamweaver MX compatible), JSP, PHP_MySQL, Shared (UltraDev 4 ColdFusion and Dreamweaver MX ColdFusion) or UD4-ColdFusion (Ultradev 4-compatible ColdFusion).

Note: A distinction between Dreamweaver MX and Ultradev 4 ColdFusion compatibility is required because the document type/server model for ColdFusion has changed since the release of Ultradev 4. For example, a server behavior in Dreamweaver MX inserts CFML code that is different from the CFMLthat is inserted by the same server behavior from Ultradev 4.

Extension Data Markup Language

Dreamweaver generates two EDML files when you use the Server Behavior Builder: a group EDML file and a participant EDML file that correspond to the names that you provide in the Server Behavior Builder. The group file defines the relevant participants, which represent code blocks, and the groups define which participants are combined to make an individual server behavior.

Group files

Group files contain a list of participants, and participant files have all server-model-specific code data. Participant files can be used by more than one extension, so several group files can refer to the same participant file.

The following example shows a high-level view of the Server Behavior Group EDML file. For a complete list of elements and attributes, see “Group EDML file tags” on page 160.

```
<group serverBehavior="Go To Detail Page.htm" dataSource="Recordset.htm">
  <groupParticipants selectParticipant="goToDetailPage_attr">
    <groupParticipant name="moveTo_declareParam"  partType="member"/>
    <groupParticipant name="moveTo_keepParams"  partType="member"/>
    <groupParticipant name="goToDetailPage_attr" partType="identifier" />
  </groupParticipants>
</group>
```

In the `groupParticipants` block tag, each `groupParticipant` tag indicates the EDML participant file that contains the code block to use. The value of the `name` attribute is the participant file name minus the .edml extension (for example, `moveTo_declareParam`).

Participant files

A participant represents a single code block on the page, such as a server tag, an HTML tag, or an attribute. A participant file must be listed in a group file to be available to a Dreamweaver document author. A single participant file can be used by several group files.

For example, the moveTo_declareParam.edml file contains the following code:

```
<participant>
     <quickSearch><![CDATA[MM_paramName]]></quickSearch>
     <insertText location="aboveHTML+80">
<![CDATA[
<% var MM_paramName = ""; %>
]]>
     </insertText>
     <searchPatterns whereToSearch="directive">
       <searchPattern><![CDATA[/var\s*MM_paramName/]]></searchPattern>
     </searchPatterns>
</participant>
```

When Dreamweaver adds a server behavior to a document, it needs to have detailed information, including where to insert the code, what the code looks like, and what parameters the Dreamweaver author or data replaced at runtime. Each participant EDML file describes these details for each block of code. Specifically, the participant file describes the following data:

- The code and where to put the unique instance. These are defined by the `insertText` tag parameters, as shown in the following example:

  ```
  <insertText location="aboveHTML+80">
  ```

- How to recognize instances already on the page, as shown in the following example:

  ```
  <searchPatterns whereToSearch="directive">
       <searchPattern><![CDATA[/var\s*MM_paramName/]]></searchPattern>
  </searchPatterns>
  ```

In the `searchPatterns` block tag, each `searchPattern` contains a pattern that finds instances of runtime code and extracts specific parameters. For more details, see "Server behavior techniques" on page 183.

The script file

Each server behavior also has an HTML file that contains functions and links to the scripts that manage the integration of the server behavior code with the Dreamweaver interface. The functions that are available for editing in this file are discussed in "Server behavior implementation functions" on page 156.

Hello World example

This example takes you through the creation of a new server behavior so you can see the files that Dreamweaver creates and how to handle them. Again, see "Adding Custom Server Behaviors" in the *Getting Started with Dreamweaver MX* manual for details about working with the Server Behavior Builder interface. The example displays "Hello World" from the ASP server. The Hello World behavior has only one participant (a single ASP tag) and does not modify or add anything else on the page.

Note: This example refers to functions that are defined later in this chapter.

Create a new dynamic page document.

1. In Dreamweaver, select the File > New menu option.
2. In the New Document dialog box, select:

   Category: Dynamic Page

   Dynamic Page: ASP JavaScript

3 Click Create.

Use the Server Behavior Builder to define your new server behavior.

Note: If the Server Behaviors panel is not open and visible, select the Window > Server Behaviors menu option.

1 In the Server Behaviors panel, select the plus (+) button and select the New Server Behavior menu option.

2 In the New Server Behavior dialog box, select:

Document Type: ASP JavaScript

Name: Hello World

(Leave the "Copy existing server behavior" checkbox unchecked.)

3 Click OK.

Define the code to insert.

1 Select the plus (+) button for Code Blocks to Insert.

2 In the Create a New Code Block dialog box, enter `Hello_World_block1` (Dreamweaver might automatically enter this information for you).

3 Click OK.

4 In the Code Block field, enter `<% Response.Write("Hello World") %>`.

5 In the Insert Code pop-up menu, select Relative to the Selection so the user can control where this code goes in the document.

6 In the Relative Position pop-up menu, select After the Selection.

7 Click OK.

In the Server Behaviors panel, you can see that the plus (+) menu contains the new server behavior in the pop-up list. Also, in the installation directory for your Dreamweaver MX files, the Configuration/ServerBehaviors/ASP_Js directory now contains three files:

Note: If you are working in a multiuser configuration, these files will appear in your Application Data folder.

- The group file: `Hello World.edml`
- The participant file: `Hello World_block1.edml`
- A script file: `Hello World.htm`

How the Server Behavior API functions are called

The Server Behavior API functions are called in the following scenarios:

- The `findServerBehaviors()` function is called when the document opens and again when the participant is edited. It searches the user's document for instances of the server behavior. For each instance it finds, `findServerBehaviors()` creates a JavaScript object, and uses JavaScript properties to attach state information to the object.
- If it is implemented, Dreamweaver calls the `analyzeServerBehavior()` function for each behavior instance that is found in the user's document after all the `findServerBehaviors()` functions are called.

  When the `findServerBehaviors()` function creates a behavior object, it usually sets the four properties (`incomplete`, `participants`, `selectedNode`, and `title`). However, it is sometimes easier to delay setting some of the properties until all the other server behaviors find instances of themselves. For example, the Move To Next Record behavior has two participants, a link object and a recordset object. Rather than finding the recordset object in its `findServerBehaviors()` function, wait until the recordset behavior's `findServerBehaviors()` function runs because the recordset finds all instances of itself.

  When the Move To Next Record behavior's `analyzeServerBehavior()` function is called, it gets an array that contains all the server behavior objects in the document. The function can look through the array for its recordset object.

  Sometimes during analysis, a single tag in the user's document is identified by two or more behaviors as being an instance of that behavior. For example, the `findServerBehaviors()` function for the Dynamic Attribute behavior might detect an instance of the Dynamic Attribute behavior that is associated with an `<input>` tag in the user's document. At the same time, the `findServerBehaviors()` function for the Dynamic Textfield behavior might look at the same `<input>` tag and detect an instance of the Dynamic Textfield behavior.

  As a result, the Server Behaviors panel shows the Dynamic Attribute block and the Dynamic Textfield. To correct this problem, the `analyzeServerBehavior()` functions need to delete all but one of these redundant server behaviors.

  To delete a server behavior, an `analyzeServerBehavior()` function can set the `"deleted"` property of any server behavior to be `true`. If the `deleted` property is still `true` when Dreamweaver finishes calling the `analyzeServerBehavior()` functions, the behavior is deleted from the list.

- When the user clicks the plus (+) button in the Server Behaviors panel, the pop-up menu appears.

  To determine the content of the menu, Dreamweaver first looks for a ServerBehaviors.xml file in the same folder as the behaviors. ServerBehaviors.xml references the HTML files that should appear in the menu.

  If the referenced HTML file contains a title tag, the contents of the title tag appear in the menu. For example, if the ServerBehaviors/ASP_Js/ GetRecords.htm file contains the tag `<title>Get More Records</title>`, `Get More Records` appears in the menu.

  If the file does not contain a title tag, the filename appears in the menu. For example, if GetRecords.htm does not contain a title tag, `GetRecords` appears in the menu.

  If there is no ServerBehaviors.xml file or the folder contains one or more HTML files that are not mentioned in ServerBehaviors.xml, Dreamweaver checks each file for a title tag and uses the title tag or filename to populate the menu.

  If you do not want a file in the ServerBehaviors folder to appear in the menu, put the following statement on the first line in the HTML file:

  ```
  <!-- MENU-LOCATION=NONE -->
  ```

- When the user chooses an item from the menu, the `canApplyServerBehavior()` function is called. If that function returns `true`, a dialog box appears. When the user clicks OK, the `applyServerBehavior()` function is called.
- If the user edits an existing server behavior by double-clicking it, Dreamweaver displays the dialog box, executes the `onLoad` handler on the `BODY` tag, if one exists, and then calls `inspectServerBehavior()`. The `inspectServerBehavior()` function populates the form elements with the current parameter values. When the user clicks OK, Dreamweaver calls `applyServerBehavior()` again.
- If the user clicks the minus (-) button, the `deleteServerBehavior()` function is called. The `deleteServerBehavior()` function removes the behavior from the document.
- When the user selects a server behavior and uses the Cut or Copy commands, Dreamweaver passes the object that represents the server behavior to its `copyServerBehavior()` function. The `copyServerBehavior()` function adds any additional properties to the server behavior object that are needed to paste it later.

  After the `copyServerBehavior()` function returns, Dreamweaver converts the server behavior object to a form that can be put on the Clipboard. When Dreamweaver converts the object, it deletes all the properties that reference objects; every property on the object that is not a number, Boolean value, or string is lost.

  When the user uses the Paste command, Dreamweaver unpacks the contents of the Clipboard and generates a new server behavior object. The new object is identical to the original, except that it does not have properties that reference objects. Dreamweaver passes the new server behavior object to `pasteServerBehavior()`. The `pasteServerBehavior()` function adds the behavior to the user's document. After `pasteServerBehavior()` returns, Dreamweaver calls the `findServerBehaviors()` function to get a new list of all the server behaviors in the user's document.

Users can copy and paste behaviors from one document to another. The `copyServerBehavior()` and `pasteServerBehavior()` functions should rely only on properties on the behavior object to exchange information.

The Server Behavior API

You can manage server behaviors with the following API functions.

analyzeServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Lets server behaviors set their `incomplete` and `deleted` flags.

After the `findServerBehaviors()` function is called for every server behavior on the page, an array of all the behaviors in the user's document appears. The `analyzeServerBehavior()` function is called for each JavaScript object in this array. For example, for a Dynamic Text behavior, Dreamweaver calls the `analyzeServerBehavior()` function in DynamicText.htm (or DynamicText.js).

One purpose of the `analyzeServerBehavior()` function is to finish setting all the properties (`incomplete`, `participants`, `selectedNode`, and `title`) on the behavior object. Sometimes it's easier to perform this task after `findServerBehaviors()` generates the complete list of server behaviors in the user's document.

The other purpose of the `analyzeServerBehavior()` function is to notice when two or more behaviors refer to the same tag in the user's document. In this case, the `deleted` property removes all but one behavior from the array.

Suppose the following three server behaviors are on a page: Recordset1, DynamicText1, and DynamicText2. Both DynamicText server behaviors need Recordset1 to exist on the page. After the server behaviors are found with `findServerBehaviors()`, Dreamweaver calls `analyzeServerBehavior()` for the three server behaviors. When `analyzeServerBehavior()` is called for DynamicText1, the function searches the array of all the server behavior objects on the page, looking for the one that belongs to Recordset1. If a server behavior object that belongs to Recordset1 cannot be found, the `incomplete` property is set to `true` so that an exclamation point appears in the Server Behaviors panel, which alerts the user that a problem exists. Similarly, when `analyzeServerBehavior()` is called for DynamicText2, the function searches for the object that belongs to Recordset1. Because Recordset1 does not depend on other server behaviors, it does not need to define the `analyzeServerBehavior()` function in this example.

Arguments

`serverBehavior`, *`[serverBehaviorArray]`*

- *`serverBehavior`* is a JavaScript object that represents the behavior to analyze.
- *`[serverBehaviorArray]`* is an array of JavaScript objects that represents all the server behaviors that are found on a page.

Returns

Nothing.

applyServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Reads values from the form elements in the dialog box and adds the behavior to the user's document. Dreamweaver calls this function when the user clicks OK in the Server Behaviors dialog box. If this function returns successfully, the Server Behaviors dialog box closes. If this function fails, it displays the error message without closing the Server Behaviors dialog box. This function can edit a user's document.

For more information, see "dwscripts.applySB()" on page 157.

Arguments

serverBehavior

serverBehavior is a JavaScript object that represents the server behavior; it is necessary to modify an existing behavior. If this is a new behavior, the argument is `null`.

Returns

An empty string if successful. An error message returns if this function fails.

canApplyServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Determines whether a behavior can be applied. Dreamweaver calls this function before displaying the Server Behaviors dialog box. If this function returns `true`, the Server Behaviors dialog box appears. If this function returns `false`, the Server Behaviors dialog box does not appear and the attempt to add a server behavior stops.

Arguments

serverBehavior

serverBehavior is a JavaScript object that represents the behavior; it is necessary to modify an existing behavior. If this is a new behavior, the argument is `null`.

Returns

`true` if the behavior can be applied; `false` otherwise.

copyServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Implementing `copyServerBehavior()` is optional. Users can copy instances of the specified server behavior. In the following example, this function is implemented for recordsets. If a user selects a recordset in the Server Behaviors panel or the Data Binding panel, using the Copy command copies the behavior to the Clipboard; using the Cut command cuts the behavior to the Clipboard. For server behaviors that do not implement this function, the Copy and Cut commands do nothing. For more information, see "How the Server Behavior API functions are called" on page 149.

The `copyServerBehavior()` function should rely only on behavior object properties that can be converted into strings to exchange information with the `pasteServerBehavior()` function. The Clipboard stores only raw text, so `participant` nodes in the document should be resolved and the resulting raw text should be saved into a secondary property.

Note: The `pasteServerBehavior()` function must also be implemented to enable the user to paste the behavior into any Dreamweaver document.

Arguments

`serverBehavior`

`serverBehavior` is a JavaScript object that represents the behavior.

Returns

`true` if the behavior copies successfully to the Clipboard; `false` otherwise.

deleteServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Removes the behavior from the user's document. This function is called when the user clicks the minus (-) button in the Server Behaviors panel. It can edit a user's document.

For more information, see "dwscripts.deleteSB()" on page 157.

Arguments

`serverBehavior`

`serverBehavior` is a JavaScript object that represents the behavior.

Returns

Nothing.

displayHelp()

Description

If this function is defined, a Help button appears below the OK and Cancel buttons in the dialog box. This function is called when the user clicks the Help button.

Arguments

None.

Returns

Nothing.

Example

```
// the following instance of displayHelp() opens
// in a browser a file that explains how to use
// the extension.
function displayHelp(){
  var myHelpFile = dw.getConfigurationPath() +
    '/ExtensionsHelp/superDuperHelp.htm';
  dw.browseDocument(myHelpFile);
}
```

findServerBehaviors()

Availability

Dreamweaver UltraDev 1

Description

Searches the user's document for instances of itself. For each instance it finds, `findServerBehaviors()` creates a JavaScript object, and it attaches state information as JavaScript properties of the object.

The four required properties are `incomplete`, `participants`, `title`, and `selectedNode`. You can set additional properties as necessary.

For more information, see "dwscripts.findSBs()" on page 156 and "dreamweaver.getParticipants()" on page 155.

Arguments

None.

Returns

An array of JavaScript objects; the length of the array is equal to the number of behavior instances that are found in the page.

inspectServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Determines the settings for the Server Behavior dialog box, based on the specified behavior object. Dreamweaver calls this function when a user displays a Server Behavior dialog box. Dreamweaver calls this function only when a user edits an existing behavior.

Arguments

serverBehavior

serverBehavior is a JavaScript object that represents the behavior. It is the same object that findServerBehaviors() returns.

Returns

Nothing.

pasteServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

If it is implemented, users can paste instances of the specified server behavior using pasteServerBehavior(). When the user pastes the server behavior, Dreamweaver organizes the contents of the Clipboard and generates a new behavior object. The new object is identical to the original, except that it lacks pointer properties. Dreamweaver passes the new behavior object to pasteServerBehavior(). The pasteServerBehavior() function relies on the properties of the behavior object to determine what to add to the user's document. The pasteServerBehavior() function then adds the behavior to the user's document. After pasteServerBehavior() returns, Dreamweaver calls the findServerBehaviors() functions to get a new list of all the server behaviors in the user's document.

Implementing pasteServerBehavior() is optional. For more information, see "How the Server Behavior API functions are called" on page 149.

Note: If you implement this function, you must also implement the copyServerBehavior() function.

Arguments

behavior is a JavaScript object that represents the behavior.

Returns

true if the behavior pastes successfully from the Clipboard; false otherwise.

dreamweaver.getParticipants()

Availability

Dreamweaver UltraDev 4

Description

The JavaScript function, dw.getParticipants(), gets a list of participants from the user's document. After Dreamweaver finds all the behavior's participants, it stores those lists. Typically, you use this function with the findServerBehaviors() function to locate instances of a behavior in the user's document.

Arguments

edmlFilename

edmlFilename is the name of the group or participant file that contains the names of the participants to locate in the user's document. This string is the filename, without the .edml extension.

Returns

The function returns an array that contains all instances of the specified participant (or, in the case of a group file, any instance of any participant in the group) that appear in the user's document. The array contains JavaScript objects, with one element in the array for each instance of each participant that is found in the user's document. The array is sorted in the order that the participants appear in the document. Each JavaScript object has the following properties:

- *participantNode* is a pointer to the participant node in the user's document.
- *participantName* is the name of the participant's EDML file (without the .edml extension).
- *parameters* is a JavaScript object that stores all the parameter/value pairs.
- *matchRangeMin* defines the character offset from the participant node of the document to the beginning of the participant content.
- *matchRangeMax* is an integer of the participant that defines the offset from the beginning of the participant node to the last character of the participant content.

Server behavior implementation functions

These functions can be added or edited within the HTML script files or the specified JavaScript files that are listed within the HTML script file.

dwscripts.findSBs()

Availability

Dreamweaver MX (this function replaces `findSBs()` from earlier versions of Dreamweaver)

Description

Finds all instances of a server behavior and all the participants on the current page. Sets the title, type, participants array, weights array, types array, `selectedNode`, and incomplete flag. This method also creates a parameter object that holds an array of user-definable properties such as recordset, name, and column name. You can return this array from the `findServerBehaviors()` function.

Arguments

serverBehaviorTitle is an optional title string that is used if no title is specified in the EDML title (useful for localization).

Returns

An array of JavaScript objects where the required properties are defined. Returns an empty array if no instances of the server behavior appear on the page.

Example

The following code searches for all instances of a particular server behavior in the current user document:

```
function findServerBehaviors() {
  allMySBs = dwscripts.findSBs();
  return allMySBs;
}
```

dwscripts.applySB()

Availability

Dreamweaver MX (this function replaces `applySB()` from earlier versions of Dreamweaver)

Description

Inserts or updates runtime code for the server behavior. If the *`sbObj`* parameter is `null`, it inserts new runtime code; otherwise, it updates existing runtime code that is indicated by the *`sbObj`* object. User settings should be set as properties on a JavaScript object and passed in as *`paramObj`*. These settings should match all the parameters that are declared as `@@paramName@@` in the EDML insertion text.

Arguments

`paramObj, sbObj`

- *`paramObj`* is the object that contains the user parameters.
- *`sbObj`* is the prior server behavior object if you are updating an existing server behavior; `null` otherwise.

Returns

`true` if the server behavior is added successfully to the user's document; `false` otherwise.

Example

In the following example, you fill the `paramObj` with the user's input and call `dwscripts.applySB`, passing in the input and your server behavior, `sbObj`.

```
function applyServerBehaviors(sbObj) {

  // get all UI values here...
  paramObj = new Object();
  paramObj.rs      = rsName.value;
  paramObj.col     = colName.value;
  paramObj.url     = urlPath.value;
  paramObj.form__tag = formObj;

  dwscripts.applySB(paramObj, sbObj);
}
```

dwscripts.deleteSB()

Availability

Dreamweaver MX (this function replaces `deleteSB()` from earlier versions of Dreamweaver)

Description

Deletes all the participants of the `sbObj` server behavior instance. The entire participant is deleted, unless the EDML file indicates special delete instructions with the `<delete>` tag. It does not delete participants that belong to more than one server behavior instance (reference count > 1).

Arguments

`sbObj` is the server behavior object instance that you want to remove from the user's document.

Returns

Nothing.

Example

The following example deletes all the participants of the `sbObj` server behavior, except the participants that are protected by the EDML file's `<delete>` tag.

```
function deleteServerBehavior(sbObj) {
  dwscripts.deleteSB(sbObj);
}
```

Editing EDML files

You must maintain Dreamweaver coding conventions when you edit a file. Pay attention to the dependency of one element upon another. For example, if you update the tags that are being inserted, you might also need to update the search patterns.

Note: EDML files are new in Dreamweaver MX. If you are working with legacy server behaviors, see the earlier versions of the Extending Dreamweaver manuals.

Regular expressions

You must understand regular expressions as they are implemented in JavaScript 1.5. Also, you must know when it is appropriate to use them in the server behavior EDML files. For example, regular expressions cannot be used in `quickSearch` values, but they are used in `searchPattern` to find and extract data.

Regular expressions describe text strings by using characters that are assigned with special meanings (metacharacters) to represent the text, break it up, and process it according to predefined rules. Regular expressions are powerful parsing and processing tools because they provide a generalized way to represent a pattern.

Good reference books on JavaScript 1.5 have a regular expression section or chapter. This section examines how Dreamweaver server behavior EDML files use regular expressions in order to find parameters in your runtime code and extract their values. Each time a user edits a server behavior, prior parameter values need to be extracted from the instances of the runtime code. This extraction process is done by using regular expressions.

You should understand a few metacharacters and metasequences (special character groupings) that are useful in server behavior EDML files, as described in the following table.

Regular Expression	Description
\	Escapes special characters. For example: \. reverts the metacharacter back to a literal period; \/ reverts the forward slash to its literal meaning; and, \) reverts the parenthesis to its literal meaning.
/ ... /i	Ignore case when searching for the metasequence
(...)	Creates a parenthetical subexpression within the metasequence
\s*	Searches for white spaces

The EDML tag `<searchPatterns whereToSearch="directive">` declares that runtime code needs to be searched. Each `<searchPattern>...</searchPattern>` subtag defines one pattern in the runtime code that must be identified. For the Redirect If Empty example, there are two patterns.

To extract parameter values from `<% if (@@rs@@.EOF) Response.Redirect("@@new__url@@"); %>`, write a regular expression that identifies any string `rs` and `new__url`:

```
<searchPattern paramNames="rs,new__url">
  /if d ((\w+)\.EOF\) Response\.Redirect\("([^\r\n]*)"\)/i
</searchPattern>
```

This process searches the user's document, and if there is a match, extracts the parameter values. The value for `rs` is extracted using the first parenthetical subexpression (`\w+`). The value for `new__url` is extracted using the second subexpression (`[^\r\n]*`).

Note: The character sequence "`[^\r\n]*`" matches any character that is not a linefeed, for both Macintosh and Windows.

Notes about EDML structure

You should use a unique filename to identify your server behavior group. If an associated participant file is used by only one group file, match the participant filename with the group name. Using this convention, the server behavior group file `updateRecord.edml` works with the participant file `updateRecord_init.edml`. When participant files might be shared between server behavior groups, assign unique descriptive names.

Note: The EDML name space is shared, regardless of folder structure, so be careful to keep names unique when you make them descriptive. Filenames should not exceed 31 characters (including the .edml extension), due to Macintosh limitations.

The runtime code for your server behavior resides inside the EDML files. The EDML parser should not confuse any of your runtime code with EDML markup, so `CDATA` must wrap around your runtime code. `CDATA` represents character data and is any text that is not EDML markup. When you use the `CDATA` tag, the EDML parser won't try to interpret it as markup, but instead, considers it as a block of plain text. `CDATA` marked blocks begin with `<![CDATA[` and end with `]]>`.

If you insert the text `Hello, World`, it is simple to specify your EDML, as shown in the following example:

```
<insertText>Hello, World</insertText>
```

However, if you insert content that has tags in it, such as `<img src='foo.gif'>`, it can confuse the EDML parser. In that case, embed it in the `CDATA` construct, as shown in the following example:

```
<insertText><![CDATA[<img src='foo.gif'>]]></insertText>
```

The ASP runtime code is wrapped within the `CDATA` tag set, as shown in the following example:

```
<![CDATA[
  <% if (@@rs@@.EOF) Response.Redirect("@@new__url@@"); %>
]]
```

Because of the `CDATA` tag, the ASP tags `<%= %>`, along with the other content within the tag, aren't processed. Instead, the Extension Data Manager (EDM) receives the uninterpreted text, as shown in the following example:

```
<% if (Recordset1.EOF) Response.Redirect("http://www.macromedia.com"); %>
```

In the following EDML definitions, the locations where `CDATA` is recommended are indicated in the examples.

Group EDML file tags

These tags and attributes are valid within the EDML group files.

EDML Tag: group

Description

Contains all specifications for a group of participants.

Parent

None.

Type

Block tag.

Required

Yes.

Attribute: version

Description

Defines the version of Dreamweaver that is current with the group file. For Dreamweaver MX, the version number is 6. If no version is specified, Dreamweaver assumes 4, or the prior release. All groups and participants that the Server Behavior Builder creates have the version attribute set to 6. The group version of this attribute currently has no effect.

Parent

`group`

Type

Attribute.

Required

No.

Attribute: serverBehavior

Description

The `serverBehavior` attribute indicates which server behavior can use the group. When any of the group's participant `quickSearch` strings are found in the document, the server behavior that is indicated by the `serverBehavior` attribute has Dreamweaver call `findServerBehaviors()`.

In some cases, if multiple groups are associated with a single server behavior, the server behavior must resolve which particular group to use.

Parent

`group`

Type

Attribute.

Required

No.

Value

The exact name (without a path) of any server behavior HTML file within a Configuration/ServerBehaviors folder, as shown in the following example:

```
<group serverBehavior="redirectIfEmpty.htm">
```

Attribute: dataSource

Description

This advanced feature supports new data sources that can be added to Dreamweaver.

Multiple versions of a server behavior can differ, depending on which data source you use. For example, the Repeat Region Server Behavior is designed for the standard Recordset.htm data source. If Dreamweaver is extended to support a new type of data source (such as a COM object), you can set `dataSource="COM.htm"` in a Group file with a different implementation of Repeat Region. The Repeat Region Server Behavior then applies the new implementation of Repeat Region if the new data source is selected.

Parent

`group`

Type

Attribute.

Required

No.

Value

The exact name of a data source file within a Configuration/DataSources folder, as shown in the following example:

```
<group serverBehavior="Repeat Region.htm" ¬
dataSource="myCOMdataSource.htm">
```

This group defines a new implementation of Repeat Region to use if you use the COM data source. In `applyServerBehaviors()`, you can indicate that this group should be applied by setting the `MM_dataSource` property on the parameter object, as shown in the following example:

```
function applyServerBehavior(ssRec) {
  var paramObj = new Object();
  paramObj.rs = getComObjectName();
  paramObj.MM_dataSource = "myCOMdataSource.htm";

  dwscripts.applySB(paramObj, sbObj);
}
```

Attribute: subType

Description

This advanced feature supports multiple implementations of a server behavior.

Multiple versions of a server behavior might differ, depending on user selection. When a server behavior is applied, but multiple group files are relevant, the correct group file can be selected by passing in a `subType` value. The group with that specific `subType` is applied.

Parent

`group`

Type

Attribute.

Required

No.

Value

A unique string that determines which group to apply, as shown in the following example:

```
<group serverBehavior="myServerBehavior.htm" ¬
subType="longVersion">
```

This group defines a the long version of `myServerBehavior`. You would also have a version with `subType="shortVersion"`. In `applyServerBehaviors()`, you can indicate which group should be applied by setting the `MM_subType` property on the parameter object, as shown in the following example:

```
function applyServerBehavior(ssRec) {
  var paramObj = new Object();
  if (longVersionChecked) {
    paramObj.MM_subType = "longVersion";
  } else {
    paramObj.MM_subType = "shortVersion";
  }
  dwscripts.applySB(paramObj, sbObj);
}
```

EDML Tag: title

Description

The string that appears in the Server Behaviors panel for each server behavior instance that is found in the current document.

Parent

`group`

Type

Block tag.

Required

No.

Value

A plain text string that can include parameter names to make each instance unique, as shown in the following example:

```
<title>Redirect If Empty (@@recordsetName@@)</title>
```

EDML Tag: groupParticipants

Description

Contains an array of `groupParticipant` declarations.

Parent

`group`

Type

Block tag.

Required

Yes.

Attribute: selectParticipant

Description

Indicates which participant should be selected and highlighted in the document when an instance is selected in the Server Behaviors panel. The server behavior instances that are listed in this panel are ordered by the selected participant, so set `selectParticipant` even if the participant is not visible.

Parent

`groupParticipants`

Type

Attribute.

Required

No.

Value

`participantName` is the exact name (without the .edml extension) of a single participant file that is listed as a group participant, as shown in the following example. See "Attribute: name" on page 163.

```
<groupParticipants selectParticipant="redirectIfEmpty_link">
```

EDML Tag: groupParticipant

Description

Represents the inclusion of a single participant in the group.

Parent

`groupParticipants`

Type

Tag.

Required

Yes (at least one).

Attribute: name

Description

Names a particular participant to be included in the group. The name attribute on the `groupParticipant` tag should be the same as the filename of the participant (without the .edml file extension).

Parent

`groupParticipant`

Type

Attribute.

Required

Yes.

Value

The exact name (without the .edml extension) of any participant file, as shown in the following example:

```
<groupParticipant name="redirectIfEmpty_init">
```

This example refers to the redirectIfEmpty_init.edml file.

Attribute: partType

Description

Indicates the type of participant.

Parent

`groupParticipant`

Type

Attribute.

Required

No.

Values

`identifier`, *`member`*, *`option`*, *`multiple`*, *`data`*

- *`identifier`* is a participant that identifies the entire group. If this participant is found in the document, the group is considered to exist whether or not other group participants are found. This is the default value if `partType` is not specified.
- *`member`* is a normal member of a group. If it is found by itself, it does not identify a group. If it is not found in a group, the group is considered incomplete.
- *`option`* indicates that the participant is optional. If it is not found, the group is still considered complete and no incomplete flag is set in the Server Behaviors panel.
- *`multiple`* indicates that the participant is optional and multiple copies of it can be associated with the server behavior. Any parameters that are unique to this participant are not used when grouping participants because they might have different values.
- *`data`* is a nonstandard participant that is used by programmers as a repository for additional group data. It is ignored by everything else.

Participant EDML files

These tags and attributes are valid within the EDML participant files.

EDML Tag: participant

Description

Contains all specifications for a single participant.

Parent

None.

Type

Block tag.

Required

Yes.

Attribute: version

Description

Defines the version of Dreamweaver that is current with the participant file. For Dreamweaver MX, the version number is 6. If no version is specified, then Dreamweaver assumes 4, or the prior release. All groups and participants that the Server Behavior Builder creates have the version attribute set to 6.

For participant files, this attribute determines if code block merging should occur. For participants without this attribute (or have it set to 4 or earlier), the inserted code blocks are not merged with other code blocks on the page. Participants that have this set to 5 or later are merged with other code blocks on the page when possible. Please note that code-block merging occurs only for participants above and below the HTML tag.

Parent

`participant`

Type

Attribute.

Required

No.

EDML Tag: quickSearch

Description

A simple search string that is used for performance reasons. It cannot be a regular expression. If the string is found in the current document, the rest of the search patterns are called to locate specific instances. This string can be empty to always use the search patterns.

Parent

`participant`

Type

Block tag.

Required

No.

Value

searchString is a literal string that exists on the page if the participant exists. It should be as unique as possible, but it does not have be definitively unique. It is not case-sensitive, but be careful with nonessential spaces that can be changed by the user, as shown in the following example:

```
<quickSearch>Response.Redirect</quickSearch>
```

If `quickSearch` is empty, it is considered to match, and more precise searches use the regular expressions that are defined in the `<searchPattern>` tags. This is helpful if a simple string cannot be used to express a reliable search pattern and regular expressions are required.

EDML Tag: insertText

Description

Provides information about what to insert in the document and where to insert it. Contains the text to be inserted in the document. Parts of the text that are customized should be indicated by `@@parameterName@@`.

In some cases, such as a translator-only participant, you might not need this tag.

Parent

`implementation`

Type

Block tag.

Required

No.

Value

The text to be inserted in the document. If any parts of the text need customizing, they can be passed in later as parameters. Parameters should be embedded in two at (@@) signs. Because this text can interfere with the EDML structure, it should use the `CDATA` construct, as shown in the following example:

```
<insertText location="aboveHTML">
  <![CDATA[<%= @@recordset@@).cursorType %>]]>
</insertText>
```

When the text is inserted, the `@@recordset@@` is replaced by a recordset name that the user supplies. For more information on conditional and repeating code blocks, see the "Adding Custom Server Behaviors" chapter of *Getting Started with Dreamweaver MX*.

Attribute: location

Description

Specifies where the participant text should be inserted. The insert location is related to the `whereToSearch` attribute of the `searchPatterns` tag, so be sure to set both carefully (see "Attribute: whereToSearch" on page 169).

Parent

`insertText`

Type

Attribute.

Required

Yes.

Values

aboveHTML[+weight], *belowHTML[+weight]*, *beforeSelection*, *replaceSelection*, *wrapSelection*, *afterSelection*, *beforeNode*, *replaceNode*, *afterNode*, *firstChildOfNode*, *lastChildOfNode*, *nodeAttribute[+attribute]*

- *aboveHTML[+weight]* inserts the text above the <HTML> tag (suitable only for server code). The weight can be an integer from 1 to 99 and is used to preserve relative order among different participants. By convention, recordsets have weight 50, so if a participant refers to recordset variables, it needs a heavier weight, such as 60, so the code is inserted below the recordset, as shown in the following example:

  ```
  <insert location="aboveHTML+60">
  ```

  If no weight is provided, it is internally assigned a weight of 100 and is added below all specifically weighted participants, as shown in the following example:

  ```
  <insert location="aboveHTML">
  ```

- *belowHTML[+weight]* is similar to the aboveHTML location, except that participants are added below the closing </HTML> tag.
- *beforeSelection* inserts the text before the current selection or insertion point. If there is no selection, it inserts the text at the end of the <BODY> tag.
- *replaceSelection* replaces the current selection with the text. If there is no selection, it inserts the text at the end of the <BODY> tag.
- *wrapSelection* balances the current selection, inserts a block tag before the selection, and adds the appropriate closing tag after the selection.
- *afterSelection* inserts the text after the current selection or insertion point. If there is no selection, it inserts the text at the end of the <BODY> tag.
- *beforeNode* inserts the text before a node, which is a specific location in the DOM. When a function such as dwscripts.applySB() is called to make the insertion, the node pointer must pass in as a parameter of the *paramObj*. The user-definable name of this parameter must be specified by the nodeParamName attribute (see "Attribute: nodeParamName" on page 168).

  In summary, if your location includes the word node, make sure that you declare the <nodeParamName> tag.

- *replaceNode* replaces a node with the text.
- *afterNode* inserts the text after a node.
- *firstChildOfNode* inserts the text as the first child of a block tag; for example, if you want to insert something at the beginning of a FORM tag.

- *lastChildOfNode* inserts the text as the last child of a block tag; for example, if you want to insert something at the end of a FORM tag (useful for adding hidden form fields).
- *nodeAttribute[+attribute]* sets an attribute of a tag node. If the attribute does not already exist, it is created.

  For example, use <insert location="nodeAttribute+ACTION" nodeParamName="form"> to set the ACTION attribute of a form. This changes the user's FORM tag from <form> to <form action="myText">.

  If no attribute is given, the *nodeAttribute* location causes the text to be added directly to the open tag. For example, use insert location="nodeAttribute" to add an optional attribute to a tag. This can be used to change a user's INPUT tag from <input type="checkbox"> to <input type="checkbox" <%if(foo)Reponse.Write("CHECKED")%>>.

  Note: For location="nodeAttribute", the last search pattern is used to determine where the attribute starts and ends. Make sure that the last pattern finds the entire statement.

Attribute: nodeParamName

Description

Used only for node-relative insert locations; indicates the name of the parameter that is used to pass in the node at insertion time.

Parent

insertText

Type

Attribute.

Required

Only if the insert location has the word node in it.

Value

tagtype__Tag is a user-specified name for the node parameter that passes with the parameter object to the dwscripts.applySB() function. For example, if you insert some text into a form, you might use a parameter called form__tag. In your server behavior applyServerBehavior() function, you could use *form__tag* to indicate the exact form to update, as shown in the following example:

```
function applyServerBehavior(ssRec) {
  var paramObj = new Object();
  paramObj.rs = getRecordsetName();
  paramObj.form__tag = getFormNode();
  dwscripts.applySB(paramObj, sbObj);
}
```

You would indicate the form__tag node parameter in your EDML file, as shown in the following example:

```
<insertText location="lastChildOfNode" nodeParamName="form__tag">
  <![CDATA[<input type="hidden" name="MY_DATA">]]>
</insertText>
```

The text is inserted as the lastChildOfNode, and the specific node passes in using the form__tag property of the parameter object.

EDML Tag: searchPatterns

Description

Provides information about how to find the participant text in the document and contains a list of patterns that are used when searching for a participant. If multiple search patterns are defined, they all must be found within the text being searched (the search patterns have a logical AND relationship), unless they are marked as optional using the `isOptional` flag.

Parent

`implementation`

Type

Block tag.

Required

No.

Attribute: whereToSearch

Description

Specifies where to search for the participant text. This is related to the insert location, so be sure to set them both carefully (see "Attribute: location" on page 166).

Parent

`searchPatterns`

Type

Attribute.

Required

Yes.

Values

`directive`, *`tag+tagName`*, *`tag+*`*, *`comment`*, *`text`*

- *`directive`* searches all server directives (server-specific tags). For ASP and JSP, this means search all `<% ... %>` script blocks.

  Note: Tag attributes are not searched, even if they contain directives.

- *`tag+tagName`* searches the contents of a specified tag, as shown in the following example:

  ```
  <searchPatterns whereToSearch="tag+FORM">
  ```

  This example indicates that only form tags should be searched. By default, the entire `outerHTML` is searched. For `INPUT` tags, specify the type after a slash (/). In this example, to search all submit buttons, enter the following code:

  ```
  <searchPatterns whereToSearch="tag+INPUT/SUBMIT">.
  ```

- *`tag+*`* searches the contents of the any tag, as shown in the following example:

  ```
  <searchPatterns whereToSearch="tag+*">
  ```

  This example indicates that all tags should be searched.

- *comment* searches only within the HTML comments <! . . . >, as shown in the following example:

  ```
  <searchPatterns whereToSearch="comment">
  ```

  This example indicates that tags such as <!-- my comment here --> are searched.

- *text* searches only within raw text sections, as shown in the following example:

  ```
  <searchPatterns whereToSearch="text">
    <searchPattern>XYZ</searchPattern>
  </searchPatterns>
  ```

  This example finds a text node that contains the text XYZ.

EDML Tag: searchPattern

Description

A pattern that is used to identify participant text and extract parameter values from it. Each parameter subexpression must be wrapped in parentheses ().

You can have patterns with no parameters (which is used to identify participant text), patterns with one parameter, or patterns with many parameters. All non-optional patterns must be found, and each parameter must be named and found exactly once.

For more information about using searchPattern, see "Finding server behaviors" on page 183.

Parent

searchPatterns

Type

Block tag.

Required

Yes.

Values

searchString, */regularExpression/*, *<empty>*

- *searchString* is a simple search string that is case-sensitive. It cannot be used to extract parameters.
- */regularExpression/* is a regular expression search pattern.
- *<empty>* is if no pattern is given. It is always considered a match, and the entire value is assigned to the first parameter.

  For example, to identify the participant text <%= RS1.Field.Items("author_id") %>, you could define a simple pattern, followed by a precise pattern that also extracts the two parameter values:

  ```
  <searchPattern>Field.Items</searchPattern>
  <searchPattern paramNames="rs,col">
    <![CDATA[
    /<%=\s*(\w+)\.Field\.Items\("(\w+)"\)/
    ]]>
  </searchPattern>
  ```

  This matches the pattern precisely and assigns the value of the first subexpression (\w+) to parameter "rs" and the second subexpression (\w+) to parameter "col".

Note: It is important that the regular expression start and end with a slash (/). Otherwise it is used as a literal string search. Regular expressions can be followed by the regular expression modifier "i" to indicate case-insensitivity (as in /pattern/i). For example, VBScript is not case-sensitive, so it should use /pattern/i. JavaScript is case-sensitive and should use /pattern/.

Sometimes you might want to assign the entire contents of the limited search location to a parameter. In that case, provide no pattern, as shown in the following example:

```
<searchPatterns whereToSearch="tag+OPTION">
  <searchPattern>MY_OPTION_NAME</searchPattern>
  <searchPattern paramNames="optionLabel" limitSearch="innerOnly">
  </searchPattern>
</searchPatterns>
```

This sets parameter "optionLabel" to the entire innerHTML of an OPTION tag.

Attribute: paramNames

Description

A comma-separated list of parameter names whose values are being extracted. These are assigned in the order of the subexpression. You can assign single parameters or use a comma-separated list to assign multiple parameters. If other parenthetical expressions are used but do not indicate parameters, extra commas can be used as placeholders in the Parameter Name list.

The parameter names should match the ones that are specified in the insertion text and the update parameters.

Parent

searchPattern

Type

Attribute.

Required

Yes.

Values

paramName1, *paramName2*, ...

Each parameter name should be the exact name of a parameter that is used in the insertion text. For example, if the insertion text contains @@p1@@, you should define exactly one parameter with that name:

```
<searchPattern paramNames="p1">patterns</searchPattern>
```

To extract multiple parameters using a single pattern, use a comma-separated list of parameter names, in the order that the subexpressions appear in the pattern. Suppose the following example shows your search pattern:

```
<searchPattern paramName="p1,,p2">/(\w+)_(BIG|SMALL)_(\w+)/¬
</searchPattern>
```

There are two parameters (with some text in between them) to extract. Given the text: <%= a_BIG_b %>, the first subexpression in the search pattern matches "a", so p1="a". The second subexpression is ignored (note the ,, in the paramName value). The third subexpression will match "b", so p2="b".

Attribute: limitSearch

Description

Limits the search to some part of the `whereToSearch` tag.

Parent

`searchPattern`

Type

Attribute.

Required

No.

Values

`all`, *`attribute+attribName`*, *`tagOnly`*, *`innerOnly`*

- *`all`* (default) searches the entire tag that is specified in the `whereToSearch` attribute.
- *`attribute+attribName`* searches only within the value of the specified attribute, as shown in the following example:

```
<searchPatterns whereToSearch="tag+FORM">
  <searchPattern limitSearch="attribute+ACTION">
    /MY_PATTERN/
  </searchPattern>
</searchPatterns>
```

  This example indicates that only the value of the `ACTION` attribute of `FORM` tags should be searched. If that attribute is not defined, the tag is ignored.

- *`tagOnly`* searches only the outer tag and ignores the `innerHTML`. It is valid only if `whereToSearch` is a tag.
- *`innerOnly`* searches only the `innerHTML` and ignores the outer tag. It is valid only if `whereToSearch` is a tag.

Attribute: isOptional

Description

A flag that indicates that the search pattern is not required to find the participant. This is useful for complex participants that might have noncritical parameters to extract. You can create some patterns for distinctly identifying a participant and have some optional patterns for extracting noncritical parameters.

Parent

`searchPattern`

Type

Attribute.

Required

No.

Values

`true`, *`false`*

- *true* if the searchPattern does not have to be found to identify the participant.
- *false* (default) if the searchPattern must be found.

For example, consider the following simple recordset string:

```
<%
var Recordset1 = Server.CreateObject("ADODB.Recordset");
Recordset1.ActiveConnection = "dsn=andescoffee;";
Recordset1.Source = "SELECT * FROM PressReleases";
Recordset1.CursorType = 3;
Recordset1.Open();
%>
```

The search patterns must identify the participant and extract several parameters. However, if a parameter such as *cursorType* is not found, you should still recognize this as a recordset. The *cursor* parameter is optional. In the EDML, the search patterns might look like the following example:

```
<searchPattern paramNames="rs">/var (\w+) = Server.CreateObject/
</searchPattern>
<searchPattern paramNames="src">/ActiveConnection = "([^\r\n]*)"/¬
</searchPattern>
<searchPattern paramNames="conn">/Source = "([^\r\n]*)"/¬
</searchPattern>
<searchPattern paramNames="cursor" isOptional="true">¬
/CursorType = (\d+)/
</searchPattern>
```

The first three patterns are required to identify the recordset. If the last parameter is not found, the recordset is still identified.

EDML Tag: updatePatterns

Description

This optional advanced feature allows precise updates of the participant. Without this tag, the participant is updated automatically by replacing the entire participant text each time. If you specify an <updatePatterns> tag, it must contain specific patterns to find and replace each parameter within the participant.

This tag is beneficial if the user edits the participant text. It performs precise updates only to the parts of the text that need changing.

Parent

implementation

Type

Block tag.

Required

No.

EDML Tag: updatePattern

Description

A specific type of regular expression that allows precise updates of participant text. There should be at least one update pattern definition for every unique parameter that is declared in the insertion text (of the form `@@paramName@@`).

Parent

`updatePatterns`

Type

Block tag.

Required

Yes (at least one, if the `updatePatterns` tag is declared).

Values

A regular expression that finds a parameter between two parenthetical subexpressions, in the form */(pre-pattern)parameter-pattern(post-pattern)/*. You need at least one update pattern defined for each unique `@@paramName@@` in the insertion text. The following example shows how your insertion text might look:

```
<insertText location="afterSelection">
  <![CDATA[<%= @@rs@@.Field.Items("@@col@@") %>]]>
</insertText>
```

A particular instance of it on a page might look like the following example:

```
<%= RS1.Field.Items("author_id") %>
```

There are two parameters, `rs` and `col`. To update this text after it is inserted on the page, you need two update pattern definitions:

```
<updatePattern paramName="rs" >
  /(\b)\w+(\.Field\.Items)/
</updatePattern>
<updatePattern paramName="col">
  /(\bItems\(")\w+("\))/
</updatePattern>
```

The literal parentheses, as well as other special regular expression characters, are escaped by preceding them with a backslash (\). The middle expression, defined as `\w+`, is updated with the latest value that passed in for parameters `"rs"` and `"col"`, respectively. The values `"RS1"` and `"author_id"` can be precisely updated with new values.

Multiple occurrences of the same pattern can be updated simultaneously by using the regular expression global flag `"g"` after the closing slash, such as */pattern/g*.

If the participant text is long and complex, you might need multiple patterns to update a single parameter, as shown in the following example:

```
<% ...
  Recordset1.CursorType = 0;
  Recordset1.CursorLocation = 2;
  Recordset1.LockType = 3;
%>
```

To update the recordset name in all three positions, you need three update patterns for a single parameter, as shown in the following example:

```
<updatePattern paramName="rs">
  /(\b)\w+(\.CursorType)/
</updatePattern>
<updatePattern paramName="rs">
  /(\b)\w+(\.CursorLocation)/
</updatePattern>
<updatePattern paramName="rs">
  /(\b)\w+(\.LockType)/
</updatePattern>
```

Now you can pass in a new value for the recordset, and it is precisely updated in three locations.

Attribute: paramName

Description

Indicates the name of the parameter whose value is used to update the participant. This parameter should match the ones that are specified in the insertion text and search parameters.

Parent

`updatePattern`

Type

Attribute.

Required

Yes.

Values

The exact name of a parameter that is used in the insertion text. For example, if the insertion text contains an `@@rs@@`, you should have a parameter with that name:

```
<updatePattern paramName="rs">pattern</updatePattern>
```

EDML Tag: delete

Description

This optional advanced feature gives control over how a participant is deleted. Without this tag, the participant is deleted by removing it completely but only if no server behaviors refer to it. By specifying a `<delete>` tag, you can specify that it should never be deleted or that only portions should be deleted.

Parent

`implementation`

Type

Tag.

Required

No.

Attribute: deleteType

Description

Used to indicate the type of delete to perform. It has different meanings, depending on whether the participant is a directive, a tag, or an attribute. By default, the entire participant is deleted.

Parent

`delete`

Type

Attribute.

Required

No.

Values

`all`, *`none`*, *`tagOnly`*, *`innerOnly`*, *`attribute+attribName`*, *`attribute+*`*

- *`all`* (default) deletes the entire directive or tag. For attributes, it deletes the entire definition.
- *`none`* is never automatically deleted.
- *`tagOnly`* removes only the outer tag but leaves the contents of the tag, `innerHTML`, intact. For attributes, it also removes the outer tag if it is a block tag. It is meaningless for directives.
- *`innerOnly`* when applied to tags, it removes only the contents (the `innerHTML`). For attributes, it removes only the value. It is meaningless for directives.
- *`attribute+attribName`* when applied to tags, it removes only the specified attribute. It is meaningless for directives and attributes.
- *`attribute+*`* removes all attributes for tags. It is meaningless for directives and attributes.

For example, if your server behavior converts selected text into a link, you can remove the link by removing the outer tag only:

```
<delete deleteType="tagOnly"/>
```

This changes a link participant from `<A HREF="...">HELLO</A>` to `HELLO`.

EDML Tag: translator

Description

Provides information for translating a participant so that it can be rendered differently and have a custom Property inspector.

Parent

`implementation`

Type

Block tag.

Required

No.

EDML Tag: searchPatterns

Description

Provides a way for Dreamweaver to find each specified instance in a document. If multiple search patterns are defined, they all must be found within the text being searched (the search patterns have a logical AND relationship), unless they are marked as optional using the `isOptional` flag.

Parent

`translator`

Type

Block tag.

Required

Yes.

EDML Tag: translations

Description

Contains a list of translation instructions where each instruction indicates where to look for the participant and what to do with the participant.

Parent

`translator`

Type

Block tag.

Required

No.

EDML Tag: translation

Description

Contains a single translation instruction that includes the location for the participant, what type of translation to perform, and the content that should replace the participant text.

Parent

`translations`

Type

Block tag.

Required

No.

Attribute: whereToSearch

Description

Specifies where to search for the text. This is related to the insert location, so be sure to set both carefully (see "Attribute: location" on page 166).

Parent

`translation`

Type

Attribute.

Required

Yes.

Attribute: limitSearch

Description

Limits the search to some part of the `whereToSearch` tag.

Parent

`translation`

Type

Attribute.

Required

No.

Attribute: translationType

Description

Indicates the type of translation to perform. These types are preset and give the translation specific functionality. For example, if you specify `"dynamic data"`, anything that is translated should behave the same as Dreamweaver dynamic data. That is, it should have the dynamic data placeholder look in the Design view (curly brace ({}) notation with dynamic background color) and appear in the Server Behaviors panel.

Parent

`translation`

Type

Attribute.

Required

Yes.

Values

dynamic data, *dynamic image*, *dynamic source*, *tabbed region start*, *tabbed region end*, *custom*

- *dynamic data* indicates that the translated directives look and behave the same as Dreamweaver dynamic data, as shown in the following example:

```
<translation whereToSearch="tag+IMAGE"
  limitSearch="attribute+SRC"
  translationType="dynamic data">
```

- *dynamic image* indicates that the translated attributes should look and behave the same as Dreamweaver dynamic images, as shown in the following example:

```
<translation whereToSearch="IMAGE+SRC"
  translationType="dynamic image">
```

- *dynamic source* indicates that the translated directives should behave the same as Dreamweaver dynamic sources, as shown in the following example:

```
<translation whereToSearch="directive"
  translationType="dynamic source">
```

- *tabbed region start* indicates that the translated <CFLOOP> tags define the beginning of a tabbed outline, as shown in the following example:

```
<translation whereToSearch="CFLOOP"
  translationType="tabbed region start">
```

- *tabbed region end* indicates that the translated </CFLOOP> tags define the end of a tabbed outline, as shown in the following example:

```
<translation whereToSearch="CFLOOP"
  translationType="tabbed region end">
```

- *custom* is the default case in which no internal Dreamweaver functionality is added to the translation. It is often used when specifying a tag to insert for a custom Property inspector, as shown in the following example:

```
<translation whereToSearch="directive"
  translationType="custom">
```

EDML Tag: openTag

Description

An optional tag that can be inserted at the beginning of the translation section. This tag lets certain other extensions find the translation, such as custom Property inspectors.

Parent

translation

Type

Block tag.

Required

No.

Values

tagName is a valid tag name. It should be unique to prevent conflicts with known tag types. For example, if you specify `<openTag>MM_DYNAMIC_CONTENT</openTag>` the dynamic data is translated to the tag `<MM_DYNAMIC_CONTENT>`.

EDML Tag: attributes

Description

Contains a list of attributes to add to the translated tag that is specified by `openTag`. Alternatively, if `openTag` is not defined and the `searchPattern` specifies `tag`, this tag contains a list of translated attributes to add to the tag that is found.

Parent

`translation`

Type

Block tag.

Required

No.

EDML Tag: attribute

Description

Specifies a single attribute (or translated attribute) to add to the translated tag.

Parent

`attributes`

Type

Block tag.

Required

Yes (at least one).

Values

attributeName="*attributeValue*" is an attribute set to a value. Typically, the attribute name is fixed, and the value contains some parameter references that are extracted by the parameter patterns, as shown in the following example:

```
<attribute>SOURCE="@@rs@@"</attribute>
<attribute>BINDING="@@col@@"</attribute>
```

or

```
<attribute>
  mmTranslatedValueDynValue="VALUE={@@rs@@.@@col@@}"
</attribute>
```

EDML Tag: display

Description

An optional display string that should be inserted in the translation.

Parent

`translation`

Type

Block tag.

Required

No.

Values

`displayString` is any string comprising text and HTML. It can include parameter references that are extracted by the parameter patterns. For example, `<display>{@@rs@@.@@col@@}</display>` causes the translation to render as `{myRecordset.myCol}`.

EDML Tag: closeTag

Description

An optional tag that should be inserted at the end of the translated section. This enables certain other extensions to find the translation, such as custom Property inspectors.

Parent

`translation`

Type

Block tag.

Required

No.

Values

`tagName` is a valid tag name; it should match a translation *`openTag`*.

Example

If you specify `<closeTag>MM_DYNAMIC_CONTENT</closeTag>`, the dynamic data is translated to end with the `</MM_DYNAMIC_CONTENT>` tag.

Using the Extension Data Manager

The APIs in this section comprise the Extension Data Manager (EDM). You can programmatically access and manipulate the data that is contained in the group and participant files by calling these functions. The EDM performs in the following manner:

- The EDM performs all EDML file input/output for group and participant files.
- The EDM acts as a server model filter by performing all data requests for the current server model.

dreamweaver.getExtDataValue()

Availability

Dreamweaver UltraDev 4

Description

Retrieves the field values from an EDML file for the specified nodes.

Arguments

qualifier(s) is a variable-length list of comma-separated node qualifiers that includes group or participant name, subblock (if any), and field name.

Returns

Field value is returned. If value is not specified, then the default value returns.

dreamweaver.getExtDataArray()

Availability

Dreamweaver UltraDev 4

Description

Retrieves an array of values from an EDML file for the specified nodes.

Arguments

qualifier(s) is a variable-length list of comma-separated node qualifiers, including group or participant name, subblock (if any), and field name.

Returns

Array of child node names.

dreamweaver.getExtParticipants()

Availability

Dreamweaver UltraDev 4

Description

Retrieves the list of participants from an EDML group file or participant files.

Arguments

value, *qualifier(s)*

- *value* is a property value or blank to ignore.
  For example `dw.getExtParticipants("", "participant");`
- *qualifier(s)* is a variable-length list of comma-separated node qualifiers of required property.

Returns

Array of participant names that have the specified property, if given, and the property matches the specified value, if given.

dreamweaver.getExtGroups()

Availability

Dreamweaver UltraDev 4

Description

Retrieves the name of the group, which is the equivalent to the server behavior's name, from an EDML group file.

Arguments

value, *qualifier(s)*

- *value* is a property value or blank to ignore.
- *qualifier(s)* is a variable length list of comma-separated node qualifiers of required property.

Returns

Array of group names that have the specified property, if given, and the property matches the specified value, if given.

dreamweaver.refreshExtData()

Availability

Dreamweaver UltraDev 4

Description

Reloads all extension data files.

Tip: You can make a useful command from this function, letting edits to server behavior EDML files be reloaded without restarting Dreamweaver MX.

Arguments

None.

Returns

Reloaded data.

Server behavior techniques

This section covers the common and advanced techniques that are used to create and edit server behaviors. Most of the suggestions involve specific settings in the EDML files.

Finding server behaviors

Writing search patterns In order to update or delete server behaviors, you must provide a way for Dreamweaver to find each instance in a document. This requires a `quickSearch` tag and at least one `searchPattern` tag, which is contained within the `searchPatterns` tag.

The `quickSearch` tag should be a string, not a regular expression, that indicates that the server behavior might exist on the page. It is not case-sensitive. It should be short and unique, and avoid spaces and other sections that can be changed by the user. The following example shows a participant that consists of the simple ASP JavaScript tag:

```
<% if (Recordset1.EOF) Response.Redirect("some_url_here") %>
```

In this case, the following `quickSearch` string checks for it:

```
<quickSearch>Response.Redirect</quickSearch>
```

For performance reasons, the `quickSearch` pattern is the beginning of the process of finding server behavior instances. If this string is found in the document and the participant identifies a server behavior (in the group file, `partType="identifier"` for this participant), the related server behavior files are loaded and `findServerBehaviors()` is called. If your participant has no reliable strings for which to search (or for debugging purposes), you can leave the `quickSearch` string empty, as shown in the following example:

```
<quickSearch></quickSearch>
```

In this case, the server behavior is always loaded and can search the document.

Next, the `searchPattern` tag searches the document more precisely and extracts parameter values from the participant code. The search patterns specify where to search (the `whereToSearch` attribute) with a series of `searchPattern` tags that contain specific patterns. These patterns can use simple strings or regular expressions. The previous example code is an ASP directive, so `whereToSearch="directive"`, and a regular expression identifies the directive and extracts the parameters, as shown in the following example:

```
<quickSearch>Response.Write</quickSearch>
<searchPatterns whereToSearch="directive">
  <searchPattern paramNames="rs,new__url">
    /if\s*\((\w+)\.EOF\)\s*Response\.Redirect\("([^\r\n]*)"\)/i
  </searchPattern>
</searchPatterns>
```

The search string is defined as a regular expression by starting and ending with a slash (/), and is followed by `i` so that it is not case-sensitive. Within the regular expression, special characters such as parentheses () and periods (.) are escaped by preceding them with a backslash (\). The two parameters `rs` and `new__url` are extracted from the string by using parenthetical subexpressions. (The parameters must be enclosed in parentheses.) In this example, they are indicated by `(\w+)` and `([^\r\n]*)`: These values correspond to the regular expression values that are normally returned by `$1` and `$2`.

Optional search patterns There might be cases where you want to identify a participant even if some parameters are not found. You might have a participant that stores some optional information such as a telephone number. For such an example, you could use the following ASP code:

```
<% //address block
  LNAME = "joe";
  FNAME = "smith";
  PHONE = "123-4567";
%>
```

You could use the following search patterns:

```
<quickSearch>address</quickSearch>
<searchPatterns whereToSearch="directive">
  <searchPattern paramNames="lname">/LNAME\s*=\s*"([^\r\n]*)"/i¬
  </searchPattern>
  <searchPattern paramNames="fname">/FNAME\s*=\s*"([^\r\n]*)"/i¬
  </searchPattern>
<searchPattern paramNames="phone">/PHONE\s*=\s*"([^\r\n]*)"/i¬
</searchPattern>
</searchPatterns>
```

In the previous example, the telephone number must be specified. However, you can make the telephone number optional, by adding the `isOptional` attribute, as shown in the following example:

```
<quickSearch>address</quickSearch>
<searchPatterns whereToSearch="directive">
  <searchPattern paramNames="lname">/LNAME\s*=\s*"([^\r\n]*)"/i¬
  </searchPattern>
  <searchPattern paramNames="fname">/FNAME\s*=\s*"([^\r\n]*)"/i¬
  </searchPattern>
  <searchPattern paramNames="phone" isOptional="true">¬
  /PHONE\s*=\s*"([^\r\n]*)"/i
  </searchPattern>
</searchPatterns>
```

Now the participant is recognized, even if the telephone number is not found.

How participants are matched If a server behavior has more than one participant, the participants must be identified in the user's document and matched. If the user applies multiple instances of the server behavior to a document, each group of participants must be matched accordingly. To ensure participants are matched correctly, you might need to change or add parameters and construct participants so they can be uniquely identified.

Matching requires some rules. Participants are matched when all parameters with the same name have the same value. Above and below the `<html>` tag, there can be only one instance of a participant with a given set of parameter values. Within the `<html>...</html>` tags, participants are also matched by their position relative to the selection or to common nodes that are used for insertion.

Participants without parameters are automatically matched, as shown in this example of a server behavior with group file:

```
<group serverBehavior="test.htm">
  <title>Test</title>
  <groupParticipants>
    <groupParticipant name="test_p1" partType="identifier" />
    <groupParticipant name="test_p2" partType="identifier" />
  </groupParticipants>
</group>
```

This example inserts two simple participants above the `<html>` tag:

```
<% //test_p1 %>
<% //test_p2 %>
<html>
```

These participants are found and matched, and Test appears once in the Server Behaviors panel. If you add the server behavior again, nothing is added because the participants already exist.

If the participants have unique parameters, multiple instances can be inserted above the `<html>` tag. For example, by adding a name parameter to the participant, a user can enter a unique name in the Test Server Behavior dialog box. If the user enters name `"aaa"`, the following participants are inserted:

```
<% //test_p1 name="aaa" %>
<% //test_p2 name="aaa" %>
<html>
```

If you add the server behavior again with a different name, such as "bbb", the document now looks like this:

```
<% //test_p1 name="aaa" %>
<% //test_p2 name="aaa" %>
<% //test_p1 name="bbb" %>
<% //test_p2 name="bbb" %>
<html>
```

There are two instances of Test listed in the Server Behaviors panel. If the user tries to add a third instance to the page and names it "aaa", nothing is added because it already exists.

Within the <html> tag, matching can also use position information. In the following example, there are two participants, one that is added before the selection and another that is added after the selection:

```
<% if (expression) { //mySBName %>
  Random HTML selection here
<% } //end mySBName %>
```

These are two participants without parameters, so they are grouped together. However, you can add another instance of this server behavior elsewhere in the HTML, as shown in the following example:

```
<% if (expression) { //mySBName %>
  Random HTML selection here
<% } //end mySBName %>
  More HTML here...
<% if (expression) { //mySBName %>
  Another HTML selection here
<% } //end mySBName %>
```

Now there are two identical instances of each participant, which is allowed within the HTML. They are matched by the order in which they occur in the document.

The following example shows a matching problem and how to avoid it. You can create a participant that computes the tax on some dynamic data and displays the result at the selection.

```
<% total = Recordset1.Fields.Item("itemPrice").Value * 1.0825 %>
<html>
<body>
  The total (with taxes) is $<%=total%>
</body>
</html>
```

The two participants are matched because they have no common parameters. However, if you add a second instance of this server behavior, you should have the following code:

```
<% total = Recordset1.Fields.Item("itemPrice").Value * 1.0825 %>
<% total = Recordset1.Fields.Item("salePrice").Value * 1.0825 %>
<html>
<body>
  The total  (with taxes) is $<%=total%>
  Sale price (with taxes) is $<%=total%>
</body>
</html>
```

This server behavior no longer works correctly because only one parameter is named `total`. To solve this problem, make sure that there is a parameter with a unique value and can be used to match the participants. In the following example, you could make the `total` variable name unique using the column name:

```
<% itemPrice_total = Recordset1.Fields.Item("itemPrice").¬
Value * 1.0825 %>
<% salePrice_total = Recordset1.Fields.Item("salePrice").¬
Value * 1.0825 %>
<html>
<body>
  The total  (with taxes) is $<%=itemPrice_total%>
  Sale price (with taxes) is $<%=salePrice_total%>
</body>
</html>
```

The search patterns now uniquely identify and match the participants.

Search pattern resolution

Dreamweaver MX supports the following actions by using the participant `searchPatterns` functionality:

- File transfer dependency
- Updating the file paths for any file reference (such as those for include files)

When Dreamweaver MX creates server models, it builds lists of patterns by scanning all the participants for special `paramNames`. To find URLs to check file dependency and to fix the pathname, Dreamweaver MX uses each `searchPattern` tag in which one of the `paramNames` attribute ends with `_url`. Multiple URLs can be specified in a single `searchPattern`.

For each translator `searchPattern` that has a `paramNames` attribute value that ends with `_includeUrl`, Dreamweaver MX uses that `searchPattern` to translate include file statements on the page. Dreamweaver MX uses a different suffix string to identify include file URLs because not all URL references are translated. Also, only a single URL can be translated as an include file.

In resolving a `searchPatterns` tag, Dreamweaver uses the following algorithm:

1. Look for `whereToSearch` attribute within the `searchPatterns` tag.
2. If the attribute value starts with `tag+`, the remaining string is assumed to be the tag name (no spaces are allowed in the tag name).
3. Look for `limitSearch` attribute within the `searchPattern` tag.
4. If the attribute value starts with `attribute+`, the remaining string is assumed to be the attribute name (no spaces are allowed in the attribute name).

If these four steps are successful, Dreamweaver MX assumes a tag/attribute combination; otherwise, Dreamweaver starts looking for `searchPattern` tags with a `paramName` that has a `_url` suffix and a regular expression that is defined. (For information about regular expressions, see the “Regular expressions” on page 158.)

The following example of a `searchPatterns` tag has no search pattern because it combines a tag (`cfinclude`) with an attribute (`template`) to isolate the URL for dependency file checking, path fixing, and so forth:

```
<searchPatterns whereToSearch="tag+cfinclude">
   <searchPattern paramNames="include_url" limitSearch="attribute+template" />
</searchPatterns>
```

The tag/attribute combination (see the previous example) does not apply to translation because Dreamweaver always translates to straight text in the JavaScript layer, whereas file dependency checking, path fixing, and so on occurs in the C layer. In the C layer, Dreamweaver internally splits the document into directives (straight text) and tags (parsed into an efficient tree structure).

Updating server behaviors

Replacement update By default, participant EDML files do not have an `<updatePatterns>` tag, and instances of the participant are updated in the document by replacing them entirely. When a user edits an existing server behavior and clicks OK, any participant that contains a parameter whose value has changed is removed and reinserted with the new value in the same location.

If the user customizes participant code in the document, the participant might not be recognized if the search patterns look for the old code. Shorter search patterns can let the user customize the participant code in their document; however, updating the server behavior instance can cause the participant to be replaced, which loses the custom edits.

Precision update In some cases, it can be desirable to let users customize the participant code after it is inserted in the document. This can be achieved by limiting the search patterns and providing update patterns in the EDML file. After the participant is added to the page, only specific parts of it are updated by the server behavior. The following example shows a simple participant with two parameters:

```
<% if (Recordset1.EOF) Response.Redirect("some_url_here") %>
```

The example might use the following search patterns:

```
<quickSearch>Response.Write</quickSearch>
<searchPatterns whereToSearch="directive">
  <searchPattern paramNames="rs,new__url">
    /if\s*\((\w+)\.EOF\)\s*Response\.Redirect\("([^\r\n]*)"\)/i
  </searchPattern>
</searchPatterns>
```

The user might add another test to a particular instance of this code, as shown in the following example:

```
<% if (Recordset1.EOF || x > 2) Response.Redirect("some_url_here") %>
```

The search patterns fail because they are looking for a parenthesis after the `EOF`. To make the search patterns more forgiving, you can shorten them by splitting them up:

```
<quickSearch>Response.Write</quickSearch>
<searchPatterns whereToSearch="directive">
  <searchPattern paramNames="rs">/(\w+)\.EOF/</searchPattern>
  <searchPattern paramNames="new__url">
    /if\s*\([^\r\n]*\)\s*Response\.Redirect\("([^\r\n]*)"/i
  </searchPattern>
</searchPatterns>
```

These shortened search patterns are flexible, so the user can add to the code. However, if the server behavior changes the URL, when the user clicks OK, the participant is replaced, and the customizations are lost. To update more precisely, add an `updatePatterns` tag that contains a pattern for updating each parameter:

```
<updatePatterns>
  <updatePattern paramNames="rs">/(\b)\w+(\.EOF)/¬
  </updatePattern>
  <updatePattern paramNames="new__url">
    /(Response\.Redirect\(")[^\r\n]*(")/i
  </updatePattern>
</updatePatterns>
```

In update patterns, the parentheses are reversed and are placed around the text before and after the parameter. For search patterns, use `textBeforeParam(param)textAfterParam`. For update patterns, use `(textBeforeParam)param(textAfterParam)`. All the text between the two parenthetical subexpressions is replaced with the new value for the parameter.

Deleting server behaviors

Default deletion and dependency counts The user can delete an instance that is selected in the Server Behaviors panel by clicking the minus (-) button or pressing Delete. All the participants are removed except for the ones that are shared by other server behaviors. Specifically, if more than one server behavior has a participant pointer to the same node, the node is not deleted.

By default, participants are deleted by removing an entire tag. If the insert location is `"wrapSelection"`, only the outer tag is removed. For attributes, the entire attribute declaration is removed. The following example shows an attribute participant on the `ACTION` attribute of a `form` tag:

```
<form action="<% my_participant %>">
```

After deleting, only `<form>` remains.

Using delete flags to limit participant deletion There might be cases where you want to limit the way that participants are deleted. This can be achieved by adding a delete tag to the EDML file. The following example shows a participant that is an `href` attribute of a link:

```
<a href="<%=MY_URL%>">Link Text</a>
```

When this attribute participant is deleted, the resulting tag is `<a>Link Text</a>`, which no longer appears as a link in Dreamweaver. It might be preferable to delete only the attribute value, which can be done by adding the following tag to the participant EDML file:

```
<delete deleteType="innerOnly"/>
```

Another approach is to remove the entire tag when the attribute is deleted by typing `<delete deleteType="tagOnly"/>`, and the resulting text is `Link Text`.

Avoiding conflicts with share-in-memory JavaScript files

If several HTML files reference a particular JavaScript file, Dreamweaver loads the JavaScript into a central location where the HTML files can share the same JavaScript source. These files contain the following line:

```
//SHARE-IN-MEMORY=true
```

If a JavaScript file has the `SHARE-IN-MEMORY` directive and an HTML file references it (by using the `SCRIPT` tag with the `SRC` attribute), Dreamweaver loads the JavaScript into a memory location where the code is implicitly included in all HTML files thereafter.

Note: Because JavaScript files loaded into this central location share memory, the files cannot duplicate any declarations. If a share-in-memory file defines a variable or function and any other JavaScript file defines the same variable or function, a name conflict occurs. When writing new JavaScript files, be aware of these files and their naming conventions.

CHAPTER 15
Data Sources

The Dreamweaver MX Data Sources API functions let you add data sources, which appear in the plus (+) menu of the Bindings panel (see "dreamweaver.dbi.getDataSources" on page 408).

Data source files are stored in the Configuration/DataSources folder. Each of the following server models has its own folder: ASP.Net/C#, ASP.Net/VisualBasic, ASP/JavaScript, ASP/VBScript, ColdFusion, JSP, and PHP/MySQL. Within each server model subfolder are HTML and EDML files that are associated with the data sources for that server model.

How data sources work

Dreamweaver users can add dynamic data by using the Bindings panel. The dynamic data objects shown on the plus (+) menu are based on the server model that is specified for the page. For example, users can insert recordsets, commands, request variables, session variables, and application variables for ASP applications.

The following steps describe the process that is involved in adding dynamic data:

1 When the user clicks the plus (+) menu in the Bindings panel, a pop-up menu appears.

To determine the contents of the menu, Dreamweaver first looks for a DataSources.xml file in the same folder as the data sources (for example, Configuration/DataSources/ASP_Js/DataSources.xml). The DataSources.xml file describes the contents of the pop-up menu; it contains references to the HTML files that should be placed in the pop-up menu.

Dreamweaver checks each referenced HTML file for a title tag. If the file contains a title tag, the content of the title tag appears in the menu. If the file does not contain a title tag, the filename is used in the menu.

After Dreamweaver finishes reading the DataSources.xml file or if the file does not exist, Dreamweaver scans the rest of the folder to find other items that should appear in the menu. If Dreamweaver finds files in the main folder that aren't in the menu, it adds them to the menu. If subfolders contain files that aren't in the menu, Dreamweaver creates a submenu and adds those files to the submenu.

2 When the user chooses an item from the plus (+) menu, Dreamweaver calls the `addDynamicSource()` function, so that code for the data source is added to the user's document.

3. Dreamweaver goes through each file in the appropriate server model folder, calling `findDynamicSources()` in each file. For each value in the returned array, Dreamweaver calls the `generateDynamicSourceBindings()` function in the same file to get a fresh list of all the fields in each data source for the user's document. Those fields are presented to the user as a tree control in the Dynamic Data or Dynamic Text dialog box or the Bindings panel. The data source tree for an ASP document might appear as shown in the following example:

```
Recordset (Recordset1)
  ColumnOneInRecordset
  ColumnTwoInRecordset
Recordset (Recordset2)
  ColumnOfRecordset
Request
  NameOfRequestVariable
  NameOfAnotherRequestVariable
Session
  NameOfSessionVariable
```

4. If the user double-clicks on a data source name in the Bindings panel to edit the data source, Dreamweaver calls `editDynamicSource()` to handle user edits within the tree.

5. If the user clicks the minus (-) button, Dreamweaver gets the current node selection from the tree and passes it to `deleteDynamicSource()`, which deletes the code that was added earlier with `addDynamicSource()`. If it does not make sense to delete the current selection, the function returns an error message. After `deleteDynamicSource()` returns, Dreamweaver refreshes the data source tree by calling `findDynamicSources()` and `generateDynamicSourceBindings()`.

6. If the user chooses a data source and clicks OK in the Dynamic Data or Dynamic Text dialog box, or clicks Insert or Bind in the Bindings panel, Dreamweaver calls `generateDynamicDataRef()`. The return value is inserted in the document at the current insertion point.

7. If the user displays the Dynamic Data or Dynamic Text dialog box to edit an existing dynamic data object, the selection in the data source tree needs to be initialized to the dynamic data object. To initialize the tree control, Dreamweaver goes through each file in the appropriate server model folder (for example, the Configuration/DataSources/ASP_Js folder), calling the implementation of `inspectDynamicDataRef()` in each file.

   Dreamweaver calls the `inspectDynamicDataRef()` function to convert the dynamic data object back from the code in the user's document to an item in the tree. (This process is the reverse of what occurs when `generateDynamicDataRef()` is called.) If `inspectDynamicDataRef()` returns an array that contains two elements, Dreamweaver provides a visual cue, showing which item in the tree is bound to the current selection.

8. Every time the user changes the selection, Dreamweaver calls the `inspectDynamicDataRef()` function to determine whether the new selection is dynamic text or a tag with a dynamic attribute. If it is dynamic, Dreamweaver displays the bindings for the current selection in the Bindings panel.

9. Using the Bindings panel or the Dynamic Data or Dynamic Text dialog box, it's possible to change the data format for a dynamic text object or a dynamic attribute that the user has already added to the page. When the format is changed, Dreamweaver calls `generateDynamicDataRef()` to get the string to insert into the user's document and passes that string to `formatDynamicDataRef()` (described in "Server Formats" on page 199). The string that returns from `formatDynamicDataRef()` is inserted in the user's document.

The Data Sources API

addDynamicSource()

Availability

Dreamweaver UltraDev 1

Description

Adds a dynamic data source. Because there is one implementation of this function in each data source file, Dreamweaver calls the appropriate implementation of the `addDynamicSource()` function when a data source is selected from the plus (+) menu.

For example, for recordsets or commands, Dreamweaver calls `dw.serverBehaviorInspector.popupServerBehavior()`, which inserts a new server behavior into the document. For request, session, and application variables, Dreamweaver displays an HTML/JavaScript dialog box to collect the name of the variable; the behavior stores the variable name for future use.

After the `addDynamicSource()` function returns, Dreamweaver erases the contents of the data source tree and calls the `findDynamicSources()` and `generateDynamicSourceBindings()` functions to repopulate the data source tree.

Returns

Dreamweaver expects nothing.

deleteDynamicSource()

Availability

Dreamweaver UltraDev 1

Description

Called when a Dreamweaver user selects a data source in the tree and clicks the minus (-) button.

For example, in Dreamweaver, if the selection is a recordset or command, `deleteDynamicSource()` calls `dw.serverBehaviorInspector.deleteServerBehavior()`. If the selection is a request, session, or application variable, the function remembers that the variable was deleted and does not display it any more. After the `deleteDynamicSource()` function returns, Dreamweaver erases the contents of the data source tree and calls `findDynamicSources()` and `generateDynamicSourceBindings()` to get a fresh list of all the data sources for the user's document.

Arguments

`sourceName`, *`bindingName`*

- *`sourceName`* is the name of the top-level node to which the child node is associated.
- *`bindingName`* is the name of the child node.

Returns

Dreamweaver expects nothing.

displayHelp()

Description

If this function is defined, a Help button appears below the OK and Cancel buttons in the dialog box. This function is called when the user clicks the Help button.

Arguments

None.

Returns

Dreamweaver expects nothing.

Example

```
// the following instance of displayHelp() opens
// in a browser a file that explains how to use
// the extension.
function displayHelp(){
  var myHelpFile = dw.getConfigurationPath() +
    '/ExtensionsHelp/superDuperHelp.htm';
  dw.browseDocument(myHelpFile);
}
```

editDynamicSource()

Availability

Dreamweaver MX

Description

Called when the user double clicks on a data source name in the Bindings panel to edit the data source. An extension developer can implement this function to handle user edits within the tree. Otherwise, the server behavior that matches the data source is automatically invoked. The extension developer can use this function to override the default implementation of server behaviors and provide a custom handler.

Arguments

sourceName, *bindingName*

- *sourceName* is the name of the top-level node to which the child node is associated.
- *bindingName* is the name of the child node.

Returns

Dreamweaver expects a Boolean value that indicates whether the function has handled the edit (`true`) or not (`false`).

findDynamicSources()

Availability

Dreamweaver UltraDev 1

Description

Returns the top-level nodes from the data source tree that appears in the Dynamic Data or Dynamic Text dialog box or the Bindings panel. Each data source file has an implementation of the `findDynamicSources()` function. When Dreamweaver refreshes the tree, Dreamweaver reads through all the files in the DataSources folder and calls the `findDynamicSources()` function in each file.

Returns

Dreamweaver expects an array of JavaScript objects where each object can have as many as five attributes:

1. The `title` property is the label string that appears to the right of the icon for each parent node. The `title` property is always required.
2. The `imageFile` property is the path of a file that contains the icon (a GIF image), which represents the parent node in the tree control in the Bindings panel or the Dynamic Data or Dynamic Text dialog box. The `imageFile` property is required.
3. The `allowDelete` property is optional. If this property is set to `false`, when the user clicks on this node in the Bindings panel, the minus (-) button is disabled. If set to `true`, the minus (-) button is enabled. If the property is not defined, the default is `true`.
4. The `dataSource` property is the simple name of the file in which the `findDynamicSources()` function is defined. For example, the `findDynamicSources()` function in Configuration/DataSources/ASP_Js/Session.htm sets the `dataSource` property to session.htm. The `dataSource` property is required.
5. The `name` property is the name of the server behavior that is associated with the data source, if one exists. Some data sources, such as recordsets, are associated with server behaviors. When you create a recordset and give it the name `rsAuthors`, the name property must equal `rsAuthors`. The `name` property is always defined, but can be an empty string (“”) if no server behavior is associated with the data source (such as a session variable).

Note: A JavaScript class that defines these properties exists in Configuration/Shared/Common/Scripts/DataSourceClass.js.

generateDynamicDataRef()

Availability

Dreamweaver UltraDev 1

Description

Generates the dynamic data object for a child node.

Arguments

`sourceName`, *`bindingName`*

- *`sourceName`* is the name of the top-level node that is associated with the child node.
- *`bindingName`* is the name of the child node from which you want to generate a dynamic data object.

Returns

Dreamweaver expects a string, which can be passed to `formatDynamicDataRef()` to format it before inserting it in a user’s document.

generateDynamicSourceBindings()

Availability

Dreamweaver UltraDev 1

Description

Returns the children of a top-level node.

Arguments

sourceName

sourceName is the name of the top-level node whose children you want to return.

Returns

Dreamweaver expects an array of JavaScript objects where each object can have as many as four properties:

1. The `title` property is the label string that appears on the right of the icon for each parent node. The `title` property is required.
2. The `allowDelete` property is an optional property. If this property is set to `false`, when the user clicks on this node in the Bindings panel, the minus (-) button is disabled. If this property is set to `true`, the minus (-) button is enabled. If the property is not defined, the default is `true`.
3. The `dataSource` property is the simple name of the file in which the `findDynamicSources()` function is defined. For example, the `findDynamicSources()` function in Configuration/DataSources/ASP_Js/Session.htm sets the `dataSource` property to session.htm. This is a required property.
4. The `name` property is the name of the server behavior that is associated with the data source, if one exists. It is a required property. Some data sources, such as recordsets, are associated with server behaviors. When you create a recordset and give it the name `rsAuthors`, the name property must equal `rsAuthors`. Other data sources, such as session variables, do not have a corresponding server behavior. Their name property must be the empty string (`""`).

Note: A JavaScript class that defines these properties exists in Configuration/Shared/Common/Scripts/DataSourceClass.js.

inspectDynamicDataRef()

Availability

Dreamweaver UltraDev 1

Description

From a dynamic data object, determines the corresponding node in the data source tree. The `inspectDynamicDataRef()` function compares the passed-in string to the string that `generateDynamicDataRef()` returns for each node in the tree. If a match is found, the `inspectDynamicDataRef()` function indicates which node in the tree matches the passed-in string. The function identifies the node by using an array that contains two elements. The first element is the parent name of the parent node, and the second element is the name of the child node. If no match is found, `inspectDynamicDataRef()` returns an empty array.

Each implementation of `inspectDynamicDataRef()` checks only for matches of its own object type. For example, the recordset implementation of `inspectDynamicDataRef()` finds a match only if the passed-in string matches a recordset node in the tree.

Arguments

`string`

`string` is the dynamic data object.

Returns

Dreamweaver expects an array of two elements (parent name and child name) for the matched node; `null` if no matches are found.

CHAPTER 16
Server Formats

"Data Sources" on page 191 discusses how Dreamweaver MX inserts dynamic data into the user's document by adding a server expression at the appropriate location. When a visitor requests the user's document from the web server, that server expression is converted to a value from a database, the contents of a request variable, or some other dynamic value. The Dreamweaver user can format how this dynamic value is presented to the visitor.

This chapter discusses the API that is used to format the dynamic data returned by the functions that are described in "Data Sources" on page 191. Functions that are described in both chapters work together to format dynamic data. If the user chooses a format for the dynamic data, Dreamweaver calls the Data Source function `generateDynamicDataRef()`, which is described in "Data Sources" on page 191, to get the string to be inserted into the user's document. Before inserting the string into the user's document, Dreamweaver passes that string to `formatDynamicDataRef()`, which is described in this chapter. The string that the `formatDynamicDataRef()` function returns is the formatted dynamic data that is finally inserted in the user's document.

The user can format dynamic data several ways. By using the Format menu in the Dynamic Data or Dynamic Text dialog box or the Bindings panel, the user can format the data before inserting it into an HTML document. If the user wants to create a format, he or she can select the Edit Format List command from the Format menu and select a format type from the plus (+) menu. The plus (+) menu contains a list of format types. Format types are basic format categories, such as Currency, DateTime, or AlphaCase. Format types collect all the common parameters for a category of format, letting you streamline the work to create a new format.

To illustrate, suppose you want to create a new currency format. Essentially, all currency formatting consists of converting a number to a string, inserting commas and decimal points, and inserting a currency symbol, such as a dollar ($) sign. The Currency format data type collects all the common parameters and prompts you for their values. When you create a new currency format, you're prompted for the required values.

Dreamweaver users can format data with built-in formats, create new formats that are based on built-in format types, or create new formats that are based on format types they created.

How data formatting works

All format files reside in the ServerFormats folder within the Configuration folder. Each server model has its own subfolder: ASP.Net_Csharp, ASP.Net_VB, ASP_Js, ASP_Vbs, ColdFusion, JSP, Shared, and UD4-ColdFusion. Each subfolder contains one XML file and multiple HTML files.

Formats.xml describes all the choices in the Format menu. None and Edit Format List are added automatically by Dreamweaver.

The folder also contains one HTML file for each currently installed format type. Format types include `AlphaCase`, `Currency`, `DateTime`, `Math`, `Number`, `Percent`, `Simple`, and `Trim`.

More about the Formats.xml file

The Formats.xml file contains one `<format>` tag for each item in the Format menu. Each `<format>` tag contains the following mandatory attributes:

- `file=`*`fileName`* is the HTML file for this format type, such as `"Currency"`.
- `title=`*`string`* is the string that appears in the Format menu, such as `"Currency - default"`.
- `expression=`*`regexp`* is a regular expression that matches the dynamic data objects that use this format. The expression is used to determine what format is currently applied to a dynamic data object. For example, the expression for the `"Currency - default"` format would be `"<%\s*=\s*FormatCurrency\(.*, -1, -2, -2, -2\)\s*%>|<%\s*=\s*DoCurrency\(.*, -1, -2, -2, -2\)\s*%>"`. The value of the expression attribute must be unique among all `<format>` tags in the file; it must be specific enough to guarantee that only instances of this format match the expression.
- `visibility=`*`[hidden | visible]`* indicates whether the value appears in the Format menu. If the value of `visibility` is `hidden`, the format does not appear in the Format menu.

The `<format>` tag can optionally contain additional, arbitrarily named attributes.

Some data formatting functions require an argument, *`format`*, which is a JavaScript object. This object is the node that corresponds to the `<format>` tag in the Formats.xml file. The object has a JavaScript property for each attribute of the corresponding `<format>` tag.

The following example shows the `<format>` tag for `"Currency - default"`:

```
<format file="Currency" title="Currency - default" ¬
expression="<%\s*=\s*FormatCurrency\(.*, -1, -2, -2, -2\)\s*%>|¬
<%\s*=\s*DoCurrency\(.*, -1, -2, -2, -2\)\s*%>"
NumDigitsAfterDecimal=-1 IncludeLeadingDigit=-2 ¬
UseParensForNegativeNumbers=-2 GroupDigits=-2/>
```

The format type for this format is `Currency`. The string `"Currency - default"` appears on the Format menu. The expression `<%\s*=\s*FormatCurrency\(.*, -1, -2, -2, -2\)\s*%>|<%\s*=\s*DoCurrency\(.*, -1, -2, -2, -2\)\s*%>` is used to find occurrences of this format in the user's document.

`NumDigitsAfterDecimal`, `IncludeLeadingDigit`, `UseParensForNegativeNumbers`, and `GroupDigits` are parameters for the `Currency` format type; they are not required. These parameters appear in the Parameters dialog box for the `Currency` format type. The Parameters dialog box appears when a user chooses the `Currency` format type from the plus (+) menu of the Edit Format List dialog box. The values that are specified for these parameters are used to define the new format.

The Edit Format List plus (+) menu

If you do not want a file in the ServerFormats folder to appear in the Edit Format List plus (+) menu, add the following statement as the first line in the HTML file:

```
<!-- MENU-LOCATION=NONE -->
```

To determine the contents of the menu, Dreamweaver first looks for a ServerFormats.xml file in the same folder as the data formats (for example, \Configuration\ServerFormats\ASP\ServerFormats.xml). The ServerFormats.xml file describes the contents of the Edit Format List plus (+) menu; it contains references to the HTML files that it lists in the menu.

Dreamweaver checks each referenced HTML file for a title tag. If the file contains a title tag, the content of the title tag appears in the menu. If the file does not contain a title tag, the filename is used in the menu.

After Dreamweaver finishes, or if this file does not exist, Dreamweaver scans the rest of the folder to find other items that should appear in the menu. If Dreamweaver find files in the main folder that aren't already in the menu, it adds them to the menu. If subfolders contain files that aren't already in the menu, Dreamweaver creates a submenu and adds those files to it.

When the data formatting functions are called

The data formatting functions are called in the following scenarios:

- In the Dynamic Data or Dynamic Text dialog box, the user chooses a node from the data source tree and a format from the Format menu. When the user selects the format, Dreamweaver calls `generateDynamicDataRef()` and passes the return value from `generateDynamicDataRef()` to `formatDynamicDataRef()`. The return value from `formatDynamicDataRef()` appears in the Code setting of the dialog box. After the user clicks OK, the string of code is inserted into the user's document. Next, Dreamweaver calls the `applyFormat()` function to insert a function declaration. See "generateDynamicDataRef()" on page 195 for more information. A similar process occurs when the user works with the Bindings panel.
- If the user changes the format or deletes the dynamic data item, the `deleteFormat()` function is called. The `deleteFormat()` function removes the support scripts from the document.
- When the user clicks the plus (+) button in the Edit Format List dialog box, Dreamweaver displays a menu that contains all the format types for the given server model. Each format type corresponds to a file in the Configuration\ServerFormats*currentServerModel* folder.

  If the user chooses a format from the plus (+) menu that requires a user-specified parameter, Dreamweaver executes the `onload` handler on the body tag and displays the Parameters dialog box, which shows the parameters for the format type. In this dialog box, the user chooses parameters for the format and clicks OK, and Dreamweaver calls the `applyFormatDefinition()` function.

  If the selected format does not need to display a dialog box and lets the user choose parameters, Dreamweaver calls `applyFormatDefinition()` when the user chooses the format type from the plus (+) menu.
- Later, if the user edits the format by selecting it in the Edit Format List dialog box and clicking the Edit button, Dreamweaver calls `inspectFormatDefinition()` before displaying the Parameters dialog box, so the form controls can be initialized to the correct values.

The Data Formatting API

applyFormat()

Availability

Dreamweaver UltraDev 1

Description

Adds a format function declaration to the user's document. When a user chooses a format from the Format field in the Dynamic Data or Dynamic Text dialog box or the Bindings panel, Dreamweaver makes two changes to the user's document: It adds the appropriate format function before the HTML tag (if it's not already there) and it changes the dynamic data object to call the appropriate format function.

Dreamweaver adds the function declaration by calling the `applyFormat()` JavaScript function in the data format file. It changes the dynamic data object by calling the `formatDynamicDataRef()` function.

The `applyFormat()` function should use the DOM to add function declarations to the top of the user's document. For example, if the user chooses Currency - Default, the function adds the declaration of the `Currency` function.

This function can edit a user's document.

Arguments

`format`

`format` is a JavaScript object that describes the format to be applied. The JavaScript object is the node that corresponds to the `<format>` tag in the Formats.xml file. The object has a JavaScript property for each attribute of the corresponding `<format>` tag.

Returns

Dreamweaver expects nothing.

applyFormatDefinition()

Availability

Dreamweaver UltraDev 1

Description

Commits the changes to a format created with the Edit Format dialog box.

Users can create, edit, or delete formats with the Edit Format List dialog box. This function is called to commit any modifications that are made. It can also set other, arbitrarily named properties on the object. Each property is stored as an attribute of the `<format>` tag in the Formats.xml file.

Arguments

`format`

`format` is a JavaScript object that corresponds to this format. The function must set the `expression` property of the JavaScript object to be the regular expression for the format. The function can also set other, arbitrarily named properties of the object. Each property is stored as an attribute of the `<format>` tag.

Returns

Dreamweaver expects the format object, if the function completes successfully. If an error occurs, the function returns an error string. If it returns an empty string, the form is closed, but the new format is not created, which is the same as a Cancel operation.

deleteFormat()

Availability

Dreamweaver UltraDev 1

Description

Removes the format function declaration from the top of the user's document.

When the user changes the format of a dynamic data object (in the Dynamic Data or Dynamic Text dialog box or the Bindings panel) or deletes a formatted dynamic data object, Dreamweaver removes the function declaration from the top of the document and removes the function call from the dynamic data object by calling the `deleteFormat()` function.

The `deleteFormat()` function should use the DOM to remove the function declaration from the top of the current document.

Arguments

`format`

`format` is a JavaScript object that describes the format to be removed. The JavaScript object is the node that corresponds to the `<format>` tag in the Formats.xml file.

Returns

Dreamweaver expects nothing.

formatDynamicDataRef()

Availability

Dreamweaver UltraDev 1

Description

Adds the format function call to the dynamic data object. When a user chooses a format from the Format field in the Dynamic Data or Dynamic Text dialog box or the Bindings panel, Dreamweaver makes two changes to the user's document: It adds the appropriate format function before the HTML tag (if it's not already there), and it changes the dynamic data object to call the appropriate format function.

Dreamweaver adds the function declaration by calling the `applyFormat()` JavaScript function in the data format file. It changes the dynamic data object by calling the `formatDynamicDataRef()` function.

The `formatDynamicDataRef()` function is called when the user selects a format from the Format field in the Bindings panel or the Dynamic Data or Dynamic Text dialog box. It does not edit the user's document.

Arguments

dynamicDataObject, *format*

- *dynamicDataObject* is a string that contains the dynamic data object.
- *format* is a JavaScript object that describes the format to be applied. The JavaScript object is the node that corresponds to the <format> tag in the Formats.xml file. The object has a JavaScript property for each attribute of the corresponding <format> tag.

Returns

Dreamweaver expects the new value for the dynamic data object.

If an error occurs, the function displays an alert message under certain conditions. If the function returns an empty string, the Format field is set to None.

inspectFormatDefinition()

Availability

Dreamweaver UltraDev 1

Description

Initializes form controls when a user tries to edit a format in the Edit Format List dialog box.

Arguments

format

format is a JavaScript object that describes the format to be applied. The JavaScript object is the node that corresponds to the <format> tag in the Formats.xml file. The object has a JavaScript property for each attribute of the corresponding <format> tag.

Returns

Dreamweaver expects nothing.

CHAPTER 17
Components

Components are modularized groups of functions, or objects, that can be used as building blocks for applications and web pages. Some component types adhere to established sets of protocols, letting developers connect components that adhere to the same protocols. Each component has a reflection API that contains metadata that describes the component functionality to the systems upon which it is loaded. Component types usually run only on specific server models that support the specifications for handling them. Web Services, ColdFusion Components, and JavaBeans are good examples of component architecture.

Each component contains generic methods through which it informs the system upon which it is loaded about the functionality that it supports (in other words, meta discovery that uses a reflection API). Adherence to component architecture lets objects be loaded dynamically.

Dreamweaver's Component panel lets users load and work with components. It lists all the available component types that are compatible with each enabled server model. For instance, because JavaBeans can work only on a JSP page, JavaBeans components appear only in the JSP server model within the Component panel. Likewise, because CFCs can work only on a ColdFusion page, they appear only in ColdFusion within the Component panel.

Extensibility lets you add new component types into the panel. After you add the new components, they appear in the Components pop-up list. You can also add instructions for setting up components that appear in the Component panel or in a dialog box (depending on the extension for which the steps are implemented) as numbered steps. The Setup Steps then display interactively as users load the new components, with checkmarks appearing next to any step that is already completed.

Component panel files

Component files are stored in the Configuration/Components/*server-model*/*ServerType* folder.

Creating a custom component that can work in the Component panel involves the following procedures:

- Preparing the files to display the component in the user interface.
- Enabling the component in JavaScript.

If you want the component type to display in tree control view, you also need to create the associated optional files and populate the tree control.

You can set a component type to work at the level of an individual web page, to a set of web pages, or to an entire site. Your JavaScript code must include the logic for component persistence—for saving itself between sessions and reloading at the start of a new session. For example, JavaBeans should contain the logic for saving themselves in the multiuser configuration directory as JavaBeansList.xml.

```
<javabeans>
<javabean classname="TestCollection.MusicCollection"
classlocation="d:\music\music.jar"></javabean>
</javabeans>
```

Note: The Configuration/Components folder has a subfolder for each implemented server model. Components are filtered, based on their server model. You can refer to the existing server models and server behaviors when creating a new component type (for more information on server models, see "Server Models" on page 217; for more information on server behaviors, see "Server Behaviors" on page 145).

Adding a service component

To add a new lightweight directory access protocol (LDAP) service using Dreamweaver MX:

1 Using existing component type files as a model, create all the required files, plus the optional files that you want, to display the new component type, as shown in the following table:

Filename	Description	Required/Optional
LDAP.htm	The extension file	Required
LDAP.js	The extension file that implements the Component API callback	Required
LDAP.gif	The image that appears in the Components pop-up list	Required
LDAPMenus.xml	The repository for metadata that organizes the Components panel structure	Optional
LDAP*.gif	Toolbar images, which can be enabled or disabled, as shown in the following example: `ToolBarImageUp.gif` `ToolBarImageDown.gif` `ToolBarImageDisabled.gif`.	Optional
LDAP*.gif	Tree node images	Optional

Note: Keep the same prefix throughout all the files that correspond to one component so that each file and its corresponding component can easily be identified.

2 Write the JavaScript code to implement the new server component. For details of the Component API functions that are available, see "Component panel API functions" on page 207.

Tip: When adding a new service, you might want to use the Components panel to browse meta information so that the information is readily available as you create the extension. Dreamweaver can browse added components and display nodes in the component tree. It provides drag-and-drop support and keyboard support in Code view.

Populating the tree control

Use the `ComponentRec` property to populate a Component panel tree control, so that it appears within the Component panel in the proper location.

Every node in a tree control must have the following properties:

Property name	Description	Required/Optional
`name`	Name of the tree node item	Required
`image`	Icon of the tree node item. If not specified a default icon is used.	Optional
`hasChildren`	Responds to clicks on the plus (+) and minus (-) buttons in the tree control by loading children.This lets you work with a tree that is not prepopulated.	Required
`toolTipText`	Tooltip text of the tree node item	Optional
`isCodeViewDraggable`	Determines whether the item can be dragged and dropped into the code view.	Optional
`isDesignViewDraggable`	Determines whether the item can be dragged and dropped into the design view.	Optional

Component panel API functions

displayInstructions()

Availability

Dreamweaver MX

Description

If there are no component instances (in other words the tree is empty), it displays instructions to add a new one.

Arguments

None.

Returns

Dreamweaver expects a string.

Example

```
function displayInstructions()
{
       return MM.MSG_WebServicesInstructions;
}
```

displayHelp()

Availability

Dreamweaver MX

Description

Displays help text for the current tree node item using the Help System using the help document.

Arguments

componentRec

componentRec is an object.

Returns

Dreamweaver expects nothing.

Example

```
function displayHelp(componentRec)
{
  displayHelp();
}
```

getComponentChildren()

Availability

Dreamweaver MX

Description

Returns a list of child `ComponentRec` objects for the active parent `ComonentRec`. To load the root-level tree items, this function needs to read its metadata from its persistent store.

Arguments

{*parentComponentRec*}

parentComponentRec is the `componentRec` of the parent. If it is omitted, Dreamweaver expects a list of `ComponentRec` objects for the root node.

Returns

Dreamweaver expects an array of `ComponentRec` objects.

Example

```
function getComponentChildren(componentRec)
{
  var cs_Children = new Array();

  if (!componentRec)
  {
    //read saved entries for java beans.
    var javabeanListPath = dw.getSiteRoot() + JavaBeanListFile;
    if (DWfile.exists(javabeanListPath))
    {
    }
  }
  else
  {
    if (componentRec.objectType == "Class")
    {
      var propertiesCompInfo = new ComponentRec("Properties",
  PROPERTIES_FILENAME, true,true,"Properties","DWJavaBeansContextProperty");
      propertiesCompInfo.objectType = "Properties";

      var methodsCompInfo = new ComponentRec("Methods", METHODS_FILENAME,
  true,true,"Methods","DWJavaBeansContextMethod");
      methodsCompInfo.objectType = "Methods";

      cs_Children.push(propertiesCompInfo);
      cs_Children.push(methodsCompInfo);
    }
    else if (componentRec.objectType == "Properties")
    {
       var Properties =
  MMJB.getProperties(componentRec.parent.getName(),componentRec.parent.classl
  ocation);
      if (Properties.length)
      {
       for (var j = 0;j < Properties.length; j++)
       {
         var propertiesCompInfo = new ComponentRec(Properties[j],
  PROPERTIES_FILENAME, true,false,Properties[j]);
         propertiesCompInfo.objectType = "Property";
         cs_Children.push(propertiesCompInfo);
       }
      }
    }
  }

  return cs_Children;
}
```

getContextMenuId()

Availability

Dreamweaver MX

Description:

An optional function that gets the Context Menu ID for the component type. Every component type can have a context menu associated with it. The Context Menu pop-up menus are defined in ComponentNameMenus.xml, and they work the same way as menu.xml. The menu string can be static or dynamic. Shortcut keys (accelerator keys) are supported.

Arguments

None.

Returns

Dreamweaver expects a string.

Example

```
function getContextMenuId()
{
  return "DWConnectionsContext";
}

<shortcutlist app="ultradev" id="DWConnectionsContext">
  <shortcut key="Cmd+I"   domRequired="false"command="clickedInsert();"
  id="DWShortcuts_ServerComponent_Insert" />
  <shortcut key="Del"domRequired="false"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Connection’))"
  command="clickedDelete();"  id="DWShortcuts_ServerComponent_Delete" />
</shortcutlist>

<menubar name="" app="ultradev" id="DWConnectionsContext">
    <menu name="" id="DWContext_Connections">
        <menuitem name="_Edit Connection..." key="Cmd+I"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Connection’))"
  command="clickedEdit();" id="DWContext_Connections_TestConnection" />
        <menuitem name="Du_plicate Connection..." key="Cmd+P"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Connection’))"
  command="clickedDuplicate();" id="DWContext_Connections_TestConnection" />
        <menuitem name="_Delete Connection..." key="Cmd+D"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Connection’))"
  command="clickedDelete();" id="DWContext_Connections_TestConnection" />
        <menuitem name="_Test Connection..." key="Cmd+T"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Connection’))"
  command="clickedTest();" id="DWContext_Connections_TestConnection" />
    <separator/>
      <menuitem name="New Recordset..."
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Table’))" key="Cmd+Q"
  command="clickedRecordset();" id="DWContext_Connections_TestConnection" />
      <menuitem name="View _Data..." key="Cmd+D"command="clickedViewData();"
  enabled="(dw.serverComponents.getSelectedNode() != null &&
  (dw.serverComponents.getSelectedNode().objectType==’Table’))"
  id="DWContext_Tables_ViewData" />
    <separator/>
      <menuitem name="_Insert" key="Cmd+I"   domRequired="false
  "command="clickedInsert();" id="DWShortcuts_ServerComponent_Insert" />
     <menuitem name="_Refresh" key="Cmd+R"
  command="dw.serverComponents.refresh()"
  id="DWContext_Connections_TestConnection" />
    </menu>
</menubar>

<menubar name="" app="ultradev" id="DWConnectionsChoosersContext">
    <menu name="" id="DWContext_ConnectionsChooser">
     <menuitem dynamic name="Choosers"app="ultradev" file="Menus/MM/
  DB_Connections.htm" id="DWContext_Connections_Chooser_List" />
    </menu>
</menubar>
```

getCodeViewDropCode()

Availability

Dreamweaver MX

Description

Gets the code that is dragged and dropped in Code view from the Component panel or the code that is cut, copied, or pasted from the Component panel.

Arguments

Dreamweaver expects `componentRec`.

Returns

Dreamweaver expects a string.

Example

```
function getCodeViewDropCode(componentRec)
{
  var codeToDrop="";
  if (componentRec)
    codeToDrop =  componentRec.name;
  return codeToDrop;
}
```

getSetupSteps()

Availability

Dreamweaver MX

Description

Dreamweaver calls this function if `setupStepsCompleted()` returns zero or a positive integer. Controls the function of server-side setup instructions, which can be implemented using extensions that use a modal dialog box and extensions that use server components.

Returns an array string for Dreamweaver to display in either the Setup Steps dialog box or the Components panel, depending on the extension type.

Arguments

None.

Returns

Dreamweaver expects an array of *n*+1 strings, where *n* is the number of steps, as described in the following list:

- The title that appears above the list of setup steps.
- For each step, the text instructions, which can include any HTML markup that is legal inside a `<li>` tag.

You can include hypertext links (`<a>` tags) in the list of steps by using the following form:

```
<a ref="#" onMouseDown="handler">Blue Underlined Text</a>
```

"handler" can be replaced by any of the following strings:

- Any JavaScript expression, such as "`dw.browseDocument('http://www.macromedia.com')`".

- `"Event:SetCurSite"` pops up a dialog box to set the current site.
- `"Event:CreateSite"` pops up a dialog box to create a new site.
- `"Event:SetDocType"` pops up a dialog box to change the document type of the user's document.
- `"Event:CreateConnection"` pops up a dialog box to create a new database connection.
- `"Event:SetRDSPassword"` pops up a dialog box to set the Remote Development Service (RDS) user name and password (ColdFusion only).
- `"Event:CreateCFDataSource"` pops up the ColdFusion administrator in a browser.

setupStepsCompleted()

Availability

Dreamweaver MX

Description

Dreamweaver calls this function before the Components tab becomes visible. Dreamweaver then calls `getSetupSteps()` if this function returns zero or a positive integer.

Arguments

None.

Returns

Dreamweaver expects an integer that represents the number of setup steps the user has already completed, as described in the following list:

- A value of either zero or a positive integer indicates the number of steps already completed.
- A value of `-1` indicates that all the necessary setup steps have been completed, so the instruction list does not appear.

handleDoubleClick()

Availability

Dreamweaver MX

Description

When the user double clicks the node in the tree, the event handler is called to allow editing. This function is optional. The function can return `false`, which indicates that the event handler is not handled. In that event, double clicking causes the default behavior, which is expanding or collapsing the tree nodes.

Arguments

`componentRec`

`componentRec` is an object that contains the following properties:

- `name` Name of the tree node item.
- `image` Optional icon for the tree node item. If omitted, Dreamweaver uses a default icon.
- `hasChildren` A Boolean value that indicates whether the tree node item is expandable. If `true`, Dreamweaver displays the plus (+) and minus (-) buttons for the tree node item.

- `toolTipText` Optional tool tip text for the tree node item.
- `isCodeViewDraggable` A Boolean value that indicates whether the tree node item can be dragged and dropped into the code view.
- `isDesignViewDraggable` A Boolean value that indicates whether the tree node item can be dragged and dropped into the design view.

Returns

Dreamweaver expects nothing.

Example

```
function handleDoubleClick(componentRec)
{
  if (componentRec &&
  ((componentRec.objectType=="Table")||
  (componentRec.objectType=="View")))
  {
  var objname = componentRec.name;
  var connname = componentRec.parent.parent.parent.name;
  var sqlstatement = "Select * from " + objname;
  MMDB.showResultset(connname,sqlstatement);
  return true;
  }
  return false;
}
```

toolbarControls()

Availability

Dreamweaver MX

Description

Every component type returns a list of `toolBarButtonRec` objects, which represents the toolbar icons, in left to right order. Each `toolBarButtonRec` object contains these properties:

Property Name	Description
`image`	Path to image file
`disabledImage`	Optional; path to disabled image looks for the toolbar button
`pressedImage`	Optional; path to pressed image looks for the toolbar button
`toolTipText`	Tooltip for the toolbar button
`toolStyle`	Left /right
`enabled`	JavaScript code that returns a Boolean value (`true` or `false`). The enablers are called when the following conditions exist: - When `dreamweaver.serverComponents.refresh()` is called - When the selection in the tree changes - When server model changes
`command`	The JavaScript code to execute. The command handler can force a refresh using `dreamweaver.serverComponents.refresh()`.
`menuId`	The menu ID for pop-up menu button when a button is clicked. When this ID is present, it overrides the command handler. Ideally, they should be mutually exclusive.

Arguments

None.

Returns

Dreamweaver expects an array of toolbar buttons in left-to-right order.

Example

```
function toolbarControls()
{
  var toolBarBtnArray = new Array();

  var plusButton = new ToolbarControlRec();
  plusButton.image= PLUSDROPBUTTONUP;
  plusButton.pressedImage= PLUSDROPBUTTONDOWN;
  plusButton.disabledImage= "DWWebServicesChoosersContext";
  plusButton.toolStyle= "left";
  plusButton.toolTipText= MM.MSG_WebServicesAddToolTipText;
  plusButton.menuId = "DWWebServicesChoosersContext";
  toolBarBtnArray.push(plusButton);

  var minusButton = new ToolbarControlRec();
  minusButton.image= MINUSBUTTONUP;
  minusButton.pressedImage= MINUSBUTTONDOWN;
  minusButton.disabledImage= MINUSBUTTONDISABLED;
  minusButton.toolStyle= "left";
  minusButton.toolTipText= MM.MSG_WebServicesDeleteToolTipText;
  minusButton.command = "clickedDelete()";
  minusButton.enabled = "(dw.serverComponents.getSelectedNode() != null &&
  dw.serverComponents.getSelectedNode() &&
  (dw.serverComponents.getSelectedNode().objectType=='Root'))";
  toolBarBtnArray.push(minusButton);

  var editWServiceButton = new ToolbarControlRec();
  editWServiceButton.image= EDITWSERVICEBUTTONUP;
  editWServiceButton.pressedImage= EDITWSERVICEBUTTONDOWN;
  editWServiceButton.disabledImage= EDITWSERVICEBUTTONDISABLED;
  editWServiceButton.toolStyle= "right";
  editWServiceButton.toolTipText= MM.MSG_WebServicesEditToolTipText ;
  editWServiceButton.command = "editWebService()";
  editWServiceButton.enabled = "(dw.serverComponents.getSelectedNode() != null
  && dw.serverComponents.getSelectedNode() &&
  (dw.serverComponents.getSelectedNode().objectType=='Root'))";
  toolBarBtnArray.push(editWServiceButton);

  var proxyButton = new ToolbarControlRec();
  proxyButton.image= PROXYBUTTONUP;
  proxyButton.pressedImage= PROXYBUTTONDOWN;
  proxyButton.disabledImage= PROXYBUTTONDISABLED;
  proxyButton.toolStyle= "right";
  proxyButton.toolTipText= MM.MSG_WebServicesRegenToolTipText;
  proxyButton.command = "reGenerateProxy()";
  proxyButton.enabled = "(dw.serverComponents.getSelectedNode() != null &&
  dw.serverComponents.getSelectedNode() &&
  (dw.serverComponents.getSelectedNode().objectType=='Root'))";
  toolBarBtnArray.push(proxyButton);

  return toolBarBtnArray;

}
```

CHAPTER 18
Server Models

Server models are the technologies that run scripts on a server. When users define a new site, they can identify the server model that they want to use at the site level and at the individual document level. This server model is used to handle any dynamic elements that the user adds to the document.

Server model configuration files are stored in the Configuration/ServerModels folder. Within that folder, each server model has its own HTML file that implements a set of functions that are required by the server model.

The Server Model API

You can customize some features of a server model using the functions that are available in the Server Model API.

Dreamweaver MX asks new users to identify server models when they first start Dreamweaver. For cases when the user does not identify a server model, you can create a dynamic dialog box that prompts the user to complete the necessary steps. This dialog box appears when the user attempts to insert a server object. For information on creating such a dialog box, refer to the functions "getSetupSteps()" on page 212 and "setupStepsCompleted()" on page 213.

You might want to create a specialized server model. Macromedia suggests that you create a new server model rather than editing any of the ones that come with Dreamweaver MX. (For information regarding creating new document types that are supported by your server model, refer to "Extensible document types in Dreamweaver" on page 22.)

When creating a new server model, you need to include an implementation of the `canRecognizeDocument()` function in your server model file. This function tells Dreamweaver the level of preference that it should give to your server model for handling that file extension when multiple server models claim a particular file extension.

canRecognizeDocument()

Availability

Dreamweaver MX

Description

When opening a document (and when more than one server model claims a file extension), Dreamweaver MX calls this function for each of the extension-associated server models to see whether any of the functions can identify whether the document is their file. If more than one server model claims the file extension, Dreamweaver gives priority to the server model that returns the highest integer.

Note: All Dreamweaver MX-defined server models return a value of `1` so third-party server models can override the file-extension association.

Arguments

dom

dom is the Macromedia document object, which is returned by the function `dreamweaver.getDocumentDOM()`.

Returns

Dreamweaver expects an integer that indicates the priority that the developer gives to the server model for the file extension. This function should return a value of `-1` if the server model does not claim the file extension; otherwise, this function should return a value greater than zero.

Example

In the following example, if the user opens a JavaScript document for the current server model, the sample code returns a value of `2`. This value lets the developer's server model take precedence over that of Macromedia.

```
var retVal = -1;
var langRE = /@\s*language\s*=\s*(\"|\')?javascript(\"|\')?/i;
// Search for the string language="javascript"
var oHTML = dom.documentElement.outerHTML;
if (oHTML.search(langRE) > -1)
   retVal = 2;
return retVal;
```

getFileExtensions()

Availability

Dreamweaver UltraDev 1, deprecated in Dreamweaver MX

Description

Returns the document file extensions with which a server model can work. For example, the ASP server model supports .asp and .htm file extensions. This function returns an array of strings, and Dreamweaver uses these strings to populate the Default Page Extension list that is found in the App Server category of the Site Definition dialog box.

Note: The Default Page Extension list exists only in Dreamweaver 4 and earlier. For Dreamweaver MX, the Site Definition dialog box does not list file extension settings. Instead, Dreamweaver MX reads the Extensions.txt file and parses the `<documenttype>` element in the mmDocumentTypes.xml file. (For more information on these two files and the `<documenttype>` element, see "Extensible document types in Dreamweaver" on page 22.)

Arguments

None.

Returns

Dreamweaver expects an array of strings that represent the allowed file extensions.

getLanguageSignatures()

Availability

Dreamweaver MX

Description

Returns an object that describes the method and array signatures that the scripting language uses. The `getLanguageSignatures()` function helps the developer map generic signature mapping to language-specific mapping for the following elements:

- The function
- Constructors
- Drop code (return values)
- Arrays
- Exceptions
- Data type mappings for primitive data types

The `getLanguageSignatures()` function returns a map of these signature declarations. Extension developers can use this map to generate language-specific code blocks that Dreamweaver drops on the page (based on the appropriate server model for the page) when the user drags and drops, for example, a Web Services method.

For examples of how to write this function, see the HTML implementation files for the JSP and the ASP.Net server models. Server model implementation files are located in the Configuration/ServerModels folder.

Arguments

None.

Returns

Dreamweaver expects an object that defines the scripting language signatures. This object should map the generic signatures to language-specific ones.

getServerExtension()

Availability

Dreamweaver UltraDev 4, deprecated in Dreamweaver MX

Description

Returns the default file extension of files that use the current server model. The `serverModel` object is set to the server model of the currently selected site if no user document is currently selected.

Arguments

None.

Returns

Dreamweaver expects a string that represents the supported file extensions.

getServerInfo()

Availability

Dreamweaver MX

Description

Returns a JavaScript object, which can be accessed from within the JavaScript code. You can retrieve this object by calling the `dom.serverModel.getServerInfo()` JavaScript function. Furthermore, `serverName`, `serverLanguage`, and `serverVersion` are special properties, which you can access through these JavaScript functions:

```
dom.serverModel.getServerName()
dom.serverModel.getServerLanguage()
dom.serverModel.getServerVersion()
```

Arguments

None.

Returns

Dreamweaver expects an object that contains the properties of your server model.

Example

```
var obj = new Object();
obj.serverName = "ASP";
obj.serverLanguage = "JavaScript";
obj.serverVersion = "2.0";
...
return obj;
```

getServerLanguages()

Availability

Dreamweaver UltraDev 1, deprecated in Dreamweaver MX

Description

Returns the supported scripting languages of a server model. This function returns an array of strings. Dreamweaver uses these strings to populate the Default Scripting Language list that is found in the App Server category of the Site Definition dialog box.

Note: The Default Scripting Language list exists only in Dreamweaver 4 and earlier. For Dreamweaver MX, the Site Definition dialog box does not list supported scripting languages nor does Dreamweaver MX use the `getServerLanguages()` function. Dreamweaver MX does not use this function because each server model has only one server language in Dreamweaver MX.

In versions of Dreamweaver other than MX, a server model can support multiple scripting languages. For example, the ASP server model supports JavaScript and VBScript.

Note: If you want a file in the ServerFormats folder to apply only to a specific scripting language, add the following statement so it is the first line in the HTML file:

```
<!-- SCRIPTING-LANGUAGE=XXX -->
```

In this example, `XXX` represents the scripting language. This statement causes the server behavior to appear in the plus (+) menu of the Server Behaviors panel only when the currently selected scripting language is `XXX`.

Arguments

None.

Returns

Dreamweaver expects an array of strings that represent the supported scripting languages.

getServerModelExtDataNameUD4()

Availability

Dreamweaver MX

Description

Returns the server model implementation name that Dreamweaver should use when accessing UltraDev 4 extension data files that reside in the Configurations/ExtensionData folder.

Arguments

None.

Returns

Dreamweaver expects a string, such as `"ASP/JavaScript"`.

getServerModelDelimiters()

Availability

Dreamweaver MX

Description

Returns the script delimiters that are used by the application server and indicates whether each can participate in merging code blocks. You can access this returned value from JavaScript by calling the `dom.serverModel.getDelimiters()` function.

Arguments

None.

Returns

Dreamweaver expects an array of objects where each object contains the following three properties:

- *`startPattern`* is a regular expression that matches the opening script delimiter (such as `"<%"`).
- *`endPattern`* is a regular expression that matches the closing script delimiter (such as `"%>"`).
- *`participateInMerge`* is a Boolean value that specifies whether the content enclosed in the listed delimiters should (`true`) or should not (`false`) participate in block merging.

getServerModelDisplayName()

Availability

Dreamweaver MX

Description

Returns the name that should appear in the user interface for this server model. You can access this value from JavaScript by calling the `dom.serverModel.getDisplayName()` function.

Arguments

None.

Returns

Dreamweaver expects a string, such as `"ASP JavaScript"`.

getServerModelFolderName()

Availability

Dreamweaver MX

Description

Returns the folder name to be used for this server model within the Configuration folder. You can access this value from JavaScript by calling the `dom.serverModel.getFolderName()` function.

Arguments

None.

Returns

Dreamweaver expects a string, such as `"ASP_JS"`.

getServerSupportsCharset()

Availability

Dreamweaver MX

Description

Returns `true` if the current server supports the given character set. From JavaScript, you can determine whether the server model supports a particular character set by calling the `dom.serverModel.getServerSupportsCharset()` function.

Arguments

`metaCharSetString`

`metaCharSetString` is a string that holds the value of the documents `"charset="` attribute.

Returns

Dreamweaver expects a Boolean value.

getVersionArray()

Availability

Dreamweaver UltraDev 1, deprecated in Dreamweaver MX

Description

Provides a mapping of server technologies to version numbers. This function is called by `dom.serverModel.getServerVersion()`.

Arguments

None.

Returns

Dreamweaver expects an array of version objects, each with a version name and version value, as listed in the following examples:

- ASP version 2.0
- ADODB version 2.1

CHAPTER 19
Data Translators

Data translators translate specialized markup—server-side includes, conditional JavaScript statements, or other code such as PHP3, JSP, CFML, or ASP—into code that can be read and displayed by Dreamweaver. In Dreamweaver, you can translate attributes within tags as well as entire tags or blocks of code. All data translators—block/tag or attribute—are HTML files.

Translated tags or blocks of code must be enclosed in locked regions to preserve the original markup. Translated attributes do not require locks, which makes inspecting the tags that contain them a simple process.

Data translation—especially for entire tags or blocks of code—might involve complex operations that either cannot be done with JavaScript or that can be done more efficiently using C. If you are familiar with C or C++, you should also read "C-Level Extensibility" on page 251.

How data translators work

Dreamweaver handles all translator files the same way, regardless of whether they translate entire tags or only attributes. At startup, Dreamweaver reads all the files in the Configuration/Translators folder and calls the `getTranslatorInfo()` function to obtain information about the translator. Dreamweaver ignores any file in which `getTranslatorInfo()` does not exist or contains an error that causes it to be undefined.

Note: To prevent JavaScript errors from interfering with startup, errors in any translator file are reported only after all translators are loaded. For more information on debugging translators, see "Finding bugs in your translator" on page 241.

Dreamweaver also calls the `translateMarkup()` function in all applicable translator files (as specified in the Translation preferences) whenever the user might have added new or changed existing content that needs translation. Dreamweaver calls `translateMarkup()` when the user performs one of the following actions:

- Opens a file in Dreamweaver
- Switches back to Design view after making changes in the HTML panel or in Code view
- Changes the properties of an object in the current document
- Inserts an object (using either the Objects panel or the Insert menu)
- Refreshes the current document after making changes to it in another application
- Applies a template to the document
- Pastes or drags content into or within the Document window
- Saves changes to a dependent file

- Invokes a command, behavior, server behavior, Property inspector, or other extension that sets the `innerHTML` or `outerHTML` property of any tag object or the `data` property of any comment object
- Selects File > Convert > 3.0 Browser Compatible
- Selects Modify > Convert > Convert Tables to Layers
- Selects Modify > Convert > Convert Layers to Tables
- Changes a tag or attribute in the Quick tag editor and presses Tab or Enter

getTranslatorInfo()

Description

Provides information about the translator and the files it can affect.

Arguments

None.

Returns

An array of strings. The elements of the array must appear in the following order:

1. *`translatorClass`* uniquely identifies the translator. This string must begin with a letter and can contain only alphanumeric characters, hyphens (-), and underscores (_).
2. *`title`* describes the translator in no more than 40 characters.
3. *`nExtensions`* specifies the number of file extensions to follow. If *`nExtensions`* is zero, the translator can run on any file. If *`nExtensions`* is zero, *`nRegExps`* is the next element in the array.
4. *`extension`* specifies a file extension (for example, `"htm"` or `"SHTML"`) that works with this translator. This string is case-insensitive and should not contain a leading period. The array should contain the same number of *`extension`* elements as are specified in *`nExtensions`*.
5. *`nRegExps`* specifies the number of regular expressions that follow. If *`nRegExps`* is zero, *`runDefault`* is the next element in the array.
6. *`regExps`* specifies a regular expression that you can check. The array should contain the same number of *`regExps`* elements as are specified in *`nRegExps`*, and at least one of the *`regExps`* must match a piece of the document's source code before the translator can act on a file.

7 *runDefault* specifies when this translator executes. The following table lists the possible values:

Value	Description
"allFiles"	Sets the translator to always execute
"noFiles"	Sets the translator to never execute
"byExtension"	Sets the translator to execute for files that have one of the file extensions that are specified in the extension
"byExpression"	Sets the translator to execute if the document contains a match for one of the specified regular expressions
"bystring"	Sets the translator to execute if the document contains a match for one of the specified strings

If you set *runDefault* to "byExtension" but do not specify any extensions (see step 4), the effect is the same as setting "allFiles". If you set *runDefault* to "byExpression" but do not specify any expressions (see *regExps*, above), the effect is the same as setting "noFiles".

- *priority* specifies the default priority for running this translator. The priority is a number between 0 and 100. If you do not specify a priority, the default priority is 100. The highest priority is 0 and 100 is lowest. When multiple translators apply to a document, this setting controls the order in which the translators are applied. The highest priority is applied first. When multiple translators have the same priority, they are applied in alphabetical order by translatorClass.

Example

The following instance of getTranslatorInfo() gives information about a translator for server-side includes:

```
function getTranslatorInfo(){
  var transArray = new Array(11);

  transArray[0] = "SSI";
  transArray[1] = "Server-Side Includes";
  transArray[2] = "4";
  transArray[3] = "htm";
  transArray[4] = "stm";
  transArray[5] = "html";
  transArray[6] = "shtml";
  transArray[7] = "2";
  transArray[8] = "<!--#include file";
  transArray[9] = "<!--#include virtual";
  transArray[10] = "byExtension";
  transArray[11] = "50";

  return transArray;
}
```

translateMarkup()

Description

Performs the translation.

Arguments

docName, *siteRoot*, *docContent*

- *docName* is a string that contains the file:// URL for the document to be translated.
- *siteRoot* is a string that contains the file:// URL for the root of the site that contains the document to be translated. If the document is outside a site, this string might be empty.
- *docContent* is a string that contains the contents of the document.

Returns

A string that contains the translated document or an empty string if nothing is translated.

Example

The following instance of `translateMarkup()` calls the C function `translateASP()`, which is contained in a DLL (Windows) or a code library (Macintosh) called ASPTrans:

```
function translateMarkup(docName, siteRoot, docContent){
  var translatedString = "";
  if (docContent.length > 0){
  translatedString = ASPTrans.translateASP(docName, siteRoot, ¬
  docContent);
  }
  return translatedString;
}
```

For an all-JavaScript example, see "A simple attribute translator example" on page 230 or "A simple block/tag translator example" on page 235.

liveDataTranslateMarkup function()

Availability

Dreamweaver UltraDev 1

Description

Translates documents when users are in the Live Data window. When the user chooses the View > Live Data menu item or clicks the Refresh button, Dreamweaver calls the `liveDataTranslateMarkup()` function instead of the `translateMarkup()` function.

Arguments

docName, *siteRoot*, *docContent*

- *docName* is a string that contains the file:// URL for the document to be translated.
- *siteRoot* is a string that contains the file:// URL for the root of the site that contains the document to be translated. If the document is outside a site, this string might be empty.
- *docContent* is a string that contains the contents of the document.

Returns

A string that contains the translated document or an empty string if nothing is translated.

Example

The following instance of `liveDataTranslateMarkup()` calls the C function `translateASP()`, which is contained in a DLL (Windows) or a code library (Macintosh) called ASPTrans:

```
function liveDataTranslateMarkup(docName, siteRoot, docContent){
  var translatedString = "";
  if (docContent.length > 0){
  translatedString = ASPTrans.translateASP(docName, siteRoot, docContent);
  }
  return translatedString;
}
```

Determining what kind of translator to use

All translators are the same to a certain extent: They must contain the `getTranslatorInfo()` and `translateMarkup()` functions, and they must reside in the Configuration/Translators folder. They differ, however, in the kind of code that they insert into the user's document and in how that code must be inspected.

- To translate small pieces of server markup that determine attribute values or that conditionally add attributes to a standard HTML tag, write an attribute translator. Standard HTML tags that contain translated attributes can be inspected with the Property inspectors that are built into Dreamweaver. It is not necessary to write a custom Property inspector (see "Adding a translated attribute to a tag" on page 229).
- To translate an entire tag (for example, a server-side include) or a block of code (for example, JavaScript, ColdFusion, PHP, or other scripting), write a block/tag translator. The code that is generated by a block/tag translator cannot be inspected with the Property inspectors that are built into Dreamweaver. You must write a custom Property inspector for the translated content if you want users to be able to change the properties of the original code (see "Locking translated tags or blocks of code" on page 234).

Adding a translated attribute to a tag

Attribute translation relies heavily on the ability of the Dreamweaver parser to ignore server markup. Dreamweaver already ignores the most common kinds of server markup (including ASP, CFML, and PHP) by default; if you use server markup that has different start and end markers, you must modify the third-party tag database to ensure that your translator works properly. For more information on modifying the third-party tag database, see "Customizing Dreamweaver" in *Using Dreamweaver*.

When Dreamweaver handles the preservation of the original server markup, the translator generates a valid attribute value that can be viewed in the Document window. (If you use server markup only for attributes that do not have a user-visible effect, you do not need a translator.)

The translator creates an attribute value that has a visible effect in the Document window by adding a special attribute, `mmTranslatedValue`, to the tag that contains the server markup. The `mmTranslatedValue` attribute and its value are not visible in the HTML panel or in Code view, nor are they saved with the document.

The mmTranslatedValue attribute must be unique within the tag. If it is likely that your translator needs to translate more than one attribute in a single tag, you must add a routine in the translator that appends numbers to mmTranslatedValue (for example, mmTranslatedValue1, mmTranslatedValue2, and so on).

The value of the mmTranslatedValue attribute must be a URL-encoded string that contains at least one valid attribute/value pair. This means that mmTranslatedValue="src=%22open.jpg%22" is a valid translation for both src="<? if (dayType == weekday) then open.jpg else closed.jpg" ?> and <? if (dayType == weekday) then src="open.jpg" else src="closed.jpg" ?>. mmTranslatedValue="%22open.jpg%22" is not valid for either example because it contains only the value, not the attribute.

Translating more than one attribute at a time

The mmTranslatedValue can contain more than one valid attribute/value pair. Consider the following untranslated code:

```
<img <? if (dayType==weekday) then src="open.jpg" width="320" ¬
  height="100" else
src="closed.jpg" width="100" height="320" ?> alt="We're open 24 ¬
hours a day from
12:01am Monday until 11:59pm Friday">
```

The following example shows how the translated markup might appear:

```
<img <? if (dayType==weekday) then src="open.jpg" width="320" ¬
height="100" else
src="closed.jpg" width="100" height="320" ?>
mmTranslatedValue="src=%22open.jpg%22 width=%22320%22 ¬
  height=%22100%22"
alt="We're open 24 hours a day from 12:01am Monday until 11:59pm ¬
Friday">
```

Notice that the spaces between the attribute/value pairs in the mmTranslatedValue are not encoded. Because Dreamweaver looks for these spaces when it attempts to render the translated value, each attribute/value pair in the mmTranslatedValue must be encoded separately and then pieced back together to form the full mmTranslatedValue. For an example of how to do this, see "A simple attribute translator example" on page 230.

A simple attribute translator example

To better understand attribute translation, it's helpful to look at an example. The following translator is "Pound Conditional" (Poco) markup, a syntax that's somewhat similar to ASP or PHP. The first step in making this translator work properly is to create a tagspec for Poco markup, which prevents Dreamweaver from parsing the untranslated Poco statements.

The following example shows the tagspec for Poco markup:

```
<tagspec tag_name="poco" start_string="<#" end_string="#>"
detect_in_attribute="true" icon="poco.gif" icon_width="17"
icon_height="15"></tagspec>
```

The poco.xml file that contains this tagspec is stored in the Configuration/ThirdPartyTags folder, along with the icon for Poco tags.

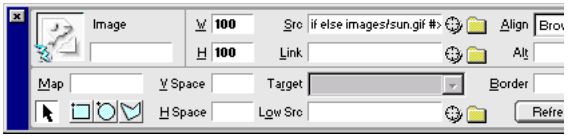
```
<html>
<head>
<title>Conditional Translator</title>
<meta http-equiv="Content-Type" content="text/html; charset=">
<script language="JavaScript">

/************************************************************
 * This translator handles the following statement syntaxes: *
 * <# if (condition) then foo else bar #>                    *
 * <# if (condition) then att="foo" else att="bar" #>        *
 * <# if (condition) then att1="foo" att2="jinkies"          *
 * att3="jeepers" else att1="bar" att2="zoinks" #>           *
 *                                                          *
 * It does not handle statements with no else clause.       *
 ************************************************************/

var count = 1;

function translateMarkup(docNameStr, siteRootStr, inStr){
var count = 1;
  // Counter to ensure unique mmTranslatedValues
var outStr = inStr;
  // String that will be manipulated
var spacer = "";
  // String to manage space between encoded attributes
var start = inStr.indexOf('<# if'); // 1st instance of Pound Conditional code

  // Declared but not initalized. //
var attAndValue;
  // Boolean indicating whether the attribute is part of
  // the conditional statement
var trueStart;
  // The beginning of the true case
var falseStart;
  // The beginning of the false case
var trueValue;
  // The HTML that would render in the true case
var attName;
  // The name of the attribute that is being'
  // set conditionally.
var equalSign;
  // The position of the equal sign just to the
  // left of the <#, if there is one
var transAtt;
  // The entire translated attribute
var transValue;
  // The value that must be URL-encoded
var back3FromStart;
  // Three characters back from the start position
  // (used to find equal sign to the left of <#
var tokens;
  // An array of all the attributes set in the true case
var end;
  // The end of the current conditional statement.


  // As long as there's still a <# conditional that hasn't been
  // translated
while (start != -1){
```

```
back3FromStart = start-3;
end = outStr.indexOf(' #>',start);
equalSign = outStr.indexOf('="<# if',back3FromStart);
attAndValue = (equalSign != -1)?false:true;
trueStart = outStr.indexOf('then', start);
falseStart = outStr.indexOf(' else', start);
trueValue = outStr.substring(trueStart+5, falseStart);
tokens = dreamweaver.getTokens(trueValue,' ');


// If attAndValue is false, find out what attribute you're
// translating by backing up from the equal sign to the
// first space. The substring between the space and the
// equal sign is the attribute.
if (!attAndValue){
  for (var i=equalSign; i > 0; i--){
    if (outStr.charAt(i) == " "){
      attName = outStr.substring(i+1,equalSign);
      break;
    }
  }
  transValue = attName + '="' + trueValue + '"';
  transAtt = ' mmTranslatedValue' + count + '="' + ¬
  escape(transValue) + '"';
  outStr = outStr.substring(0,end+4) + transAtt + ¬
  outStr.substring(end+4);

// If attAndValue is true, and tokens is greater than
// 1, then trueValue is a series of attribute/value
// pairs, not just one. In that case, each attribute/value
// pair must be encoded separately and then added back
// together to make the translated value.
}else if (tokens.length > 1){
  transAtt = ' mmTranslatedValue' + count + '="'
  for (var j=0; j < tokens.length; j++){
    tokens[j] = escape(tokens[j]);
    if (j>0){
      spacer=" ";
    }
    transAtt += spacer + tokens[j];
  }
  transAtt += '"';
  outStr = outStr.substring(0,end+3) + transAtt + ¬
  outStr.substring(end+3)

// If attAndValue is true and tokens is not greater
// than 1, then trueValue is a single attribute/value pair.
// This is the simplest case, where all that is necessary is
// to encode trueValue.
}else{
  transValue = trueValue;
  transAtt = ' mmTranslatedValue' + count + '="' + ¬
  escape(transValue) + '"';
  outStr = outStr.substring(0,end+3) + transAtt + ¬
  outStr.substring(end+3);
}

// Increment the counter so that the next instance
// of mmTranslatedValue will have a unique name, and
// then find the next <# conditional in the code.
count++;
start = outStr.indexOf('<# if',end);
```

```
  }

 // Return the translated string.
 return outStr
}

function getTranslatorInfo(){
 returnArray = new Array(7);

 returnArray[0] = "Pound_Conditional";     // The translatorClass
 returnArray[1] = "Pound Conditional Translator"; // The title
 returnArray[2] = "2";                    // The number of extensions
 returnArray[3] = "html";                 // The first extension
 returnArray[4] = "htm";                  // The second extension
 returnArray[5] = "1";                    // The number of expressions
 returnArray[6] = "<#";                   // The first expression
 returnArray[7] = "byString";                  //
 returnArray[8] = "50";                   //



 return returnArray
}

</script>
</head>

<body>
</body>
</html>
```

Inspecting translated attributes

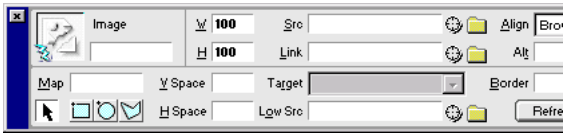
When server markup specifies a single attribute and the attribute is represented in a Property inspector, Dreamweaver displays the server markup in the Property inspector.



The markup appears whether a translator is associated with it. The translator runs whenever the user edits the server markup that is shown in the panel.

Note: The lightning bolt icon does not appear when text or table cells, rows, or columns are selected. Translation continues if the user edits server markup in the panel, and a translator exists to handle that type of markup.

When server markup controls more than one attribute in a tag, the server markup does not appear in the Property inspector. However, the lightning bolt shows that translated markup exists for the selected element.



The fields in the Property inspector are still editable; users can enter values for attributes that might be controlled by server markup, which results in duplicate attributes. If both a translated value and a regular value are set for a particular attribute, Dreamweaver displays the translated value in the Document window. You must decide whether your translator looks for duplicate attributes and remove them.

Locking translated tags or blocks of code

In most cases, you want a translator to change markup so that Dreamweaver can display it, but you want the original markup—not the changes—to be saved. To address this need, Dreamweaver provides special XML tags in which to wrap translated content and to refer to the original code.

When you use these XML tags, the contents of the original attributes are duplicated in Code view. If the file is saved, the original, untranslated markup is written to the file. The untranslated content is what Dreamweaver displays in Code view.

The syntax of the XML tags is shown in the following example:

```
<MM:BeginLock translatorClass="translatorClass" ¬
type="tagNameOrType" depFiles="dependentFilesList" ¬
orig="encodedOrignalMarkup">
Translated content
<MM:EndLock>
```

where:

translatorClass is the unique identifier for the translator (the first string in the array that `getTranslatorInfo()` returns).

tagNameOrType is a string that identifies the type of markup (or the tag name that is associated with the markup) that is contained in the lock. The string can contain only alphanumeric, hyphen (-), or underscore (_) characters. You can check this value in the `canInspectSelection()` function of a custom Property inspector to determine if the Property inspector is the right one for the content. For more information, see “Creating Property inspectors for locked content” on page 239. Locked content cannot be inspected by any of the Dreamweaver built-in Property inspectors. For example, specifying `type="IMG"` does not make the Image panel appear.

dependentFilesList is a string that contains a comma-separated list of files on which the locked markup depends. Files are referenced as URLs, relative to the user’s document. If the user updates one of the files in the *dependentFilesList*, Dreamweaver automatically retranslates the content in the document that contains the list.

encodedOriginalMarkup is a string that contains the original, untranslated markup, encoded using a small subset of URL encoding (use `%22 for "`, `%3C for <`, `%3E for >`, and `%25 for %`). The quickest way to URL-encode a string is to use the `escape()` method. For example, if `myString` equals `’<img src="foo.gif">’`, `escape(myString)` returns `%3Cimg%20src=%22foo.gif%22%3E`.

The following example shows the locked portion of code that might be generated from the translation of the server-side include `<!--#include virtual="/footer.html" -->`:

```
<MM:BeginLock translatorClass="MM_SSI" type="ssi" ¬
depFiles="C:\sites\webdev\footer.html" orig="%3C!--#include ¬
virtual=%22/footer.html%22%20--%3E">
<!-- begin footer -->
<CENTER>
<HR SIZE=1 NOSHADE WIDTH=100%>

<BR>

[<A TARGET="_top" HREF="/">home</A>]
[<A TARGET="_top" HREF="/products/">products</A>]
[<A TARGET="_top" HREF="/services/">services</A>]
[<A TARGET="_top" HREF="/support/">support</A>]
[<A TARGET="_top" HREF="/company/">about us</A>]
[<A TARGET="_top" HREF="/help/">help</A>]
</CENTER>
<!-- end footer -->
<MM:EndLock>
```

A simple block/tag translator example

To better understand translation, it's helpful to look at a translator that is written entirely in JavaScript (that is, one that does not rely on a C library for any functionality). The following translator would be more efficient if it was written in C, but the JavaScript version is simpler, which makes it perfect for demonstrating how translators work.

As with most translators, this one is designed to mimic server behavior. Assume that your web server is configured to replace the `KENT` tag with a different picture of an engineer, depending on the day of the week, the time of day, and the user's platform. The translator does the same thing, only locally.

```
<html>
<head>
<title>Kent Tag Translator</title>
<meta http-equiv="Content-Type" content="text/html; charset=">
<script language="JavaScript">
/*********************************************************
 * The getTranslatorInfo() function provides information  *
 * about the translator, including its class and name,    *
 * the types of documents that are likely to contain the  *
 * markup to be translated, the regular expressions that  *
 * a document containing the markup to be translated      *
 * would match (whether the translator should run on all  *
 * files, no files, in files with the specified           *
 * extensions, or in files matching the specified         *
 * expressions).                                          *
 *********************************************************/
function getTranslatorInfo(){
  //Create a new array with 6 slots in it
  returnArray = new Array(6);
```

```
  returnArray[0] = "DREAMWEAVER_TEAM"// The translatorClass
  returnArray[1] = "Kent Tags"// The title
  returnArray[2] = "0" // The number of extensions
  returnArray[3] = "1"// The number of expressions
  returnArray[4] = "<kent"// Expression
  returnArray[5] = "byExpression"// run if the file contains "<kent"
  return returnArray;
}

/*************************************************************************
* The translateMarkup() function performs the actual translation.         *
* In this translator, the translateMarkup() function is written           *
* entirely in JavaScript (that is, it does not rely on a C library) --    *
* and it's also extremely inefficient. It's a simple example, however,    *
* which is good for learning.                                             *
*************************************************************************/
function translateMarkup(docNameStr, siteRootStr, inStr){
  var outStr = "";                 // The string to be returned after
  translation
  var start = inStr.indexOf('<kent>');     // The first position of the KENT
  tag
                                           // in the document.
  var replCode = replaceKentTag();        // Calls the replaceKentTag()
  function
                                    // to get the code that will replace KENT.
  var outStr = "";                 // The string to be returned after
  translation

   //If the document does not contain any content, terminate the translation.
  if ( inStr.length <= 0 ){
     return "";
  }

  // As long as start, which is equal to the location in inStr of the
  // KENT tag, is not equal to -1 (that is, as long as there is another
  // KENT tag in the document)
  while (start != -1){
     // Copy everything up to the start of the KENT tag.
     // This is very important, as translators should never change
     // anything other than the markup that is to be translated.
     outStr = inStr.substring(0, start);
     // Replace the KENT tag with the translated HTML, wrapped in special
     // locking tags. For more information on the replacement operation, see
     // the comments in the replaceKentTag() function.
     outStr = outStr + replCode;

     // Copy everything after the KENT tag.
     outStr = outStr + inStr.substring(start+6);

     // Use the string you just created for the next trip through
     // the document. This is the most inefficient part of all.
     inStr = outStr;
     start = inStr.indexOf('<kent>');

  }
  // When there are no more KENT tags in the document, return outStr.
  return outStr;
}
```

```
/*************************************************************
* The replaceKentTag() function assembles the HTML that will  *
* replace the KENT tag and the special locking tags that will *
* surround the HTML. It calls the getImage() function to      *
* determine the SRC of the IMG tag.                           *
*************************************************************/
function replaceKentTag(){
  // The image to display.
  var image = getImage();
  // The location of the image on the local disk.
  var depFiles = dreamweaver.getSiteRoot() + image;
  // The IMG tag that will be inserted between the lock tags.
  var imgTag = '<IMG SRC="/' + image + '" WIDTH="320" HEIGHT="240"
  ALT="Kent">\n';
  // 1st part of the opening lock tag. The remainder of the tag is assembled
  below.
  var start = '<MM:BeginLock translatorClass="DREAMWEAVER_TEAM" type="kent"';
  // The closing lock tag.
  var end = '<MM:EndLock>';

  //Assemble the lock tags and the replacement HTML.
  var replCode = start + ' depFiles="' + depFiles + '"';
  replCode = replCode + ' orig="%3Ckent%3E">\n';
  replCode = replCode + imgTag;
  replCode = replCode + end;

  return replCode;
}

/*********************************************************************
 * The getImage() function determines which image to display      *
 * based on the day of the week, the time of day and the          *
 * user's platform. The day and time are figured based on UTC     *
 * time (Greenwich Mean Time) minus 8 hours, which gives          *
 * Pacific Standard Time (PST). No allowance is made for Daylight *
 * Savings Time in this routine.                                  *
*********************************************************************/
function getImage(){
  var today = new Date();            // Today's date & time.
  var day = today.getUTCDay();       // Day of the week in the GMT time zone.
                                     // 0=Sunday, 1=Monday, and so on.
  var hour = today.getUTCHours();    // The current hour in GMT, based on the
                                     // 24-hour clock.
  var SFhour = hour - 8;             // The time in San Francisco, based on the
                                     // 24-hour clock.
  var platform = navigator.platform; // User's platform. All Windows machines
                                  // are identified by Dreamweaver as "Win32",
                                     // all Macs as "MacPPC".
  var imageRef;                      // The image reference to be returned.
// If SFhour is negative, you have two adjustments to make.
   // First, subtract one from the day count because it is already the wee
   // hours of the next day in GMT. Second, add SFhour to 24 to
   // give a valid hour in the 24-hour clock.
   if (SFhour < 0){
     day = day - 1;
       // The day count back one would make it negative, and it's Saturday,
       // so set the count to 6.
       if (day < 0){
           day = 6;
       }
     SFhour = SFhour + 24;
   }
```

```
  // Now determine which photo to show based on whether it's a workday or a
  // weekend; what time it is; and, if it's a time and day when Kent is
  // working, what platform the user is on.

  //If it's not Sunday
  if (day != 0){
     //And it's between 10am and noon, inclusive
     if (SFhour >= 10 && SFhour <= 12){
        imageRef = "images/kent_tiredAndIrritated.jpg";
      //Or else it's between 1pm and 3pm, inclusive
     }else if (SFhour >= 13 && SFhour <= 15){
        imageRef =  "images/kent_hungry.jpg";
     //Or else it's between 4pm and 5pm, inclusive
     }else if (SFhour >= 16 && SFhour <= 17){
        //If user is on Mac, show Kent working on Mac
        if (platform == "MacPPC"){
           imageRef = "images/kent_gettingStartedOnMac.jpg";
        //If user is on Win, show Kent working on Win
        }else{
           imageRef = "images/kent_gettingStartedOnWin.jpg";
        }
     //Or else it's after 6pm but before the stroke of midnight
     }else if (SFhour >= 18){
         //If it's Saturday
         if (day == 6){
           imageRef = "images/kent_dancing.jpg";
        //If it's not Saturday, check the user's platform
        }else if (platform == "MacPPC"){
           imageRef = "images/kent_hardAtWorkOnMac.jpg";
        }else{
           imageRef = "images/kent_hardAtWorkOnWin.jpg";
        }
     }else{
        imageRef = "images/kent_sleeping.jpg";
     }
  //If it's after midnight and before 10am, or anytime on Sunday
  }else{
     imageRef = "images/kent_sleeping.jpg";
  }

  return imageRef;
}


</script>
</head>

<body>
</body>
</html>
```

Creating Property inspectors for locked content

After you've created a translator, you need to create a Property inspector for the content so the user can change its properties (for example, the file to be included or one of the conditions in a conditional statement). Inspecting translated content is a unique problem for several reasons:

- The user might want to change the properties of the translated content, and those changes must be reflected in the untranslated content.
- The DOM contains the translated content (that is, the lock tags and the tags they surround are nodes in the DOM), but the `outerHTML` property of the `documentElement` and the `dreamweaver.getSelection()` and `dreamweaver.nodeToOffsets()` functions act on the untranslated source.
- The tags you inspect are different before and after translation.

A Property inspector for the `HAPPY` tag might have a comment that looks similar to the following code:

```
<!-- tag:HAPPY,priority:5,selection:exact,hline,vline, attrName:xxx,¬
  attrValue:yyy -->
```

The Property inspector for the translated `HAPPY` tag, however, would have a comment that looks similar to the following code:

```
<!-- tag:*LOCKED*,priority:5,selection:within,hline,vline -->
```

The `canInspectSelection()` function for the untranslated `HAPPY` Property inspector is simple: Because the selection type is exact, it can return `true` without further analysis. For the translated `HAPPY` Property inspector, this function is more complicated; the keyword `*LOCKED*` indicates that the Property inspector is appropriate when the selection is within a locked region, but because a document can have several locked regions, further checks must be performed to determine if the Property inspector matches this particular locked region.

Another problem is inherent in inspecting translated content. When you call `dom.getSelection()`, the values that return by default are offsets into the untranslated source. To expand the selection properly so that the locked region (and only the locked region) is selected, use the following method:

```
var currentDOM = dw.getDocumentDOM();
var offsets = currentDOM.getSelection();
var theSelection = currentDOM.offsetsToNode(offsets[0],offsets[0]+1);
```

Using `offsets[0]+1` as the second argument ensures that you remain within the opening lock tag when you convert the offsets to a node. If you use `offsets[1]` as the second argument, you risk selecting the node above the lock.

After you make the selection (after ensuring that its nodeType is `node.ELEMENT_NODE`), you can inspect the `type` attribute to see if the locked region matches this Property inspector, as shown in the following example:

```
if (theSelection.nodeType == node.ELEMENT_NODE && ¬
theSelection.getAttribute('type') == 'happy'){
  return true;
}else{
  return false
}
```

To populate the fields in the Property inspector for the translated tag, you must parse the value of the `orig` attribute. For example, if the untranslated code is `<HAPPY TIME="22">` and the Property inspector has a field that is labeled Time, you must extract the value of the `TIME` attribute from the `orig` string.

```
function inspectSelection() {
  var currentDOM = dw.getDocumentDOM();
  var currSelection = currentDOM.getSelection();
  var theObj = currentDOM.offsetsToNode¬
  (curSelection[0],curSelection[0]+1);

  if (theObj.nodeType != Node.ELEMENT_NODE) {
    return;
  }

  // To convert the encoded characters back to their
  // original values, use the unescape() method.
  var origAtt = unescape(theObj.getAttribute("ORIG"));

  // Convert the string to lower case for processing
  var origAttLC = origAtt.toLowerCase();

  var timeStart = origAttLC.indexOf('time="');
  var timeEnd = origAttLC.indexOf('"',timeStart+6);
  var timeValue = origAtt.substring(timeStart+6,timeEnd);

  document.layers['timelayer'].document.timeForm.timefield.¬
  value = timeValue;
}
```

After you parse the `orig` attribute in order to populate the fields in the Property inspector for the translated tag, your next step is probably to set the value of the `orig` attribute if the user changes the value in any of the fields. You might find restrictions against making changes in a locked region. You can avoid this problem by changing the original markup and retranslating.

The Property inspector for translated server-side includes (Configuration/Inspectors/ssi_translated.js) demonstrates this technique in its `setComment()` function. Rather than rewriting the `orig` attribute, the Property inspector assembles a new SSI comment. It inserts that comment into the document in place of the old one by rewriting the entire document contents, which generates a new `orig` attribute. The following code summarizes this technique:

```
// Assemble the new include comment. radioStr and URL are
// variables defined earlier in the code.
newInc = "<!--#include " + radioStr + "=" + '"' + URL + '"' ¬
+" -->";

// Get the contents of the document.
var entireDocObj = dreamweaver.getDocumentDOM();
var docSrc = entireDocObj.documentElement.outerHTML;

// Store everything up to the SSI comment and everything after
// the SSI comment in the beforeSelStr and afterSelStr variables.
var beforeSelStr = docSrc.substring(0, curSelection[0] );
var afterSelStr  = docSrc.substring(curSelection[1]);

// Assemble the new contents of the document.
docSrc = beforeSelStr + newInc + afterSelStr;

// Set the outerHTML of the HTML tag (represented by
// the documentElement object) to the new contents,
// and then set the selection back to the locked region
// surrounding the SSI comment.
entireDocObj.documentElement.outerHTML = docSrc;
entireDocObj.setSelection(curSelection[0], curSelection[0]+1);
```

Finding bugs in your translator

If the `translateMarkup()` function contains certain types of errors, the translator loads properly, but it fails silently when invoked. Although failing silently prevents Dreamweaver from becoming unstable, it can hinder development, especially when you need to find one small syntax error in multiple lines of code.

If your translator fails, one effective debugging method is to turn the translator into a command, as described in the following steps:

1. Copy the entire contents of the translator file to a new document, and save it in the Configuration/Commands folder inside the Dreamweaver application folder.
2. At the top of the document, between the `SCRIPT` tags, add the following function:

```
function commandButtons(){
  return new Array( "OK","translateMarkup(dreamweaver.¬
  getDocumentPath('document'), dreamweaver.getSiteRoot(), ¬
  dreamweaver.getDocumentDOM().documentElement.outerHTML); ¬
  window.close()", "Cancel", "window.close()");
}
```

3. At the end of the `translateMarkup()` function, comment out the `return whateverTheReturnValueIs` line, and replace it with `dreamweaver.getDocumentDOM().documentElement.outerHTML = whateverTheReturnValueIs`:

```
  // return theCode;
  dreamweaver.getDocumentDOM().documentElement.outerHTML = ¬
  theCode;
}
/* end of translateMarkup() */
```

4. In the `BODY` of the document, add a form with no text boxes:

```
<body>
<form>
Hello.
</form>
</body>
```

5. Restart Dreamweaver and select your translator command from the Commands menu. When you click OK, the `translateMarkup()` function is called, simulating translation.

   If no error message appears and translation still fails, you probably have a logic error in your code.

6. Add `alert()` statements in strategic spots throughout the `translateMarkup()` function so you can make sure you're getting the proper branches and so you can check the values of variables and properties at different points:

```
for (var i=0; i< foo.length; i++){
  alert("we're at the top of foo.length array, and the value ¬
  of i is " + i);
  /* rest of loop */
}
```

7. After adding in the `alert()` statements, choose your command from the Commands menu, click Cancel, and choose it again. This reloads the command file and incorporates your changes.

CHAPTER 20
JavaScript Debugger Modules

A JavaScript Debugger module is an extensibility module that inserts special code into a document so the code can interface with the JavaScript Debugger. The modules are located with the Dreamweaver Program Files in the Configuration/Debugger subfolder. These modules insert specific JavaScript and HTML into a working document to create a "debug version" of the document the next time that the JavaScript Debugger runs. The debug version is simply a set of temporary replicated files for the HTML document and each external JavaScript file, created by Macromedia Dreamweaver MX and saved in the current working folder. The debug version of the HTML file appears in the browser. The JavaScript that is inserted into the temporary files, called instrumentation, communicates with the Dreamweaver JavaScript Debugger as the JavaScript executes in the browser.

For information about JavaScript Debugger API Commands, see "JavaScript debugger functions" on page 498.

How the JavaScript Debugger module works

Dreamweaver comes with two JavaScript Debugger modules, one for each supported browser, Netscape Navigator and Microsoft Internet Explorer. To provide support for a different browser, you must create a new module and use `dom.instrumentDocument` and `dreamweaver.startDebugger` to debug the document in that browser.

When you call `dom.instrumentDocument`, the specified module receives callbacks as Dreamweaver parses the JavaScript in the document. So, for example, you could create a JavaScript Debugger module that inserts comments or records information about the JavaScript code, instead of inserting debugging enhancements.

When `dom.instrumentDocument` is called with a specific module, the following steps occur:

1. Dreamweaver calls `getIncludeFiles()` in the module. This function returns the list of files that will be referenced from the HTML instrumentation code that is returned from `getHeadInstrument()` and `getBodyInstrument()`, which are called in steps 13 and 14. The include files can be any type of file, such as an external JavaScript file, JavaApplet, or ActiveX control. All the files must be in the Configuration/Debugger subfolder with the module. Dreamweaver will copy the include files to the directory that contains the file being debugged, and then will delete the include files from that directory when Dreamweaver exits.
2. Next, the HTML document is scanned for script tags and event handlers. The code inside the script tag, in an external JavaScript file or in an event handler, is called a block.

   Note: An external JavaScript file is a file that is specified as the src attribute of a SCRIPT tag.

3. Dreamweaver parses script tags in the `HEAD` section first.

4. When Dreamweaver finds a script tag or event handler, it calls the `startBlock()` function of the module and passes in the name of the file and the line and character offsets from the beginning of the file.
5. Dreamweaver begins parsing the JavaScript code in the block.
6. When Dreamweaver finds a JavaScript statement, such as a variable declaration, it calls `getStepInstrument()`, passing the line and character offsets and other information. The module returns a string of JavaScript code that is inserted before the statement. You must take care to insert valid JavaScript code. For each call to `getStepInstrument()`, Dreamweaver records the line number as a valid breakpoint line regardless of the instrumentation that returns. So, when the debugger is started with `dw.startDebugger()`, the breakpoints that are already set by the user will be moved to one of these valid lines.
7. When Dreamweaver finds a function declaration, it calls `getFunctionStartInstrument()` to receive the instrumentation to be inserted at the beginning of the function.

   Note: This is not considered a valid breakpoint line.
8. Dreamweaver continues parsing the function, calling `getStepInstrument()` for each statement in the function.
9. When Dreamweaver comes to a return statement, or the end of the function, it calls `getFunctionEndInstument()` to receive the instrumentation to be inserted before the function returns.

   Note: This is not considered a valid breakpoint line.
10. If Dreamweaver encounters a syntax error or warning in the JavaScript block, it calls `reportError()` or `reportWarning()`, respectively. After an error is encountered, Dreamweaver stops parsing the block. Other blocks continue to be parsed.
11. After Dreamweaver has parsed all the script blocks in the `HEAD` section, it calls `getHeadInstrument()` to get the HTML instrumentation to insert in the `HEAD` section.

    Note: This function should return HTML, not JavaScript. If the module needs to insert JavaScript code in the `HEAD`, it must enclose it in a `SCRIPT` tag.
12. Dreamweaver begins processing the JavaScript blocks (`SCRIPT` tags and event handlers) in the `BODY` section of the document.
13. After the last block in the `BODY` section is processed, Dreamweaver calls `getBodyInstrument()` to get the HTML instrumentation to insert in the `BODY` section.

    Note: This function should return HTML, not JavaScript.
14. After Dreamweaver calls `getBodyInstrument()`, there is one final call to `startBlock()` and `getStepInstrument()` for an auto-breakpoint. The instrumentation does not correspond to any user-defined `SCRIPT` tag, but instead, it is inserted in a new `SCRIPT` tag after the `BODY` instrumentation. Unlike other calls to `getStepInstrument()`, this line is not considered a valid line on which the user can set a breakpoint, but instead, it is treated as a special breakpoint where the debugger always stops.
15. Finally, Dreamweaver calls `getOnUnloadInstrument()` to get JavaScript instrumentation to be inserted in the `onUnload` handler of the `BODY` tag. If the document already has an `onUnload` handler, this instrumentation is inserted after the user-defined `onUnload` code.

The JavaScript Debugger module API

The JavaScript Debugger module API lets you customize the way you create the debug version of a document. You need to create a debugger module if you want to make the Dreamweaver JavaScript Debugger work with a browser other than Netscape Navigator and Internet Explorer, which Dreamweaver supports. You can create a module for a specialized purpose, such as counting the number of JavaScript statements that are used in a particular document.

Note: Currently only `SCRIPT` tags and event handlers are parsed for instrumentation. There are some other ways to use JavaScript in HTML documents, such as JavaScript URLs, JavaScript entities, and conditional comments, but these methods are not currently supported.

The JavaScript Debugger module API functions are significant only in the context of module files. Specifically, Dreamweaver automatically calls the `getStepInstrument()` function if it is defined in the module file. For any other extension file, a function named `getStepInstrument()` acts as a user-defined function; you must call it explicitly.

Unlike working with functions in the main JavaScript API, you are responsible for writing the body of each function and returning a value, if required, for the modules. For the functions in the main API, you call and pass arguments, and Dreamweaver generates return values, if any. For the JavaScript Debugger modules, Dreamweaver calls the functions and passes arguments to them, and you generate return values, if any.

All the JavaScript Debugger module functions are optional. If a function is not defined, nothing happens when Dreamweaver calls it.

getFunctionEndInstrument()

Availability

Dreamweaver 4

Description

Called after the last statement in a function declaration. If any return statements exist, this module is called to insert instrumentation after the return value is evaluated and before the function can return strings.

Arguments

None.

Returns

Dreamweaver expects a string that contains the JavaScript to insert at the end of the function.

getFunctionStartInstrument()

Availability

Dreamweaver 4

Description

Called before the first statement in a function declaration. The `getStepInstrument()` function is also called for the statement.

Arguments

None.

Returns

Dreamweaver expects a string that contains the JavaScript to insert at the beginning of the function.

getBodyInstrument()m

Availability

Dreamweaver 4

Description

This function is called exactly once after all the blocks in the `HEAD` section are processed by the instrumentation JavaScript.

Arguments

None.

Returns

Dreamweaver expects a string that contains HTML to insert at the top of the `<BODY>` section.

getHeadInstrument()

Availability

Dreamweaver 4

Description

This function is called exactly once after all blocks in the `HEAD` section are processed by the instrumentation JavaScript, but before the `BODY` section blocks are instrumented.

Arguments

None.

Returns

Dreamweaver expects a string that contains HTML to insert at the top of the `<HEAD>` section.

getIncludedFileList()

Availability

Dreamweaver 4

Description

Called to get a list of files that are referenced by the code that is inserted in the head or body from the `getHeadInstrument()` and `getBodyInstrument()` functions. These files must be located in the Configuration/Debugger directory of the Dreamweaver program files with the instrumentation debugger module.

Arguments

None.

Returns

Dreamweaver expects an array of filenames that should be copied to the directory with the file that is processed by the instrumentation JavaScript.

getOnUnloadInstrument()

Availability

Dreamweaver 4

Description

This function is called exactly once after `getHeadInstrument()` is called.

Arguments

None.

Returns

Dreamweaver expects a string that contains JavaScript to insert at the end of the `onUnload` event handler of the `BODY` tag.

getStepInstrument()

Availability

Dreamweaver 4

Description

Called for each statement that is parsed inside a block. A call is always made to `StartBlock()` before this function is called. Dreamweaver records each line for which it calls this function as a valid breakpoint line. When the debugger starts, all breakpoints are moved to valid breakpoint lines.

Arguments

`lineNumber`, *`offset`*, *`bisInFuncton`*

- *`lineNumber`* is the line number of the next statement that is relative to the start of the block (1-based index).
- *`offset`* is the offset of the first character of the next statement that is relative to the start of the block (0-based index).
- *`bisInFunction`* is a Boolean value that indicates if the step is in a function definition (`true`) or in the global scope (`false`).

Returns

Dreamweaver expects a string that contains the JavaScript code to insert before the statement.

reportError()

Availability

Dreamweaver 4

Description

Called when a syntax error is detected. The errors and warnings are not necessarily reported in order.

Arguments

`FileURL, fileName`, *`errorNumber`*, *`strDesc`*, *`lineNumber`*, *`offset`*

- *`fileURL`* is the full path name of the report file, expressed as a file://URL, of the file containing the error.
- *`fileName`* is the name of the file.
- *`errorNumber`* is the numeric identifier of the error that occurred.
- *`strDesc`* is the description of the error.
- *`lineNumber`* is the line number in which the error occurred, relative to the start of the block.
- *`offset`* is the offset of the character at which the error occurred, relative to the start of the block.

Returns

Nothing.

reportWarning()

Availability

Dreamweaver 4

Description

Called when a warning is detected in the file.

Arguments

fileURL, fileName, *errorNumber*, *strDesc*, *lineNumber*, *offset*

- *fileURL* is the full path name of the report file, expressed as a file://URL, of the file containing the error.
- *fileName* is the name of the file.
- *errorNumber* is the numeric identifier of the warning that occurred.
- *strDesc* is the description of the warning.
- *lineNumber* is the line number in which the warning occurred, relative to the start of the block.
- *offset* is the offset of the character at which the warning occurred, relative to the start of the block.

Returns

Dreamweaver expects nothing.

startBlock()

Availability

Dreamweaver 4

Description

Indicates the beginning of a new block of JavaScript code. The block can be a script tag, event handler, or external .js file.

Arguments

fileName, *lineNumber*, *offset*

- *fileName* is the name of the HTML document or .js file that contains the block. The location is specified by a relative path to the source HTML document.
- *lineNumber* is the line number in the HTML document or .js file in which the block begins (1-based index).
- *offset* is the offset of the first character of JavaScript code from the beginning of the file (0-based index).

Returns

Dreamweaver expects nothing.

CHAPTER 21
C-Level Extensibility

The C-level extensibility mechanism lets you implement Macromedia Dreamweaver MX extensibility files using a combination of JavaScript and your own C code. You define functions using C, bundle them in a DLL or shared library, save the library in the Configuration/JSExtensions folder within the Dreamweaver application folder, and then call the functions from JavaScript using the JavaScript interpreter that is built into Dreamweaver.

For example, you might want to define a Dreamweaver object that inserts the contents of a user-specified file into the current document. Because client-side JavaScript does not provide support for file I/O, you must write a function in C to provide this functionality.

You can use the following HTML and JavaScript to create a simple Insert Text from File object. Notice that the `objectTag()` function calls a C function named `readContentsOfFile()`, which is stored in a library named `myLibrary`.

```
<HTML>
<HEAD>
<SCRIPT>
function objectTag() {
  fileName = document.forms[0].myFile.value;
  return myLibrary.readContentsOfFile(fileName);
}
</SCRIPT>
</HEAD>

<BODY>
<FORM>
Enter the name of the file to be inserted:
<INPUT TYPE="file" NAME="myFile">
</FORM>
</BODY>
</HTML>
```

The `readContentsOfFile()` function accepts a list of arguments from the user, unpacks the argument that contains the filename, reads the contents of the file, and packages the contents of the file as the return value. For more information about the JavaScript data structures and functions that appear in `readContentsOfFile()`, see "C-level extensibility and the JavaScript interpreter" on page 253.

```
JSBool
readContentsOfFile(JSContext *cx, JSObject *obj, unsigned int ¬
argc, jsval *argv, jsval *rval)
{
  char *fileName, *fileContents;
  JSBool success;
  unsigned int length;

  /* Make sure caller passed in exactly one argument. If not,
   * then tell the interpreter to abort script execution.  */
  if (argc != 1){
    JS_ReportError(cx, "Wrong number of arguments", 0);
  return JS_FALSE;
  }

  /* Convert the argument to a string */
  fileName = JS_ValueToString(cx, argv[0], &length);
  if (fileName == NULL){
    JS_ReportError(cx, "The argument must be a string", 0);
    return JS_FALSE;
  }

  /* Use the string (the file name) to open and read a file */
  fileContents = exerciseLeftToTheReader(fileName);

  /* Store file contents in rval, which is the return value ¬
     passed
  * back to the caller */
  success = JS_StringToValue(cx, fileContents, 0, *rval);
  free(fileContents);

  /* Return true to continue or false to abort the script */
  return success;
}
```

To ensure that the `readContentsOfFile()` function executes as designed rather than causing a JavaScript error, you must register the function with the JavaScript interpreter by including a function called `MM_Init()` in your library. When Dreamweaver loads the library at startup, it calls the `MM_Init()` function to get the following three pieces of information:

- The JavaScript name of the function
- A pointer to the function
- The number of arguments that the function expects

The following example shows how `MM_Init()` function for `myLibrary` might look:

```
void
MM_Init()
{
  JS_DefineFunction("readContentsOfFile", readContentsOfFile, 1);
}
```

Your library must include exactly one instance of the following macro:

```
/* MM_STATE is a macro that expands to some definitions that are
 * needed to interact with Dreamweaver. This macro must
 * be defined exactly once in your library. */
MM_STATE
```

Note: The library can be implemented in either C or C++, but the file that contains `MM_Init()` and `MM_STATE` must be implemented in C. The C++ compiler garbles function names, which makes it impossible for Dreamweaver to find the `MM_Init()` function.

C-level extensibility and the JavaScript interpreter

The C code in your library must interact with the Dreamweaver JavaScript interpreter at three different times:

- At startup, to register the library's functions
- When the function is called, to unpack the arguments that are being passed from JavaScript to C
- Before the function returns, to package the return value

To accomplish these tasks, the interpreter defines several data types and exposes an API. Definitions for the data types and functions that are listed in this section appear in the `mm_jsapi.h` file. For your library to work properly, you must include `mm_jsapi.h` at the top of each file in your library with the following line:

```
#include "mm_jsapi.h"
```

Including the `mm_jsapi.h` file includes, in turn, `mm_jsapi_environment.h`, which defines the `MM_Environment` structure.

Data Types

The JavaScript interpreter defines the following data types.

typedef struct JSContext JSContext

Description

A pointer to this opaque data type passes to the C-level function. Some functions in the API accept this pointer as one of their arguments.

typedef struct JSObject JSObject

Description

A pointer to this opaque data type passes to the C-level function. This data type represents an object, which may be an array object or some other object type.

typedef struct jsval jsval

Description

An opaque data structure that can contain an integer, or a pointer to a float, string, or object. Some functions in the API can be used to read the values of function arguments by reading the contents of a `jsval`, and some can be used to write the function's return value by writing a `jsval`.

typedef enum { JS_FALSE = 0, JS_TRUE = 1 } JSBool

Description

A simple data type that stores a Boolean value.

The C-level API

The C-level extensibility API consists of the following functions.

typedef JSBool (*JSNative)(JSContext *cx, JSObject *obj, unsigned int argc, jsval *argv, jsval *rval)

Description

This function signature describes C-level implementations of JavaScript functions in the following situations:

- `cx` is a pointer to an opaque `JSContext` structure, which must be passed to some of the functions in the JavaScript API. This variable holds the interpreter's execution context.
- *`obj`* is a pointer to the object in whose context the script executes. While the script is running, the `this` keyword is equal to this object.
- *`argc`* is the number of arguments being passed to the function.
- *`argv`* is a pointer to an array of `jsvals`. The array is `argc` elements in length.
- *`rval`* is a pointer to a single `jsval`. The function's return value should be written to `*rval`.

The function returns `JS_TRUE` upon success or `JS_FALSE` upon failure. If the function returns `JS_FALSE`, the current script stops executing and an error message appears.

JSBool JS_DefineFunction()

Description

Registers a C-level function with the JavaScript interpreter in Dreamweaver. After `JS_DefineFunction()` registers the C-level function that you specify in the *`call`* argument, you can invoke it in a JavaScript script by referring to it with the name that you specify in the *`name`* argument. The `name` is case-sensitive.

Typically, this function is called from `MM_Init()`, which Dreamweaver calls during startup.

Arguments

`char *`*`name`*, `JSNative` *`call`*, `unsigned int` *`nargs`*

- *`name`* is the name of the function as it is exposed to JavaScript.
- *`call`* is a pointer to a C-level function. The function must accept the same arguments as `readContentsOfFile`, and it must return a `JSBool`, which indicates success or failure.
- *`nargs`* is the number of arguments that the function expects to receive.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

char *JS_ValueToString()

Description

Extracts a function argument from a `jsval`, converts it to a string, if possible, and passes the converted value back to the caller.

Note: Do not modify the returned buffer pointer or you might corrupt the data structures of the JavaScript interpreter. To change the string, you must copy the characters into another buffer and create a new JavaScript string.

Arguments

`JSContext *cx, jsval v, unsigned int *pLength`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *v* is the `jsval` from which the string is to be extracted.
- *pLength* is a pointer to an unsigned integer. This function sets `*plength` equal to the length of the string in bytes.

Returns

A pointer to a null-terminated string on success or to `NULL` on failure. The calling routine must not free this string when it is finished with it.

JSBool JS_ValueToInteger()

Description

Extracts a function argument from a `jsval`, converts it to an integer (if possible), and passes the converted value back to the caller.

Arguments

`JSContext *cx, jsval v, long *lp`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *v* is the `jsval` from which the integer is to be extracted.
- *lp* is a pointer to a 4-byte integer. This function stores the converted value in `*lp`.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_ValueToDouble()

Description

Extracts a function argument from a `jsval`, converts it to a double (if possible), and passes the converted value back to the caller.

Arguments

`JSContext *cx, jsval v, double *dp`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *v* is the `jsval` from which the double is to be extracted.
- *dp* is a pointer to an 8-byte double. This function stores the converted value in `*dp`.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_ValueToBoolean()

Description

Extracts a function argument from a `jsval`, converts it to a Boolean value (if possible), and passes the converted value back to the caller.

Arguments

`JSContext *cx, jsval v, JSBool *bp`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *v* is the `jsval` from which the boolean is to be extracted.
- *bp* is a pointer to a `JSBool`. This function stores the converted value in `*bp`.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_ValueToObject()

Description

Extracts a function argument from a `jsval`, converts it to an object (if possible), and passes the converted value back to the caller. If the object is an array, use `JS_GetArrayLength()` and `JS_GetElement()` to read its contents.

Arguments

`JSContext *cx, jsval v, JSObject **op`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *v* is the `jsval` from which the object is to be extracted.
- *op* is a pointer to a (`JSObject *`). This function stores the converted value in `*op`.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_StringToValue()

Description

Stores a string return value in a `jsval`. It allocates a new JavaScript string object.

Arguments

`JSContext *cx, char *bytes, size_t sz, jsval *vp`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *bytes* is the string to be stored in the `jsval`. The string data is copied, so the caller should free the string when it is no longer needed. If the string size is not specified (see the *sz* argument), the string must be null-terminated.
- *sz* is the size of the string, in bytes. If *sz* is 0, the length of the null-terminated string is computed automatically.
- *vp* is a pointer to the `jsval` into which the contents of the string should be copied.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_DoubleToValue()

Description

Stores a floating-point number return value in a `jsval`.

Arguments

`JSContext *cx`, `double dv`, `jsval *vp`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *dv* is an 8-byte floating-point number.
- *vp* is a pointer to the `jsval` into which the contents of the double should be copied.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSVal JS_BooleanToValue()

Description

Stores a Boolean return value in a `jsval` structure.

Arguments

`JSBool bv`

Returns

A JSVal structure that contains the Boolean value that you passed to the function as an argument.

JSVal JS_IntegerToValue()

Description

Stores an integer return value in a `jsval`.

Arguments

`long lv`

Returns

A JSVal structure that contains the integer that you passed to the function as an argument.

JSVal JS_ObjectToValue()

Description

Stores an object return value in a `jsval`. Use `JS_ NewArrayObject()` to create an array object; use `JS_SetElement()` to define its contents.

Arguments

`JSObject *obj`

Returns

A JSVal structure that contains the object that you passed to the function as an argument.

char *JS_ObjectType()

Description

Given an object reference, `JS_ObjectType()` returns the class name of the object. For example, if the object is a DOM object, the function would return "`Document`". If the object is a node in the document, the function would return "`Element`". For an array object, the function would return "`Array`".

Note: Do not modify the returned buffer pointer or you might corrupt the data structures of the JavaScript interpreter.

Arguments

`JSObject *obj`

Typically, this argument is passed in and converted using `JS_ValueToObject()`.

Returns

A pointer to a null-terminated string. The caller should not free this string when it finishes.

JSObject *JS_NewArrayObject()

Description

Creates a new object that contains an array of `jsvals`.

Arguments

`JSContext *cx, unsigned int length, jsval *v`

- `cx` is the opaque `JSContext` pointer that passed to the JavaScript function.
- `length` is the number of elements that the array can hold.
- `v` is an optional pointer to the `jsvals` to be stored in the array. If the return value is not `null`, `v` is an array containing `length` elements. If the return value is `null`, the initial content of the array object is undefined (and can be set using `JS_SetElement()`).

Returns

A pointer to a new array object, or `null` upon failure.

long JS_GetArrayLength()

Description

Given a pointer to an array object, gets the number of elements in the array.

Arguments

`JSContext *cx, JSObject *obj`

- `cx` is the opaque `JSContext` pointer that passed to the JavaScript function.
- `obj` is a pointer to an array object.

Returns

The number of elements in the array or -1 upon failure.

JSBool JS_GetElement()

Description

Reads a single element of an array object.

Arguments

`JSContext *cx`, `JSObject *obj`, `unsigned int index`, `jsval *v`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *obj* is a pointer to an array object.
- *index* is an integer index into the array. The first element is index `0`, and the last element is index (`length - 1`).
- *v* is a pointer to a `jsval` where the contents of the `jsval` in the array should be copied.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_SetElement()

Description

Writes a single element of an array object.

Arguments

*JSContext *cx, JSObject *obj, unsigned int index, jsval *v*

- *cx* is the opaque `JSContext` pointer that was passed to the JavaScript function.
- *obj* is a pointer to an array object.
- *index* is an integer index into the array. The first element is index `0`, and the last element is index (`length - 1`).
- *v* is a pointer to a `jsval` whose contents should be copied to the `jsval` in the array.

Returns

A Boolean value that indicates success (`JS_TRUE`) or failure (`JS_FALSE`).

JSBool JS_ExecuteScript()

Description

Compiles and executes a JavaScript string. If the script generates a return value, it returns in `*rval`.

Arguments

`JSContext *cx`, `JSObject *obj`, *char *script*, *unsigned int sz*, `jsval *rval`

- *cx* is the opaque `JSContext` pointer that passed to the JavaScript function.
- *obj* is a pointer to the object in whose context the script executes. While the script is running, the `this` keyword is equal to this object. Usually this is the `JSObject` pointer that passed to the JavaScript function.
- *script* is a string that contains JavaScript code. If the string size is not specified (see the *sz* argument), the string must be null-terminated.

- *sz* is the size of the string, in bytes. If sz is 0, the length of the null-terminated string is computed automatically.
- *rval* is a pointer to a single jsval. The function's return value is stored in *rval.

Returns

A Boolean value that indicates success (JS_TRUE) or failure (JS_FALSE).

JSBool JS_ReportError()

Description

Describes the reason for a script error. Call this function before returning JS_FALSE to give the user information about why the script failed (for example, "wrong number of arguments").

Arguments

JSContext *cx, char *error, size_t sz

- *cx* is the opaque JSContext pointer that passed to the JavaScript function.
- *error* is a string that contains the error message. The string is copied, so the caller should free the string when it is no longer needed. If the string size is not specified (see the *sz* argument, below), the string must be null-terminated.
- *sz* is the size of the string, in bytes. If sz is 0, the length of the null-terminated string is computed automatically.

Returns

A Boolean value that indicates success (JS_TRUE) or failure (JS_FALSE).

File Access and Multiuser Configuration API

Macromedia recommends that you always use the File Access and Multiuser Configuration API to access the file system through C-level extensions. For files other than configuration files, the functions access the specified file or folder.

Dreamweaver MX supports multiple-user configurations for the multiple-user operating systems of Windows XP, Windows 2000, Windows NT, and Mac OS X.

Typically, you install Dreamweaver MX in a restricted folder such as C:\Program Folders in Windows. As a result, only users with Administrator privileges can make changes in the Dreamweaver MX configuration folder. To enable users on multiple-user operating systems to create and maintain individual configurations, Dreamweaver MX creates a separate configuration folder for each user. Any time Dreamweaver MX or a JavaScript extension writes to the configuration folder, Dreamweaver MX automatically writes to the user configuration folder instead. In this way, Dreamweaver MX lets each user customize the Dreamweaver MX configuration settings without disturbing the customized configurations of other users.

Dreamweaver MX creates the user configuration folder in a location where the user has full read and write access. The following table shows the specific location of the user configuration folder for each of the supported platforms:

Platform	User Configuration Folder
Macintosh OS X	MacHD:Users:*username*:Library:Application Support:Macromedia: ¬ Dreamweaver MX:Configuration
Windows 2000, Windows XP	C:\Documents and Settings*username*\Application Data\Macromedia\ ¬ Dreamweaver MX\Configuration
Windows NT	C:\WinNT\profiles*username*\Application Data\Macromedia\ ¬ Dreamweaver MX\Configuration

There are many cases where JavaScript extensions open files and write to the configuration folder. JavaScript extensions can access the file system by using DWFile, MMNotes, or passing a URL to `dw.getDocumentDOM()`. When an extension accesses the file system in a configuration folder, it generally uses `dw.getConfigurationPath()` and adds the filename, or it gets the path by accessing the `dom.URL` of an open document and adding the filename. An extension can also get the path by accessing the `dom.URL` and stripping the filename. The `dw.getConfigurationPath()` function and the `dom.URL` always return a URL in the Dreamweaver MX configuration folder, even if the document is located in the user configuration folder.

Any time a JavaScript extension opens a file in the Dreamweaver MX configuration folder Dreamweaver MX traps the access and checks the user configuration folder first. If a JavaScript extension saves data to disk in the Dreamweaver MX configuration folder through DWFile or MMNotes, Dreamweaver MX intercepts the call and redirects it to the user configuration folder.

For example, in Windows 2000 or Windows XP, if the user asks for "`file:///C|/Program Files/Macromedia/Dreamweaver MX/Configuration/Objects/Common/Table.htm`", Dreamweaver MX first looks to see if there is a file called file:///C|/Documents and Settings/username/Macromedia/Dreamweaver MX/Configuration/Objects/Common/Table.htm and, if it exists, uses it instead.

C-level extensions, or shared libraries, must use the File Access and Multiuser Configuration API to read and write to the configuration folder. Using the File Access and Multiuser Configuration API lets Dreamweaver read and write to the user configuration folder and ensures that the file operations do not fail due to insufficient access privileges. If your C-level extension accesses files in the configuration folder that were created through JavaScript with DWFile, MMNotes, or DOM manipulations, it is essential that you use the File Access and Multiuser Configuration API because these files might be located in the user configuration folder.

Note: Most JavaScript extensions will not need to be changed to write to the user Configuration folder. Only C shared libraries that write to the Configuration folder need to be updated to use the File Access and Multiuser Configuration API functions.

When you delete a file from Dreamweaver Configuration folder, Dreamweaver MX adds an entry to a mask file to indicate which files in the Configuration folder should not appear in the user interface. A masked file or folder will appear not to exist to Dreamweaver although it might physically exist in the folder.

For example, if you used the trash can in the Snippets panel to delete a Snippets folder called `javascript` and a file called `onepixelborder.csn`, Dreamweaver MX writes a file in the user configuration folder called `mm_deleted_files.xml`, which looks like the following example:

```
<?xml version = "1.0" encoding="utf-8" ?>
  <deleteditems>
  <item name="snippets/javascript/" />
  <item name="snippets/html/onepixelborder.csn" />
  </deleteditems>
```

As Dreamweaver MX populates the Snippets panel, it reads all the files in the user's Configuration/Snippets folder and all the files in the Dreamweaver MX Configuration/Snippets folder, except the Configuration/Snippets/javascript folder and the Configuration/Snippets/html/onepixelborder.csn file, and adds the resulting list of files to the Snippets panel list.

If a C-level extension calls the `MM_ConfigFileExists()` function for the URL `"file:///c|Program Files/macromedia/Dreamweaver MX/Configuration/Snippets/javascript/onepixelborder.csn"`, it returns `false`. Likewise, if a JavaScript extension tries to call `dw.getDocumentDom("file:///c|Program Files/Macromedia/Dreamweaver MX/Configuration/Snippets/javascript/onepixelborder.csn")`, it returns `Null`.

You can modify the mm_deleted_files.xml file to prevent Dreamweaver from showing files in the user interface, such as objects, canned content in the new dialog, and so on. You can call `MM_DeleteConfigfile()` to add file pathnames to mm_deleted_files.xml.

JS_Object MM_GetConfigFolderList()

Availability

Dreamweaver MX

Description

Gets a list of files, folders, or both for the specified folder. If you specify a configuration folder, the function gets a list of the folders that exists in both the user Configuration folder and the Dreamweaver Configuration folder, subject to filtering by mm_deleted_files.xml.

Arguments

`char *fileURL, char *constraints`

- *`char *fileUrl`* is a pointer to a string that names the folder for which you want a list of the contents. The string must have the format of a file URL (file://). The function accepts valid wildcard characters of asterisks (*) and question marks (?) in the file URL string. Use asterisks (*) to represent one or more unspecified characters, and question marks (?) to represent a single unspecified character.
- *`char *contstraints`* can be files or directories or `NULL`. If you specify `NULL`, `MM_GetConfigFolderList()` returns both files and directories.

Returns

`JSObject` is an array that contains the list of files or folders in either the user Configuration folder or the Dreamweaver Configuration folder, subject to filtering by the mm_deleted_files.xml file.

Examples

```
JSObject *jsobj_array;
jsobj_array = MM_GetConfigFolderList("file:///¬
  c|/Program Files/Macromedia/Dreamweaver MX/Configuration", "directories" );
```

JSBool MM_ConfigFileExists()

Availability

Dreamweaver MX

Description

Checks whether the specified file exists. If it is a file in a configuration folder, this function checks to see if the file exists in the user Configuration folder or the Dreamweaver MX Configuration folder. The function also checks to see if the filename is listed in the mm_deleted_files.xml file. If the name is listed in this file, the function returns `false`.

Arguments

`char *fileUrl` is a pointer to a string that names the file for which you are checking, which is provided in the format of a file URL (file:///).

Returns

`JSBool`

Example

```
char *dwConfig = "file:///c|/Program Files/Macromedia/Dreamweaver   MX/
  Configuration/Extensions.txt";
int fileno = 0;
if(MM_ConfigFileExists(dwConfig))
{
  fileno = MM_OpenConfigFile(dwConfig, "read");
}
```

int MM_OpenConfigFile()

Availability

Dreamweaver MX

Description

Opens the file and returns an operating system file handle. You can use the operating system file handle in calls to system file functions. You must close the file handle with a call to `_close`.

If the file is a configuration file, it finds the file in either the user Configuration folder or the Dreamweaver MX Configuration folder. If you open the configuration file for writing, the function creates the file in the user Configuration folder, even if it exists in the Dreamweaver MX Configuration folder.

Note: If you want to read the file before writing to it, open the file in "`read`" mode. When you want to write to the file, close the read handle and open the file again in "`write`" or "`append`" mode.

Arguments

`char *fileURL, char *mode`

`char *fileURL` is a pointer to a string that names the file that you are opening, which is provided as a file URL (file:///). If it specifies a path in the Dreamweaver MX Configuration folder, `MM_OpenConfigFile()` will resolve the path before opening the file.

`char *mode` is a string that specifies how you want to open the file. You can specify NULL, "`read`", "`write`", or "`append`" mode. If you specify "write" and the file does not exist, `MM_OpenconfigFile()` creates it. If you specify "`write`", `MM_OpenConfigFile()` opens the file with an exclusive share. If you specify "`read`", `MM_OpenConfigFile()` opens the file with a nonexclusive share.

If you open the file in "write" mode, any existing data in the file is truncated prior to writing new data. If you open the file in "append" mode, any data you write is appended to the end of the file.

Returns

An integer that is the operating system file handle for this file. Returns -1 if the file cannot be found or it does not exist.

Example

```
char *dwConfig = "file:///c|/Program Files/Macromedia/Dreamweaver MX/
  Configuration/Extensions.txt";
int = fileno;
if(MM_ConfigFileExists(dwConfig))
{
  fileno = MM_OpenConfigFile(dwConfig, "read");
}
```

JSBool MM_GetConfigFileAttributes()

Availability

Dreamweaver MX

Description

Finds the file and returns the attributes of the file. You can set any of the arguments except *fileURL* to NULL if you do not need the value.

Arguments

*char *fileURL, unsigned long *attrs, unsigned long *filesize, unsigned long *modtime, unsigned long *createtime*

*char *fileURL* is a pointer to a string that names the file for which you want the attributes, which is provided as a file URL (file:///). If *fileURL* specifies a path in the Dreamweaver MX Configuration folder, MM_GetConfigFileAttributes() resolves the path before opening the file.

*unsigned long *attrs* is the address of an integer that contains the returned attribute bits (see "JSBool MM_SetConfigFileAttributes()" on page 265 for available attributes).

*unsigned long *filesize* is the address of an integer that, on return, contains the file size in bytes.

*unsigned long *modtime* is the address of an integer that, on return, contains the time that the file was last modified. The time is given as the operating-system time value (same as DWFile::getModificationDate).

*unsigned long *createtime* is the address of an integer that, on return, contains the time that the file was created. The time is given as the operating-system time value (same as DWFile::getCreationDate).

Returns

JSBool

Returns JS_FALSE if the file does not exist or an error occurs in getting the attributes.

Example

```
char dwConfig = "file:///c|/Program Files/Macromedia/Dreamweaver MX/
  Configuration/Extensions.txt";
unsigned long attrs;
unsigned long filesize;
unsigned long modtime;
unsigned long createtime;
MM_GetConfigAttributes(dwConfig, &attrs, &filesize, &modtime, &createtime);
```

JSBool MM_SetConfigFileAttributes()

Availability

Dreamweaver MX

Description

Sets the attributes that you specify for the file, if they are different from the current attributes.

If the specified file URL is in the Dreamweaver MX Configuration folder, this function first copies the file to the user Configuration folder before it sets the attributes. If the attributes are the same as the current file attributes, the file is not copied.

Arguments

*char *fileURL, unsigned long attrs*

*char *fileURL* is a pointer to a string that names the file for which you want to set the attributes, which is provided as a file URL (file://).

unsigned long attrs specifies the attribute bits to set on the file. You can use a logical OR on the following constants to set the attributes:

```
MM_FILEATTR_NORMAL
MM_FILEATTR_RDONLY
MM_FILEATTR_HIDDEN
MM_FILEATTR_SYSTEM
MM_FILEATTR_SUBDIR
```

Returns

JSBool

Returns JS_TRUE if there is no error. If the file does not exist or is masked for deletion, the function returns JS_FALSE.

Example

```
char *dwConfig = "file:///c|/Program Files/Macromedia/Dreamweaver   MX/
  Configuration/Extensions.txt";
unsigned long attrs;
attrs = (MM_FILEATTR_NORMAL | MM_FILEATTR_RDONLY);
int fileno = 0;
if(MM_SetConfigFileAttrs(dwConfig, attrs))
{
  fileno = MM_OpenConfigFile(dwConfig);
}
```

JSBool MM_CreateConfigFolder()

Availability

Dreamweaver MX

Description

Creates a folder in the specified location.

If *fileURL* specifies a folder below the Dreamweaver MX Configuration folder, the function creates the folder in the user Configuration folder. If *fileURL* does not specify a folder below the Dreamweaver MX Configuration folder, the function creates the specified folder, including all higher-level folders in the path if they do not already exist.

Arguments

*char *fileURL* is a pointer to a file URL string (file:///) that names the configuration folder that you want to create.

Returns

JSBool

Example

```
char *dwConfig = "file:///c|/Program Files\Macromedia\Dreamweaver
  MX\Configuration\Extensions.txt";
MM_CreateConfigFolder(dwConfig);
```

JSBool MM_RemoveConfigFolder()

Availability

Dreamweaver MX

Description

Removes the folder and its files and subfolders. If the folder is in the Dreamweaver MX Configuration folder, it masks the folder for deletion in mm_deleted_files.xml.

Arguments

*char *fileURL* is a pointer to a string that names the folder to remove, which is provided as a file URL (file:///).

Returns

JSBool

Example

```
char *dwConfig = "file:///c|/Program Files\Macromedia\Dreamweaver
  MX\Configuration\Objects";
MM_RemoveConfigFolder(dwConfig);
```

JSBool MM_DeleteConfigFile()

Availability

Dreamweaver MX

Description

Deletes the file, if it exists. If the file exists in the Dreamweaver MX Configuration folder, the function masks the file for deletion in mm_deleted_files.xml.

If *fileURL* is not in the Dreamweaver MX Configuration folder, the function deletes the specified file.

Arguments

*char *fileURL* is a pointer to a string that names the configuration folder to remove, which is provided as a file URL (file:///).

Returns

JSBool

Example

```
char dwConfig = "file:///c:|Program Files\Macromedia\Dreamweaver
  MX\Configuration\Objects\insertbar.xml";
MM_DeleteConfigFile(dwConfig);
```

Calling a C function from JavaScript

After you understand how C-level extensibility works in Dreamweaver and its dependency on certain data types and functions, it's useful to know how to build a library and call a function.

This example requires four files, which are included in the Extending/c_files folder inside the Dreamweaver application folder:

- mm_jsapi.h is a header file that includes definitions for the data types and functions that are described in "C-level extensibility and the JavaScript interpreter" on page 253.
- mm_jsapi_environment.h, which defines the MM_Environment.h structure.
- Sample.c is an example file that defines the `computeSum()` function.
- Sample.mak is a makefile that you can use to build Sample.c into a DLL with Microsoft Visual C++; Sample.proj is the equivalent file for building a CFM Library with Metrowerks CodeWarrior. If you use another tool, you can create the makefile.

To build the DLL in Windows:

1. In Microsoft Visual C++, choose File > Open Workspace and select Sample.mak.
2. Choose Build > Rebuild All.

   When the build operation finishes, a file called Sample.dll appears in the folder that contains Sample.mak (or one of its subfolders).

To build the shared library on the Macintosh:

1. Open Sample.proj in Metrowerks CodeWarrior.
2. Build the project to generate a CFM Library.

   When the build operation finishes, a file called Sample appears in the folder that contains Sample.proj (or in one of its subfolders).

To call the computeSum() function from the Insert Horizontal Rule object:

1. Create a folder called JSExtensions in the Configuration folder within the Dreamweaver application folder.
2. Copy Sample.dll (Windows) or Sample (Macintosh) to the JSExtensions folder.
3. In a text editor, open the file called horizontal_rule.htm in the Configuration/Objects/Common folder.

4 Add the line `alert(Sample.computeSum(2,2));` to the `objectTag()` function so that it appears as shown in the following example:

```
function objectTag() {
  // Return the html tag that should be inserted
  alert(Sample.computeSum(2,2));
  return "<HR>";
}
```

5 Save the file and restart Dreamweaver.

To execute the computeSum() function:

Choose Insert > Horizontal Rule.

A dialog box that contains the number 4 (the result of computing the sum of 2 plus 2) appears.

Part III
Utility APIs

Understand the Dreamweaver utility functions that you can use to acccess local and web-based files, work with Fireworks and Flash objects, manage database connections, create new database connection types, access JavaBeans components, and integrate Dreamweaver with various source control systems.

- Chapter 22, “The File I/O API”
- Chapter 23, “The HTTP API”
- Chapter 24, “The Design Notes API”
- Chapter 25, “The Fireworks Integration API”
- Chapter 26, “The Flash Objects API”
- Chapter 27, “The Database API”
- Chapter 28, “The Database Connectivity API”
- Chapter 29, “The JavaBeans API”
- Chapter 30, “The Source Control Integration API”

CHAPTER 22
The File I/O API

Macromedia Dreamweaver MX includes a C shared library called DWfile that gives authors ofobjects, commands, behaviors, data translators, floating panels, and Property inspectors the ability to read and write files on the local file system. This chapter describes the File I/O API and how to use it.

For general information on how C libraries interact with the JavaScript interpreter in Dreamweaver, see "C-Level Extensibility" on page 251.

Accessing configuration folders

On Microsoft Windows NT, Windows 2000, and Windows XP, and on Mac OS X platforms, users have their own copies of configuration files. Whenever Dreamweaver MX writes to a configuration file, Dreamweaver writes it to the user's Configuration folder. Similarly, when Dreamweaver reads a configuration file, Dreamweaver looks for it first in the user's Configuration folder, then in the application's Configuration folder. DWFile functions use the same mechanism. In other words, if your extension reads or writes a file in the Configuration folder, your extension accesses the user's Configuration folder too. For more information about Configuration folders on multiuser platforms, see "Extension folders" on page 18.

The File I/O API

All functions in the File I/O API are methods of the `DWfile` object. Optional arguments are enclosed in braces ({}). Functions with an availability of 2 were included in the version of DWfile that was supplied as a download for Dreamweaver 2 from the Macromedia website. This version of DWfile might have been installed with third-party objects.

DWfile.copy()

Availability

Dreamweaver 3

Description

Copies the specified file to a new location.

Arguments

`originalURL`, *`copyURL`*

- The first argument is the file you want to copy, which is expressed as a file:// URL.
- The second argument is the location where you want to save the copied file, which is expressed as a file:// URL.

Returns

`true` if the copy succeeds; `false` otherwise.

Example

The following code copies a file called myconfig.cfg to myconfig_backup.cfg.

```
var fileURL = "file:///c|/Config/myconfig.cfg";
var newURL ="file:///c|/Config/myconfig_backup.cfg";
DWfile.copy(fileURL, newURL);
```

DWfile.createFolder()

Availability

Dreamweaver 2

Description

Creates a folder (directory) at the specified location.

Arguments

folderURL

The argument is the location of the folder you want to create, which is expressed as a file:// URL.

Returns

`true` if the folder is successfully created; `false` otherwise.

Example

The following code tries to create a folder called tempFolder at the top level of the C drive and displays an alert box that indicates whether the operation is successful.

```
var folderURL = "file:///c|/tempFolder";
if (DWfile.createFolder(folderURL)){
  alert("Created " + folderURL);
}else{
  alert("Unable to create " + folderURL);
}
```

DWfile.exists()

Availability

Dreamweaver 2

Description

Tests for the existence of the specified file.

Arguments

fileURL

The argument is the requested file, which is expressed as a file:// URL.

Returns

`true` if the file exists; `false` otherwise.

Example

The following code checks for a file called mydata.txt and displays an alert box that tells the user whether the file exists.

```
var fileURL = "file:///c|/temp/mydata.txt";
if (DWfile.exists(fileURL)){
  alert( fileURL + " exists!");
}else{
  alert( fileURL + " does not exist.");
}
```

DWfile.getAttributes()

Availability

Dreamweaver 2

Description

Gets the attributes of the specified file or folder.

Arguments

fileURL

The argument is the file or folder for which you want to get attributes, which is expressed as a file:// URL.

Returns

A string that represents the attributes of the specified file or folder. If the file or folder does not exist, this function returns a `null` value. The following characters in the string represent the attributes:

- `R` is read only.
- `D` is folder (directory).
- `H` is hidden.
- `S` is system file or folder.

Example

The following code gets the attributes of the mydata.txt file and displays an alert box if the file is read only.

```
var fileURL = "file:///c|/temp/mydata.txt";
var str = DWfile.getAttributes(fileURL);
if (str && (str.indexOf("R") != -1)){
  alert(fileURL + " is read only!");
}
```

DWfile.getModificationDate()

Availability

Dreamweaver 2

Description

Gets the time when the file was last modified.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file for which you are checking the last modified time.

Returns

A string that contains a hexadecimal number that represents the number of time units that have elapsed since some base time. The exact meaning of time units and base time is platform-dependent; in Windows, for example, a time unit is 100ns, and the base time is January 1st, 1600.

Example

It's useful to call the function twice and compare the return values because the value that this function returns is platform-dependent and is not a recognizable date and time. For example, the following code gets the modification dates of file1.txt and file2.txt and displays an alert box that indicates which file is newer.

```
var file1 = "file:///c|/temp/file1.txt";
var file2 = "file:///c|/temp/file2.txt";
var time1 = DWfile.getModificationDate(file1);
var time2 = DWfile.getModificationDate(file2);
if (time1 == time2){
  alert("file1 and file2 were saved at the same time");
}else if (time1 < time2){
  alert("file1 older that file2");
}else{
  alert("file1 is newer than file2");
}
```

DWfile.getCreationDate()

Availability

Dreamweaver 4

Description

Gets the time that the file was created.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file for which you are checking the creation time.

Returns

A string that contains a hexadecimal number that represents the number of time units that have elapsed since some base time. The exact meaning of time units and base time is platform-dependent; in Windows, for example, a time unit is 100ns, and the base time is January 1st, 1600.

Example

You can call this function and the `DWfile.getModificationDate()` function on a file to compare the modification date to the creation date.

```
var file1 = "file:///c|/temp/file1.txt";
var time1 = DWfile.getCreationDate(file1);
var time2 = DWfile.getModificationDate(file1);
if (time1 == time2){
  alert("file1 has not been modified since it was created");
}else if (time1 < time2){
  alert("file1 was last modified on " + time2);
}
```

DWfile.getCreationDateObj()

Availability

Dreamweaver MX

Description

Gets the JavaScript object that represents the time when the file was created.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file for which you are checking the creation time.

Returns

A JavaScript Date object that represents the date and time when the specified file was created.

DWfile.getModificationDateObj()

Availability

Dreamweaver MX

Description

Gets the JavaScript object that represents the time when the file was last modified.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file for which you are checking the time of the most recent modification.

Returns

A JavaScript Date object that represents the date and time when the specified file was last modified.

DWfile.getSize()

Availability

Dreamweaver MX

Description

Gets the size of a specified file.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file for which you are checking the size.

Returns

An integer that represents the actual size, in bytes, of the specified file.

DWfile.listFolder()

Availability

Dreamweaver 2

Description

Gets a list of the contents of the specified folder.

Arguments

folderURL [,constraint]

- The first argument is the folder for which you want a contents list, which is expressed as a file:// URL, plus an optional wildcard file mask. Valid wildcards are asterisks (*), which match 1 or more characters and question marks (?), which match a single character.
- The second argument, if supplied, must be either `"files"` (return only files) or `"directories"` (return only directories). If it is omitted, the function returns files and directories.

Returns

An array of strings that represents the contents of the folder.

Example

The following code gets a list of all the text (.txt) files in the temp folder and displays the list in an alert box.

```
var folderURL = "file:///c|/temp";
var fileMask = "*.txt";
var list = DWfile.listFolder(folderURL + "/" + fileMask, "files");
if (list){
  alert(folderURL + " contains: " + list.join("\n"));
}
```

DWfile.read()

Availability

Dreamweaver 2

Description

Reads the contents of the specified file into a string.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file you want to read.

Returns

A string that contains the contents of the file, or `null` if the read fails.

Example

The following code reads the file mydata.txt and, if successful, displays an alert box with the contents of the file.

```
var fileURL = "file:///c|/temp/mydata.txt";
var str = DWfile.read( fileURL);
if (str){
  alert( fileURL + " contains: " + str);
}
```

DWfile.remove()

Availability

Dreamweaver 3

Description

Moves the specified file to the Recycling Bin or Trash.

Arguments

fileURL

The argument, which is expressed as a file:// URL, is the file you want to remove.

Returns

`true` if the operation succeeds; `false` otherwise.

Example

The following example uses `DWfile.getAttributes()` to determine whether the file is read-only and `confirm()` to display a Yes/No dialog box to the user.

```
function deleteFile(){
  var delAnyway = false;
  var selIndex = document.theForm.menu.selectedIndex;

  var selFile = document.theForm.menu.options[selIndex].value;
  if (DWfile.getAttributes(selFile).indexOf('R') != -1){
    delAnyway = confirm('This file is read-only. Delete anyway?');
    if (delAnyway){
      DWfile.remove(selFile);
    }
  }
}
```

DWfile.setAttributes()

Availability

Dreamweaver MX

Description

Sets the system-level attributes of a particular file.

Arguments

fileURL, *strAttrs*

- *fileURL* identifies the file, which is expressed as a file:// URL, for which you are setting the attributes.
- *strAttrs* specifies the system-level attributes for the file that is identified by *fileURL*. The following table describes valid attribute values and their meaning.

Attribute Value	Description
R	Read only
W	Writable (overrides R)
H	Hidden
V	Visible (overrides H)

Acceptable values for the strAttrs string are R, W, H, V, RH, RV, WH, or WV.

You should not use R and W together because they are mutually exclusive. If you combine them, R becomes meaningless, and the file is set as writable (W). You should not use H and V together because they are also mutually exclusive. If you combine them, H becomes meaningless, and the file is set as visible (V).

If you specify H or V without specifying an R or W read/write attribute, the existing read/write attribute for the file is not changed. Likewise, if you specify R or W, without specifying an H or V visibility attribute, the existing visibility attribute for the file is not changed.

Returns

Nothing.

DWfile.write()

Availability

Dreamweaver 2

Description

Writes the specified string to the specified file. If the specified file does not yet exist, it is created.

Arguments

fileURL, *text* [, *mode*]

- The first argument, which is expressed as a file:// URL, is the file to which you are writing.
- The second argument is the string to be written.
- The third argument, if supplied, must be "append". If this argument is omitted, the contents of the file are overwritten by the string.

Returns

`true` if the string is successfully written to the file, `false` otherwise.

Example

The following code attempts to write the string `"xxx"` to the mydata.txt file and displays an alert if the write succeeds. It then tries to append the string `"aaa"` to the file and displays a second alert if the write succeeds. After executing this script, the mydata.txt file contains the text `xxxaaa` and nothing else.

```
var fileURL = "file:///c|/temp/mydata.txt";
if (DWfile.write(fileURL, "xxx")){
  alert("Wrote xxx to " + fileURL);
}
if (DWfile.write(fileURL, "aaa", "append")){
  alert("Appended aaa to " + fileURL);
}
```

CHAPTER 23
The HTTP API

Extensions are not limited to working within the local file system. Macromedia Dreamweaver MX provides a mechanism to get information from and send information to a web server through use of hypertext transfer protocol (HTTP). This chapter describes the HTTP API and how to use it.

The HTTP API

All functions in the HTTP API are methods of the `MMHttp` object. Most of these functions take a URL as an argument, and most return an object. The default port for URL arguments is 80. To specify a port other than 80, append a colon and the port number to the URL, as shown in the following example:

```
MMHttp.getText("http://www.myserver.com:8025");
```

For functions that return an object, the object has two properties: `statusCode` and `data`.

`statusCode` indicates the status of the operation; possible values include, but are not limited to, the following values:

- 200: Status OK
- 400: Unintelligible request
- 404: Requested URL not found
- 405: Server does not support requested method
- 500: Unknown server error
- 503: Server capacity reached

For a comprehensive list of status codes for your server, check with your Internet service provider or system administrator.

The value of the `data` property varies according to the function; possible values are specified in the individual function listings.

Functions that return an object also have a callback version. Callback functions let other functions execute while the web server processes an HTTP request. This is useful if you are making multiple HTTP requests from Dreamweaver. The callback version of a function passes its ID and return value directly to the function that is specified as its first argument.

Optional arguments are enclosed in braces ({ }).

MMHttp.clearTemp()

Description

Deletes all the files in the Configuration/Temp folder, which is located inside the Dreamweaver application folder.

Arguments

None.

Returns

Nothing.

Example

The following code, when saved in a file inside the Configuration/Shutdown folder, removes all the files from the Configuration/Temp folder when the user quits Dreamweaver:

```
<html>
<head>
<title>Clean Up Temp Files on Shutdown</title>
</head>
<body onLoad="MMHttp.clearTemp()">
</body>
</html>
```

MMHttp.getFile()

Description

Gets the file at the specified URL and saves it in the Configuration/Temp folder, which is located inside the Dreamweaver application folder. Dreamweaver automatically creates subfolders that mimic the folder structure of the server; for example, if the specified file is at http://www.dreamcentral.com/people/index.html, Dreamweaver stores the index.html file in the People folder inside the www.dreamcentral.com folder.

Arguments

URL {,prompt} {,saveURL} {,titleBarLabel}

- *URL* is an absolute URL on a web server; if http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.
- *prompt* is a Boolean value that specifies whether to prompt the user to save the file. If *saveURL* is outside the Configuration/Temp folder, a *prompt* value of `false` is ignored for security reasons.
- *saveURL* is the location on the user's hard disk where the file should be saved, which is expressed as a file:// URL. If prompt is `true` or *saveURL* is outside the Configuration/Temp folder, the user can override *saveURL* in the Save dialog box.
- *titleBarLabel* is the label that should appear in the title bar of the Save dialog box.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the location where the file is saved, which is expressed as a file:// URL. Normally the `statusCode` property of the object contains the status code that is received from the server. However, if a disk error occurs while Dreamweaver is saving the file on the local drive, the `statusCode` property contains an integer that represents one of the following error codes if the operation is not successful:

- 1: Unspecified error
- 2: File not found
- 3: Invalid path
- 4: Number of open files limit reached
- 5: Access denied
- 6: Invalid file handle
- 7: Cannot remove current working directory
- 8: No more directory entries
- 9: Error setting file pointer
- 10: Hardware error
- 11: Sharing violation
- 12: Lock violation
- 13: Disk full
- 14: End of file reached

Example

The following code gets an HTML file, saves all the files in the Configuration/Temp folder, and then opens the local copy of the HTML file in a browser:

```
var httpReply = MMHttp.getFile("http://www.dreamcentral.com/¬
people/profiles/scott.html",
false);
if (httpReply.statusCode == 200){
  var saveLoc = httpReply.data;
  dw.browseDocument(saveLoc);
}
```

MMHttp.getFileCallback()

Description

Gets the file at the specified URL, saves it in the Configuration/Temp folder inside the Dreamweaver application folder, and then calls the specified function with the request ID and reply result. When saving the file locally, Dreamweaver automatically creates subfolders that mimic the directory structure of the server; for example, if the specified file is at http://www.dreamcentral.com/people/index.html, Dreamweaver stores the index.html file in the People folder inside the www.dreamcentral.com folder.

Arguments

callbackFunction, *URL* {,*prompt*} {,*saveURL*} {,*titleBarLabel*}

- *callbackFunction* is the name of the JavaScript function to call when the HTTP request is complete.
- *URL* is an absolute URL on a web server; if http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.
- *prompt* is a Boolean value that specifies whether to prompt the user to save the file. If *saveURL* is outside the Configuration/Temp folder, a *prompt* value of `false` is ignored for security reasons.
- *saveURL* is the location on the user's hard disk where the file should be saved, which is expressed as a file:// URL. If prompt is `true` or *saveURL* is outside the Configuration/Temp folder, the user can override *saveURL* in the Save dialog box.
- *titleBarLabel* is the label that should appear in the title bar of the Save dialog box.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the location where the file was saved, which is expressed as a file:// URL. Normally the `statusCode` property of the object contains the status code that is received from the server. However, if a disk error occurs while Dreamweaver is saving the file on the local drive, the `statusCode` property contains an integer that represents an error code. See "MMHttp.getFile()" on page 282 for a list of possible error codes.

MMHttp.getText()

Description

Retrieves the contents of the document at the specified URL.

Arguments

URL

URL is an absolute URL on a web server. If http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the contents of the document.

Example

The following code gets the contents of a file on a web server and puts it in a new, untitled Dreamweaver document:

```
var httpReply = MMHttp.getText("http://www.dreamcentral.com/¬
people/profiles/lori.html");
if (httpReply.statusCode == 200){
  var newDoc =  dw.createDocument();
  newDoc.documentElement.outerHTML = httpReply.data;
}
```

MMHttp.getTextCallback()

Description

Retrieves the contents of the document at the specified URL and passes it to the specified function.

Arguments

callbackFunc, *URL*

- *callbackFunc* is the name of the JavaScript function to call when the HTTP request is complete.
- *URL* is an absolute URL on a web server; if http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the contents of the document.

Example

The following code populates a form field with the text that the `MMHttp.GetTextCallback()` function returns, or it shows an error message if the function returns an error:

```
var requestID = MMHttp.getTextCallback("httpCallback", ¬
"www.dreamcentral.com/index.html")

function httpCallback(requestID,reply) {
  if (reply.statusCode == 200) {
    document.theForm.docContents.value = reply.data;
  }else{
    alert("Request #: " + requestID + "returned the following ¬
    error: " + reply.statusCode);
  }
}
```

MMHttp.postText()

Description

Uses an HTTP `post` request to pass the specified data to the specified URL. Typically the data that is associated with a post operation is form-encoded text, but it can be any type of data that the server expects to receive.

Arguments

`URL`, *`dataToPost {`*,*`contentType}`*

- *`URL`* is an absolute URL on a web server; if http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.
- *`dataToPost`* is the data to be posted. If the third argument is `"application/x-www-form-urlencoded"` or is omitted, *`dataToPost`* must be form-encoded, according to section 8.2.1 of the RFC 1866 specification (available at http://www.faqs.org/rfcs/rfc1866.html).
- *`contentType`* is the content type of the data to be posted. If omitted, this argument defaults to `"application/x-www-form-urlencoded"`.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the data that results from the post operation.

MMHttp.postTextCallback()

Description

Performs an HTTP post of the text to the specified URL and passes the reply from the server to the specified function. Typically the data associated with a post operation is form-encoded text, but it can be any type of data that the server expects to receive.

Arguments

`callbackFunc`, *`URL`*, *`dataToPost {`*,*`contentType}`*

- *`callbackFunc`* is the name of the JavaScript function to call when the HTTP request is complete.
- *`URL`* is an absolute URL on a web server; if http:// is omitted from the URL, Dreamweaver MX assumes HTTP protocol.
- *`dataToPost`* is the data to be posted. If the third argument is `"application/x-www-form-urlencoded"` or is omitted, *`data`* must be form-encoded, according to section 8.2.1 of the RFC 1866 specification (available at http://www.faqs.org/rfcs/rfc1866.html).
- *`contentType`* is the content type of the data to be posted. If omitted, this argument defaults to `"application/x-www-form-urlencoded"`.

Returns

An object that represents the reply from the server. The `data` property of this object is a string that contains the data that results from the post operation.

CHAPTER 24
The Design Notes API

Macromedia Dreamweaver, Fireworks, and Flash MX give web designers and developers a way to store and retrieve extra information about documents—information such as review comments, change notes, or the source file for a GIF or JPEG—in files that are called Design Notes.

MMNotes is a C shared library that lets extensions authors read and write Design Notes files. As with the DWfile shared library, MMNotes has a JavaScript API that makes it possible to call the functions that are contained in the library from objects, commands, behaviors, floating panels, Property inspectors, and data translators.

MMNotes also has a C API that lets other applications read and write Design Notes files. The MMNotes shared library can be used independently, even if Dreamweaver is not installed.

For more information about using the Design Notes feature from within Dreamweaver, see *Using Dreamweaver*.

How Design Notes work

Each Design Notes file stores information for a single document. If one or more documents in a folder has a Design Notes file associated with it, Dreamweaver creates a _notes subfolder where Design Notes files can be stored. The _notes folder and the Design Notes files that it contains are not visible in the Site panel, but they appear in the Finder (Macintosh) or Windows Explorer. A Design Notes filename comprises the main filename plus the extension .mno. For example, the Design Notes file that is associated with avocado8.gif is avocado8.gif.mno.

Design Notes files are XML files that store information in a series of key/value pairs. The key describes the type of information that is being stored, and the value represents the information itself. Keys are limited to 64 characters.

The following example shows the Design Notes file for foghorn.gif:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<info>
  <infoitem key="FW_source" value="file:///C|sites/¬
  dreamcentral/images/sourceFiles/foghorn.png" />
  <infoitem key="Author" value="Heidi B." />
  <infoitem key="Status" value="Final draft, approved ¬
  by Jay L." />
</info>
```

The Design Notes JavaScript API

All functions in the Design Notes JavaScript API are methods of the `MMNotes` object. Optional arguments are enclosed in braces ({ }).

MMNotes.close()

Description

Closes the specified Design Notes file and saves any changes. If all the key/value pairs are removed, Dreamweaver deletes the Design Notes file. If it is the last Design Notes file in the _notes folder, Dreamweaver also deletes the folder.

Note: Always call MMNotes.close() when you finish with Design Notes to cause Dreamweaver to write to the file.

Arguments

`fileHandle`

The argument is the file handle that `MMNotes.open()` returns.

Returns

Nothing.

Example

See "MMNotes.set()" on page 292.

MMNotes.filePathToLocalURL()

Description

Converts the specified local drive path to a file:// URL.

Arguments

`drivePath`

The argument is a string that contains the full drive path.

Returns

A string that contains the file:// URL for the specified file.

Example

A call to `MMNotes.filePathToLocalURL('C:\sites\webdev\index.htm')` returns `"file:///c|/sites/webdev/index.htm"`.

MMNotes.get()

Description

Gets the value of the specified key in the specified Design Notes file.

Arguments

`fileHandle`, *`keyName`*

- The first argument is the file handle that `MMNotes.open()` returns.
- The second argument is a string that contains the name of the key.

Returns

A string that contains the value of the key.

Example

See "MMNotes.getKeys()" on page 289.

MMNotes.getKeyCount()

Description

Gets the number of key/value pairs in the specified Design Notes file.

Arguments

fileHandle

The argument is the file handle that `MMNotes.open()` returns.

Returns

An integer that represents the number of key/value pairs in the Design Notes file.

MMNotes.getKeys()

Description

Gets a list of all the keys in a Design Notes file.

Arguments

fileHandle

The argument is the file handle that `MMNotes.open()` returns.

Returns

An array of strings where each string contains the name of a key.

Example

The following code might be used in a custom floating panel to display the Design Notes information for the active document:

```
var noteHandle = MMNotes.open(dw.getDocumentDOM().URL);
var theKeys = MMNotes.getKeys(noteHandle);
var noteString = "";
var theValue = "";
for (var i=0; i < theKeys.length; i++){
  theValue = MMNotes.get(noteHandle,theKeys[i]);
  noteString +=  theKeys[i] + " = " theValue + "\n";
}
document.theForm.bigTextField.value = noteString;
// always close noteHandle
MMNotes.close(noteHandle);
```

MMNotes.getSiteRootForFile()

Description

Determines the site root for the specified Design Notes file.

Arguments

fileURL

The argument is the path to a local file, which is expressed as a file:// URL.

Returns

A string that contains the path of the Local Root folder for the site, which is expressed as a file:// URL, or an empty string if Dreamweaver is not installed or the Design Notes file is outside any site that is defined with Dreamweaver. This function searches for all the sites that are defined in Dreamweaver versions 3, 4, and MX as well as UltraDev versions 1 and 4.

MMNotes.getVersionName()

Description

Gets the version name of the MMNotes shared library, which indicates the application that implemented it.

Arguments

None.

Returns

A string that contains the name of the application that implemented the MMNotes shared library.

Example

Calling `MMNotes.getVersionName()` from a Dreamweaver command, object, behavior, Property inspector, floating panel, or data translator returns `"Dreamweaver"`. Calling `MMNotes.getVersionName()` from Fireworks also returns `"Dreamweaver"` because Fireworks uses the same version of the library, which was created by the Dreamweaver engineering team.

MMNotes.getVersionNum()

Description

Gets the version number of the MMNotes shared library.

Arguments

None.

Returns

A string that contains the version number.

MMNotes.localURLToFilePath()

Description

Converts the specified file:// URL to a local drive path.

Arguments

fileURL

The argument is the path to a local file, which is expressed as a file:// URL.

Returns

A string that contains the local drive path for the specified file.

Example

A call to `MMNotes.localURLToFilePath('file:///MacintoshHD/images/moon.gif')` returns `"MacintoshHD:images:moon.gif"`.

MMNotes.open()

Description

Opens the Design Notes file that is associated with the specified file or creates one if none exists.

Arguments

`filePath`, *`{bForceCreate}`*

- The first argument is the path to the main file with which the Design Notes file is associated, which is expressed as a file:// URL.
- The second argument is a Boolean value that indicates whether to create the Note even if Design Notes is turned off for the site or if *`filePath`* is not associated with any site.

Returns

The file handle for the Design Notes file or zero if the file was not opened or created.

Example

See "MMNotes.set()" on page 292.

MMNotes.remove()

Description

Removes the specified key (and its value) from the specified Design Notes file.

Arguments

`fileHandle`, *`keyName`*

- The first argument is the file handle that `MMNotes.open()` returns.
- The second argument is a string that contains the name of the key to be removed.

Returns

A Boolean value that indicates whether the operation is successful.

MMNotes.set()

Description

Creates or updates one key/value pair in a Design Notes file.

Arguments

fileHandle, *keyName*, *valueString*

- The first argument is the file handle that `MMNotes.open()` returns.
- The second argument is a string that contains the name of the key.
- The third argument is a string that contains the value.

Returns

A Boolean value that indicates whether the operation is successful.

Example

The following code opens the Design Notes file that is associated with a file in the dreamcentral site called peakhike99/index.html, adds a new key/value pair, changes the value of an existing key, and then closes the Note file.

```
var noteHandle = MMNotes.open('file:///c|/sites/dreamcentral/
peakhike99/index.html',true);
if(noteHandle > 0){
  MMNotes.set(noteHandle,"Author","M. G. Miller");
  MMNotes.set(noteHandle,"Last Changed","August 28, 1999");
  MMNotes.close(noteHandle);
}
```

The Design Notes C API

In addition to the JavaScript API, the MMNotes shared library also exposes a C API that lets other applications create Design Notes files. It is not necessary to call these C functions directly if you use the MMNotes shared library in Dreamweaver because the JavaScript versions of the functions call them.

This section contains descriptions of the functions, their arguments, and their return values. You can find definitions for the functions and data types in the MMInfo.h file in the Extending/c_files folder inside the Dreamweaver application folder.

Optional arguments are enclosed in braces ({ }).

void CloseNotesFile()

Description

Closes the specified Design Notes file and saves any changes. If all key/value pairs are removed from the Note file, Dreamweaver deletes it. Deletes the _notes folder when the last Design Notes file is deleted.

Arguments

`FileHandle` *noteHandle*

The argument is the file handle that `OpenNotesFile()` returns.

Returns

Nothing.

BOOL FilePathToLocalURL()

Description

Converts the specified local drive path to a file:// URL.

Arguments

`const char* drivePath, char* localURLBuf, int localURLMaxLen`

- The first argument is a string that contains the full drive path.
- The second argument is the buffer where the file:// URL should be stored.
- The third argument is the maximum size of `localURLBuf`.

Returns

A Boolean value that indicates whether the operation is successful; stores the file:// URL in `localURLBuf`.

BOOL GetNote()

Description

Gets the value of the specified key in the specified Design Notes file.

Arguments

`FileHandle noteHandle, const char keyName[64], char* valueBuf, int valueBufLength`

- The first argument is the file handle that `OpenNotesFile()` returns.
- The second argument is a string that contains the name of the key.
- The third argument is the buffer where the value should be stored.
- The fourth argument is the integer that `GetNoteLength(noteHandle, keyName)` returns, which indicates the maximum length of the value buffer.

Returns

A Boolean value that indicates whether the operation is successful; stores the value of the key in `valueBuf`.

Example

The following code gets the value of the `comments` key in the Design Notes file that is associated with welcome.html:

```
FileHandle noteHandle = OpenNotesFile("file:///c|/sites/avocado8/¬
iwjs/welcome.html");
if(noteHandle > 0){
  int valueLength = GetNoteLength( noteHandle, "comments");
  char* valueBuffer = new char[valueLength + 1];
  GetNote(noteHandle, "comments", valueBuffer, valueLength + 1);
  printf("Comments: %s",valueBuffer);
  CloseNotesFile(noteHandle);
}
```

int GetNoteLength()

Description

Gets the length of the value associated with the specified key.

Arguments

`FileHandle noteHandle, const char keyName[64]`

- The first argument is the file handle that `OpenNotesFile()` returns.
- The second argument is a string that contains the name of the key.

Returns

An integer that represents the length of the value.

Example

See "BOOL GetNote()" on page 293.

int GetNotesKeyCount()

Description

Gets the number of key/value pairs in the specified Design Notes file.

Arguments

`FileHandle noteHandle`

The argument is the file handle that `OpenNotesFile()` returns.

Returns

An integer that represents the number of key/value pairs in the Design Notes file.

BOOL GetNotesKeys()

Description

Gets a list of all the keys in a Design Notes file.

Arguments

`FileHandle noteHandle, char* keyBufArray[64], int keyArrayMaxLen`

- The first argument is the file handle that `OpenNotesFile()` returns.
- The second argument is the buffer array where the keys should be stored.
- The third argument is the integer that `GetNotesKeyCount(noteHandle)` returns, indicating the maximum number of items in the key buffer array.

Returns

A Boolean value that indicates whether the operation is successful; stores the key names in *keyBufArray*.

Example

The following code prints the key names and values of all the keys in the Design Notes file that are associated with welcome.html:

```
typedef char[64] InfoKey;
FileHandle noteHandle = OpenNotesFile("file:///c|/sites/avocado8/¬
iwjs/welcome.html");
if (noteHandle > 0){
  int keyCount = GetNotesKeyCount(noteHandle);
  if (keyCount <= 0)
    return;
  InfoKey* keys = new InfoKey[keyCount];
  BOOL succeeded = GetNotesKeys(noteHandle, keys, keyCount);

  if (succeeded){
    for (int i=0; i < keyCount; i++){
      printf("Key is: %s\n", keys[i]);
      printf("Value is: %s\n\n", GetNote(noteHandle, keys[i]);
    }
  }
  delete []keys;
}
CloseNotesFile(noteHandle);
```

BOOL GetSiteRootForFile()

Description

Determines the site root for the specified Design Notes file.

Arguments

`const char*` *`filePath`*, `char*` *`siteRootBuf`*, `int` *`siteRootBufMaxLen`*, `{InfoPrefs*` *`infoPrefs`*`}`

- The first argument is the file URL of the file for which you want the site root.
- The second argument is the buffer where the site root should be stored.
- The third argument is the maximum size of *`siteRootBuf`*.
- The optional fourth argument is a reference to a `struct` in which the preferences for the site should be stored.

Returns

A Boolean value that indicates whether the operation is successful; stores the site root in *`siteRootBuf`*. If *`infoPrefs`* is specified, the function also returns the Design Notes preferences for the site. The `InfoPrefs` struct has two variables: `bUseDesignNotes` and `bUploadDesignNotes`, both of type `BOOL`.

BOOL GetVersionName()

Description

Gets the version name of the MMNotes shared library, which indicates the application that implemented it.

Arguments

`char* versionNameBuf, int versionNameBufMaxLen`

- The first argument is the buffer where the version name should be stored.
- The second argument is the maximum size of *versionNameBuf*.

Returns

A Boolean value that indicates whether the operation is successful; stores "Dreamweaver" in *versionNameBuf*.

BOOL GetVersionNum()

Description

Gets the version number of the MMNotes shared library, which allows you to determine whether certain functions are available.

Arguments

`char* versionNumBuf, int versionNumBufMaxLen`

- The first argument is the buffer where the version number should be stored.
- The second argument is the maximum size of *versionNumBuf*.

Returns

A Boolean value that indicates whether the operation is successful; stores the version number in *versionNumBuf*.

BOOL LocalURLToFilePath()

Description

Converts the specified file:// URL to a local drive path.

Arguments

`const char* localURL, char* drivePathBuf, int drivePathMaxLen`

- The first argument is the path to a local file, which is expressed as a file:// URL.
- The second argument is the buffer where the local drive path should be stored.
- The third argument is the maximum size of *drivePathBuf*.

Returns

A Boolean value that indicates whether the operation is successful; stores the local drive path in *drivePathBuf*.

FileHandle OpenNotesFile()

Description

Opens the Design Notes file that is associated with the specified file or creates one if none exists.

Arguments

`const char* localFileURL, {BOOL bForceCreate}`

- The first argument is a string that contains the path to the main file with which the Design Notes file is associated, which is expressed as a file:// URL.
- The second argument is a Boolean value that indicates whether to create the Design Notes file even if Design Notes is turned off for the site or if `localFileURL` is not associated with any site.

FileHandle OpenNotesFilewithOpenFlags()

Description

Opens the Design Notes file that is associated with the specified file or creates one if none exists. You can open the file in read-only mode.

Arguments

`const char* localFileURL, {BOOL bForceCreate}, {BOOL bReadOnly}`

- The first argument is a string that contains the path to the main file with which the Design Notes file is associated, which is expressed as a file:// URL.
- The second argument is a Boolean value that indicates whether to create the Design Notes file even if Design Notes are turned off for the site or `filePath` is not associated with any site. The default value is `false`. This argument is optional, but you need to specify it if you specify the third argument.
- The third argument is a Boolean value that indicates whether to open the file in read-only mode. The default value is `false`. Optional. Available starting in version 2 of MMNotes.dll.

BOOL RemoveNote()

Description

Removes the specified key (and its value) from the specified Design Notes file.

Arguments

`FileHandle noteHandle, const char keyName[64]`

- The first argument is the file handle that `OpenNotesFile()` returns.
- The second argument is a string that contains the name of the key to remove.

Returns

A Boolean value that indicates whether the operation is successful.

BOOL SetNote()

Description

Creates or updates one key/value pair in a Design Notes file.

Arguments

`FileHandle` *`noteHandle`*, `const char` *`keyName[64]`*, `const char*` *`value`*

- The first argument is the file handle that `OpenNotesFile()` returns.
- The second argument is a string that contains the name of the key.
- The third argument is a string that contains the value.

Returns

A Boolean value that indicates whether the operation is successful.

CHAPTER 25
The Fireworks Integration API

FWLaunch is a C shared library that gives authors of objects, commands, behaviors, and Property inspectors the ability to communicate with Macromedia Fireworks. This chapter describes the Fireworks Integration API and how to use it; for general information on how C libraries interact with the JavaScript interpreter in Macromedia Dreamweaver MX, see "C-Level Extensibility" on page 251.

All functions in the Fireworks Integration API are methods of the `FWLaunch` object. Optional arguments are enclosed in braces ({ }).

FWLaunch.bringDWToFront()

Availability

Dreamweaver 3, Fireworks 3

Description

Brings Dreamweaver to the front.

Arguments

None.

Returns

Nothing.

FWLaunch.bringFWToFront()

Availability

Dreamweaver 3, Fireworks 3

Description

Brings Fireworks to the front if it is running.

Arguments

None.

Returns

Nothing.

FWLaunch.execJsInFireworks()

Availability

Dreamweaver 3, Fireworks 3

Description

Passes the specified string of JavaScript to Fireworks for execution.

Arguments

javascriptOrFileURL

The argument is either a string of literal JavaScript or the path to a .js or .jsf file, which is expressed as a file:// URL.

Returns

A cookie object if the JavaScript passes successfully or a nonzero error code that indicates one of the following errors occurred:

- Invalid usage; *javascriptOrFileURL* is specified as `null` or an empty string, or the path to the .js or .jsf file is invalid.
- File I/O error; Fireworks cannot create a Response file because the disk is full.
- Error notifying Dreamweaver that the user is not running a valid version of Dreamweaver (3 or later).
- Error launching Fireworks process; the function does not launch a valid version of Fireworks (3 or later).
- User cancelled the operation.

FWLaunch.getJsResponse()

Availability

Dreamweaver 3, Fireworks 3

Description

Determines whether Fireworks is still executing the JavaScript passed to it by `FWLaunch.execJsInFireworks()`, whether the script completed successfully, or whether an error occurred.

Arguments

progressTrackerCookie

The argument is the cookie object that `FWLaunch.execJsInFireworks()` returns.

Returns

A string that contains the result of the script passed to `FWLaunch.execJsInFireworks()` if the operation completed successfully, `null` if Fireworks is still executing the JavaScript, or a nonzero error code that indicates one of the following errors occurred:

- Invalid usage; a JavaScript error occurred as Fireworks executed the script.
- File I/O error; Fireworks cannot create a Response file because the disk is full.
- Error notifying Dreamweaver; the user is not running a valid version of Dreamweaver (3 or later).

- Error launching Fireworks process; the function does not launch a valid version of Fireworks (3 or later).
- User cancelled the operation.

Returns

The following code passes the string `"prompt('Please enter your name:')"` to `FWLaunch.execJsInFireworks()` and checks for the result:

```
var progressCookie = FWLaunch.execJsInFireworks("prompt('Please enter your
  name:')");
var doneFlag = false;
while (!doneFlag){
  // check for completion every 1/2 second
  setTimeout('checkForCompletion()',500);
}

function checkForCompletion(){
  if (progressCookie != null) {
    var response = FWLaunch.getJsResponse(progressCookie);
    if (response != null) {
      if (typeof(response) == "number") {
        // error or user-cancel, time to close the window
        // and let the user know we got an error
        window.close();
        alert("An error occurred.");
      }else{
        // got a valid response!
        alert("Nice to meet you, " + response);
        window.close();
      }
        doneFlag = true;
    }
  }
}
```

FWLaunch.mayLaunchFireworks()

Availability

Dreamweaver 2, Fireworks 2

Description

Determines whether it is possible to launch a Fireworks optimization session.

Arguments

None.

Returns

A Boolean value that indicates whether the platform is Windows or Macintosh; if Macintosh, indicates whether another Fireworks optimization session is already running.

FWLaunch.optimizeInFireworks()

Availability

Dreamweaver 2, Fireworks 2

Description

Launches a Fireworks optimization session for the specified image.

Arguments

docURL, *imageURL*, *{targetWidth}*, *{targetHeight}*

- The first argument is the path to the active document, which is expressed as a file:// URL.
- The second argument is the path to the selected image. If the path is relative, it is relative to *docURL*.
- The third argument, if supplied, is the width to which the image should be resized.
- The fourth argument, if supplied, is the height to which the image should be resized.

Returns

Zero, if a Fireworks optimization session successfully launches for the specified image; otherwise, a nonzero error code that indicates one of the following errors occurred:

- Invalid usage; *docURL*, *imageURL*, or both are specified as `null` or an empty string.
- File I/O error; Fireworks cannot create a response file because the disk is full.
- Error notifying Dreamweaver that the user is not running a valid version of Dreamweaver (2 or later).
- Error launching Fireworks process; the function does not launch a valid version of Fireworks (2 or later).
- User cancelled the operation.

FWLaunch.validateFireworks()

Availability

Dreamweaver 2, Fireworks 2

Description

Looks for the specified version of Fireworks on the user's hard disk.

Arguments

{versionNumber}

The argument is a floating-point number that is greater than or equal to 2; it represents the required version of Fireworks. If this argument is omitted, the default is `2`.

Returns

A Boolean value that indicates whether the specified version of Fireworks was found.

Example

The following code checks whether Fireworks 3 is installed:

```
if (FWLaunch.validateFireworks(3.0)){
  alert( "Fireworks 3.0 is installed.");
}else{
  alert( "Fireworks 3.0 is not installed.");
}
```

A simple Fireworks integration example

The following command asks Fireworks to prompt the user for his or her name and returns the name to Dreamweaver:

```
<html>
<head>
<title>Prompt in Fireworks</title>
<meta http-equiv="Content-Type" content="text/html; ¬
charset=iso-8859-1">
<script>

function commandButtons(){
  return new Array("Prompt", "promptInFireworks()", "Cancel", ¬
  "readyToCancel()", "Close","window.close()");
}

var gCancelClicked = false;
var gProgressTrackerCookie = null;

function readyToCancel() {
  gCancelClicked = true;
}

function promptInFireworks() {
  var isFireworks3 = FWLaunch.validateFireworks(3.0);
  if (!isFireworks3) {
    alert("You must have Fireworks 3.0 or later to use this ¬
    command");
  return;
}

  // Tell Fireworks to execute the prompt() method.
  gProgressTrackerCookie = FWLaunch.execJsInFireworks¬
  ("prompt('Please enter your name:')");

  // null means it wasn't launched, a number means an error code
  if (gProgressTrackerCookie == null || ¬
  typeof(gProgressTrackerCookie) == "number") {
    window.close();
    alert("an error occurred");
    gProgressTrackerCookie = null;
  } else {
    // bring Fireworks to the front
    FWLaunch.bringFWToFront();
    // start the checking to see if Fireworks is done yet
    checkOneMoreTime();
  }
}

function checkOneMoreTime() {
  // Call checkJsResponse() every 1/2 second to see if Fireworks
   // is done yet
  window.setTimeout("checkJsResponse();", 500);
}

function checkJsResponse() {
  var response = null;

  // The user clicked the cancel button, close the window
  if (gCancelClicked) {
    window.close();
```

```
      alert("cancel clicked");
    } else {
      // We're still going, ask Fireworks how it's doing
      if (gProgressTrackerCookie != null)
        response = ¬
        FWLaunch.getJsResponse(gProgressTrackerCookie);

      if (response == null) {
        // still waiting for a response, call us again in 1/2 a
        // second
        checkOneMoreTime();
      } else if (typeof(response) == "number") {
        // if the response was a number, it means an error
        // occurred
        // the user cancelled in Fireworks
        window.close();
        alert("an error occurred.");

      } else {
        // got a valid response!  This return value might not
        // always be a useful one, since not all functions in
        // Fireworks return a string, but we know this one does,
        // so we can show the user what we got.
        window.close();
        FWLaunch.bringDWToFront();  // bring Dreamweaver to the ¬
        front
        alert("Nice to meet you, " + response + "!");
      }
    }
}

</script>
</head>
<body>
<form>
<table width="313" nowrap>
<tr>
<td>This command asks Fireworks to execute the prompt() ¬
function. When you click Prompt, Fireworks comes forward and ¬
asks you to enter a value into a dialog box. That value is then ¬
returned to Dreamweaver and displayed in an alert.</td>
</tr>
</table>
</form>
</body>
</html>
```

CHAPTER 26
The Flash Objects API

The Flash Objects API lets extension developers build objects that create simple Macromedia Flash content. This API provides a way to set parameters in a Flash Generator template (.swt file) and output a Flash Movie or Image file. The API lets you create new Flash objects as well as read and manipulate existing Flash objects. The Flash button and Flash text features are built using this API.

The .swt file is a Flash Generator Template file, which contains all the necessary information required to construct a Flash Object file (.swf). These API functions let you create a new .swf file (or Image file) from a .swt file by replacing the parameters of the .swt file with real values. For more information on Flash, see the Flash manual.

SWFFile.createFile()

Description

Generates a new Flash Object file with the specified template and array of parameters. Also creates a GIF, PNG, JPEG, and MOV version of the title if filenames for those formats are specified.

If you want to specify an optional parameter that follows optional parameters you do not want to include, you need to specify empty strings for the extraneous parameters. For example, if you want to specify a .png file, but not a .gif file, you need to specify an empty string before specifying the.png filename.

Arguments

templateFile, *templateParams*, *swfFileName*, *{gifFileName}*, *{pngFileName}*, *{jpgFileName}*, *{movFileName}*, *{generatorParams}*

- *templateFile* is a path to a Template file, which is expressed as a file:// URL. This file can be a .swt file.
- *templateParams* is an array of name/value pairs where the names are the parameters in the .swt file and the values are what you want to specify for those parameters. For an .swf file to be recognized by Macromedia Dreamweaver MX as a Flash object, the first parameter must be `"dwType"`. Its value should be a string that represents the name of the object type, such as `"Flash Text"`.
- *swfFileName* is the output filename of an .swf file, which is expressed as a file:// URL, or an empty string to ignore.
- *{gifFileName}* is the output filename of a .gif filename, which is expressed as a file://URL Optional.

- *{pngFileName}* is the output filename of a .png filename, which is expressed as a file:// URL. Optional.
- *{jpgFileName}* is the output filename of a .jpeg filename, which is expressed as a file:// URL. Optional.
- *{movFileName}* is the output filename of a QuickTime movie filename, which is expressed as a file:// URL. Optional.
- *{generatorParams}* is an array of strings that represents optional Generator command line flags. Optional. Each flag must be followed in the array by its data items. Some commonly used flags are listed in the following table.

Option Flag	Data	Description	Example
-defaultsize	Width, height	Sets the output image size to the specified width and height	`"-defaultsize", "640", "480"`
-exactFit	None	Stretches the contents in the output image to fit exactly into the specified output size	`"-exactFit"`

Returns

A string that contains one of the following values:

- `"noError"` means the call completed successfully.
- `"invalidTemplateFile"` means the specified Template file is invalid or not found.
- `"invalidOutputFile"` means at least one of the specified output filenames is invalid.
- `"invalidData"` means that one or more of the *templateParams* is invalid.
- `"initGeneratorFailed"` means Generator cannot be initialized.
- `"outOfMemory"` means there is insufficient memory to complete the operation.
- `"unknownError"` means an unknown error occurred.

Example

The following JavaScript creates a Flash Object file of type `"myType"`, which replaces any occurrence of `"text"` inside the Template file with `"Hello World"`. It creates a .gif file as well as an .swf file.

```
var params = new Array;
params[0] = "dwType";
params[1] = "myType";
params[2] = "text";
params[3] = "Hello World";
errorString = SWFFile.createFile( "file:///MyMac/test.swt", ¬
params, "file:///MyMac/test.swf", "file:///MyMac/test.gif");
```

SWFFile.getNaturalSize()

Description

Returns the natural size of any Flash movie.

Arguments

fileName

fileName is a path to the Flash movie, which is expressed as a file:// URL.

Returns

An array that contains two elements that represent the width and the height of the movie or `null` if the file is not a Flash file.

SWFFile.getObjectType()

Description

Returns the Flash object type; the value that passed in the `dwType` parameter when the file was created by the `SWFFile.createFile()` function.

Arguments

fileName

fileName is a path to a Flash Object file, which is expressed as a file:// URL. This file is usually an .swf file.

Returns

A string that represents the object type, or `null` if the file is not a Flash Object file or if the file cannot be found.

Example

The following code checks to see if the file, test.swf, is a Flash object of type `myType`:

```
if ( SWFFile.getObjectType("file:///MyMac/test.swf") == ¬
"myType" ){
  alert ("This is a myType object.");
}else{
  alert ("This is not a myType object.");
}
```

SWFFile.readFile()

Description

Reads a Flash Object file.

Arguments

fileName

fileName is a path to a Flash Object file, which is expressed as a file:// URL.

Returns

An array of strings where the first array element is the full path to the template .swt file. The following strings represent the parameters (name/value pairs) for the object. Each name is followed in the array by its value. The first name/value pair is `"dwType"` followed by its value; `null` is returned if the file cannot be found or if it is not a Flash Object file.

Example

Calling `var params = SWFFile.readFile("file:///MyMac/test.swf")` returns the following values in the parameters array:

```
"file:///MyMac/test.swt"  // template file used to create this .swf file
"dwType"                  // first parameter
"myType"                  // first parameter value
"text"                    // second parameter
"Hello World"             // second parameter value
```

CHAPTER 27
The Database API

Functions in the Database API let you manage database connections and access information that is stored in databases.

In managing database connections, you can get the user name and password that are needed to make a connection to a database, open up a database connection dialog box, and so on.

In accessing database information, you can, for example, fetch metadata that describes the schema or structure of a database. This metadata includes information such as the names of tables, columns, stored procedures, and views. You can also show the results of executing a database query or stored procedure. When accessing a database through this API, you use structured query language (SQL) statements.

Database API functions are used at design time when users are building web applications, as opposed to runtime, when the web application is deployed.

You can use these functions in any extension. In fact, the Macromedia Dreamweaver MX Server Behavior, Data Format, and Data Sources API functions all use these database functions.

The following example shows how the server behavior function, `getDynamicBindings()`, is defined for Recordset. (For all Dreamweaver versions other than MX, the file Recordset.js is located in the Configuration/ServerBehaviors folder; for Dreamweaver MX, this JavaScript file is located in the Configuration/ServerBehaviors/Shared folder.)

Note: This example uses the `MMDB.getColumnList()` function.

```
function getDynamicBindings(elementNode)
{
  var ss = findSSrec(elementNode, LABEL_Type)

  var connString = ss.activeconnection
  var connName =  ss.connectionName
  var statement = ss.source
  var rsName = ss.rsName

  var pa = new Array()

  if (String(ss.ParamArray) != "undefined")
  {
    for (var i = 0; i < ss.ParamArray.length; i++)
    {
      pa[i] = new Array()
      pa[i][0] = ss.ParamArray[i].name
      pa[i][1] = ss.ParamArray[i].value
    }
  }

  var statement = ReplaceParamsWithVals(statement, pa)
  return MMDB.getColumnList(connName, statement)
}
```

Database connection functions

Database connection functions let you make and manage any connection, including the Dreamweaver MX-provided ADO, ColdFusion, and JDBC connections. These functions interface with the Connection Manager only; they do not access a database. For functions that access a database, see "Database access functions" on page 324.

MMDB.deleteConnection()

Availability

Dreamweaver MX

Description

Deletes the named database connection.

Arguments

connName

connName is the name of the database connection as it is specified in the Connection Manager. This argument identifies, by name, the database connection to delete.

Returns

Nothing.

Example

```
//deletes a connection
function clickedDelete()
{
  var selectedObj = dw.serverComponents.getSelectedNode();
  if (selectedObj && selectedObj.objectType=="Connection")
  {
    var connRec = MMDB.getConnection(selectedObj.name);
    if (connRec)
    {
      MMDB.deleteConnection(selectedObj.name);
      dw.serverComponents.refresh();
    }
  }
}
```

MMDB.getColdFusionDsnList()

Availability

Dreamweaver UltraDev 4

Description

Gets the ColdFusion data source names (DSNs) from the site server, using the `getRdsUserName()` and `getRdsPassword()` functions.

Arguments

None.

Returns

An array that contains the ColdFusion DSNs that are defined on the server for the current site.

MMDB.getConnection()

Availability

Dreamweaver UltraDev 4, enhanced in Dreamweaver MX

Description

Gets a named connection object. Connection objects contain the following properties:

Property	Description
name	Connection name
type	Indicates, if `useHTTP` is `false`, which DLL to use for connecting to a database at runtime
string	Runtime ADO connection string or JDBC URL
dsn	ColdFusion DSN
driver	Runtime JDBC Driver
username	Runtime user name
password	Runtime password
useHTTP	String that contains either `true` or `false`, specifying whether to use a remote driver (HTTP connection) at design time; otherwise, use a local driver (DLL)
includePattern	Regular expression used to find the file include statement on the page during Live Data and Preview In Browser

Property	Description
variables	Array of page variable names and their corresponding values used during Live Data and Preview In Browser
catalog	Used to restrict the metadata that appears (for more information, see "MMDB.getProcedures()" on page 328)
schema	Used to restrict the metadata that appears (for more information, see "MMDB.getProcedures()" on page 328)
filename	Filename of dialog box that was used to create the connection

Note: These properties are the standard ones implemented by Dreamweaver MX. Developers can define their own connection types and add new properties to this standard set or provide a different set of properties.

Parameters

name

name is a string variable that specifies the name of the connection that you want to reference.

Returns

A reference to a named connection object.

MMDB.getConnectionList()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of all the connection strings that are defined in the Connection Manager.

Arguments

None.

Returns

An array of strings where each string is the name of a connection as it appears in the Connection Manager.

Example

A call to `MMDB.getConnectionList()` could return the strings `["EmpDB", "Test", TestEmp"]`.

MMDB.getConnectionName()

Availability

Dreamweaver UltraDev 1

Description

Gets the connection name that corresponds to the specified connection string. This function is useful when you need to reselect a connection name in the user interface from data on the page.

If you have a connection string that references two drivers, you can specify both the connection string and the driver that corresponds to the connection name that you want returned. For example, you could have two connections:

Connection 1 has the following properties:

```
ConnectionString="jdbc:inetdae:velcro-qa-5:1433?database=pubs"
DriverName="com.inet.tds.TdsDriver"
```

Connection 2 has the following properties:

```
ConnectionString="jdbc:inetdae:velcro-qa-5:1433?database=pubs"
DriverName="com.inet.tds.TdsDriver2"
```

The connection strings for Connection 1 and Connection 2 are the same. Connection 2 connects to a more recent version of `TdsDriver`. You should pass the driver name to this function to fully qualify the connection name you want returned.

Arguments

`connString {,driverName}`

- *`connString`* is the connection string that gets the connection name.
- *`driverName`* is an optional argument that further qualifies *`connString`*.

Returns

A connection name string that corresponds to the connection string.

Example

The following code returns the string `"EmpDB"`:

```
var connectionName = MMDB.getConnectionName ¬
("dsn=EmpDB;uid=;pwd=");
```

MMDB.getConnectionString()

Availability

Dreamweaver UltraDev 1

Description

Gets the connection string that is associated with the named connection.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

A connection string that corresponds to the named connection.

Example

The code `var connectionString = MMDB.getConnectionString ("EmpDB")` returns different strings for an ADO or JDBC connection.

For an ADO connection, the following string could be returned:

```
"dsn=EmpDB;uid=;pwd=";
```

For a JDBC connection, the following string could be returned:

```
"jdbc:inetdae:192.168.64.49:1433?database=pubs&user=JoeUser&¬
password=joesSecret"
```

MMDB.getDriverName()

Availability

Dreamweaver UltraDev 1

Description

Gets the driver name that is associated with the specified connection. Only a JDBC connection has a driver name.

Arguments

connName

connName is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

A string that contains the driver name.

Example

The statement `MMDB.getDriverName ("EmpDB");` might return the following string:

```
"jdbc/oracle/driver/JdbcOracle"
```

MMDB.getDriverUrlTemplateList()

Availability

Dreamweaver UltraDev 4, deprecated in Dreamweaver MX.

Note: For Dreamweaver UltraDev 4, the list of JDBC drivers are stored in the connections.xml file located in the Configuration/Connections folder. Each driver has an associated URL template. This funtion returns the list of JDBC drivers.

For Dreamweaver MX, these drivers and URL templates are hard-coded in the JDBC dialog boxes. In addition, this function is an empty function definition to eliminate undefined-function errors. The following example shows how a JDBC driver and URL template are hard-coded:

```
var DEFAULT_DRIVER = "COM.ibm.db2.jdbc.app.DB2Driver";
var DEFAULT_TEMPLATE = "jdbc:db2:[database name]";
```

For Dreamweaver MX, there is a dialog box for each driver/URL template pair.

In summary, Dreamweaver UltraDev 4 developers need to add a new entry to the XML, and Dreamweaver MX developers need to implement a new dialog box.

Description

Gets JDBC Drivers and respective URL templates.

Arguments

None.

Returns

An array that contains JDBC drivers that have been detected on the user's system and their respective URL templates, if they are specified. The array has an even number of elements that contain: `Driver1`, `UrlTemplate1`, `Driver2`, `UrlTemplate2`, and so on.

MMDB.getLocalDsnList()

Availability

Dreamweaver UltraDev 4

Description

Gets ODBC DSNs that are defined on the user system.

Arguments

None.

Returns

An array that contains the ODBC DSNs that are defined on the user's system.

MMDB.getPassword()

Availability

Dreamweaver UltraDev 1

Description

Gets the password that is used for the specified connection.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

A password string that is associated with the connection name.

Example

The statement `MMDB.getPassword ("EmpDB");` might return `"joessecret"`.

MMDB.getRdsPassword()

Availability

Dreamweaver UltraDev 4

Description

Gets the Remote Development Services (RDS) password.

Arguments

None.

Returns

A string that contains the RDS password.

MMDB.getRdsUserName()

Availability

Dreamweaver UltraDev 4

Description

Gets the RDS user name.

Arguments

None.

Returns

A string that contains the name of the RDS user.

MMDB.getRemoteDsnList()

Availability

Dreamweaver UltraDev 4, enhanced in Dreamweaver MX

Description

Gets the ODBC DSNs from the site server. The `getRdsUserName()` and `getRdsPassword()` functions are used when the server model of the current site is ColdFusion. This function provides an option for a developer to specify a URL parameter string to be appended to the Remote Connectivity URL that `MMDB.getRemoteDsnList()` generates. If the developer provides a parameter string, this function passes it to the HTTP connectivity scripts.

Arguments

{urlParams}

`urlParams` is a string that contains a list of *`name=value`* expressions, which are separated by ampersand (`&`) characters. You must not enclose values with quotes. Some characters, such as the space in the value "`Hello World`," need to be encoded. For example, here is a valid sample argument that you can pass to `MMDB.getRemoteDsnList()`:

```
a=1&b=Hello%20World
```

Returns

Returns an array that contains the ODBC DSNs that are defined on the server for the current site

MMDB.getRuntimeConnectionType()

Availability

Dreamweaver UltraDev 1

Description

Returns the runtime connection type of the specified connection name. This function can return one of the following values: `"ADO"`, `"ADODSN"`, `"JDBC"`, or `"CFDSN"`.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

A string that corresponds to the connection type.

Example

The following code returns the string `"ADO"` for an ADO connection:

```
var connectionType = MMDB.getRuntimeConnectionType ("EmpDB")
```

MMDB.getUserName()

Availability

Dreamweaver UltraDev 1

Description

Returns a user name for the specified connection.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

A user name string that is associated with the connection name.

Example

The statement `MMDB.getUserName ("EmpDB");` might return `"amit"`.

MMDB.hasConnectionWithName()

Availability

Dreamweaver UltraDev 4

Description

Determines whether a connection of a given name exists.

Arguments

name

name is the connection name.

Returns

Returns a Boolean value that indicates whether a connection with the specified name exists.

MMDB.needToPromptForRdsInfo()

Availability

Dreamweaver MX

Description

Determines whether Dreamweaver should open the RDS Login Information dialog box.

Arguments

force

force, which holds a Boolean value, indicates whether the fact that the user has previously cancelled out of the RDS login dialog box should be ignored when trying to determine whether the user still needs to be prompted for RDS login information.

Returns

A Boolean value that indicates whether the user needs to be prompted for RDS login information.

MMDB.needToRefreshColdFusionDsnList()

Availability

Dreamweaver MX

Description

Tells the Connection Manager to empty the cache and get the ColdFusion data source list from the application server the next time a user requests the list.

Arguments

None.

Returns

Nothing.

MMDB.popupConnection()

Availability

Dreamweaver MX

Description

Invokes a connection dialog box. This function has the following three signatures:

- If the argument list consists only of *dialogFileName* (a string), popupConnection( ) causes Dreamweaver to launch the connection dialog box so you can define a new connection.
- If the argument list consists only of *connRec* (a connection reference), popupConnection( ) causes Dreamweaver to launch the connection dialog box in edit mode for the named connection for editing. In this mode, the name text box is dimmed.
- If the argument list consists of *connRec* and *bDuplicate* (a Boolean value), popupConnection( ) causes Dreamweaver to launch the connection dialog box in duplicate mode. In this mode, the name text box is blanked out and the remaining properties are copied to define a duplicate connection.

Arguments

dialogFileName
or
connRec
or
connRec, *bDuplicate*

- *dialogFileName* is a string that contains the name of an HTML file that resides in the Configuration/Connections/*server-model* folder. This HTML file defines the dialog box that is used to create a connection. This file must implement three JavaScript API functions: findConnection(), inspectConnection(), and applyConnection(). Typically, you create a .js file that implements these functions and then include the .js file in the HTML file. (For more information on creating a connection, see "The Database Connectivity API" on page 337.)
- *connRec* is a reference to an existing Connection object.
- *bDuplicate* is a Boolean value.

Returns

Nothing. The defined connection dialog box appears.

MMDB.setRdsPassword()

Availability

Dreamweaver UltraDev 4

Description

Sets the RDS password.

Arguments

password is a string that contains the RDS password.

Returns

Nothing.

MMDB.setRdsUserName()

Availability

Dreamweaver UltraDev 4

Description

Sets the RDS user name.

Arguments

username is the name of a valid RDS user.

Returns

Nothing.

MMDB.showColdFusionAdmin()

Availability

Dreamweaver MX

Description

Displays the ColdFusion Administrator dialog box.

Arguments

None.

Returns

Nothing. The ColdFusion Administrator dialog box appears.

MMDB.showConnectionMgrDialog()

Availability

Dreamweaver UltraDev 1

Description

Displays the Connection Manager dialog box.

Arguments

Nothing.

Returns

Nothing. The Connection Manager dialog box appears.

MMDB.showOdbcDialog()

Availability

Dreamweaver UltraDev 4 (Windows only)

Description

Displays the System ODBC Administration dialog box or the ODBC Data Source Administrator dialog box.

Arguments

None.

Returns

Nothing. The System ODBC Administration dialog box or the ODBC Data Source Administrator dialog box appears.

MMDB.showRdsUserDialog()

Availability

Dreamweaver UltraDev 4

Description

Displays RDS user name and password dialog box.

Arguments

username, *password*

- *username* is the initial user name.
- *password* is the initial password.

Returns

An object that contains the new values in the `username` and `password` properties. Either property that is not being defined indicates that the user cancelled from the dialog box.

MMDB.showRestrictDialog()

Availability

Dreamweaver UltraDev 4

Description

Displays the Restrict dialog box.

Arguments

catalog, *schema*

- *catalog* is the initial catalog value.
- *schema* is the initial schema value.

Returns

An object that contains the new values in the `catalog` and `schema` properties. Either property that is not being defined indicates that the user cancelled from the dialog box.

MMDB.testConnection()

Availability

Dreamweaver UltraDev 4

Description

Tests connection settings. Displays a modal dialog box that describes the results.

Arguments

This function expects a single argument—an Array object that contains values from the following list, which are appropriate for the current server model. For properties that do not apply to the connection being tested, set them to empty ("").

- *type* indicates, when `useHTTP` is `false`, which DLL to use for connecting to a database at design time. Used to test connection settings.
- *string* is the ADO connection string or JDBC URL.
- *dsn* is the Data Source Name.
- *driver* is the JDBC driver.
- *username* is the user name.
- *password* is the password.
- *useHTTP* is a Boolean value. A value of `true` specifies that Dreamweaver should use an HTTP connection at design time; otherwise, Dreamweaver uses a DLL.

Returns

A Boolean value. If the connection test is successful, `testConnection()` returns `true`; `false` otherwise.

Database access functions

Database access functions let you query a database. For the collection of functions that manage a database connection, see "Database connection functions" on page 312.

The following list describes some of the arguments that are common to the functions that are available:

- Most database access functions use a connection name as an argument. You can see a list of valid connection names in the Connection Manager, or you can use `MMDB.getConnectionList()` to get a list of all the connection names programmatically.
- Stored procedures often require parameters. There are two ways of specifying parameter values for database access functions. First, you can provide an array of parameter values (*paramValuesArray*). If you specify only parameter values, the values need to be in the sequence in which the stored procedure requires the parameters. Second, you specify parameter values to provide an array of parameter names (*paramNameArray*). (You can use the `MMDB.getSPParamsAsString()` function to get the parameters of the stored procedure.) If you provide parameter names, the values that are specified in *paramValuesArray* need to be in the sequence in which the parameter names are specified in *paramNameArray*.

MMDB.getColumnAndTypeList()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of columns and their types from an executed SQL `SELECT` statement.

Arguments

`connName`, *`statement`*

- *`connName`* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *`statement`* is the SQL `SELECT` statement to execute.

Returns

An array of strings that represents a list of columns (and their types) that match the `SELECT` statement, or an error if the SQL statement is invalid or the connection cannot be made.

Example

The code `var columnArray = MMDB.getColumnAndTypeList("EmpDB","Select * from Employees")` returns the following array of strings:

```
columnArray[0] = "EmpName"
columnArray[1] = "varchar"
columnArray[2] = "EmpFirstName"
columnArray[3] = "varchar"
columnArray[4] = "Age"
columnArray[5] = "integer"
```

MMDB.getColumnList()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of columns from an executed SQL `SELECT` statement.

Arguments

`connName`, *`statement`*

- *`connName`* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *`statement`* is the SQL `SELECT` statement to execute.

Returns

An array of strings that represents a list of columns that match the `SELECT` statement, or an error if the SQL statement is invalid or the connection cannot be made.

Example

The code `var columnArray = MMDB.getColumnList("EmpDB","Select * from Employees")` returns the following array of strings:

```
columnArray[0] = "EmpName"
columnArray[1] = "EmpFirstName"
columnArray[2] = "Age"
```

MMDB.getColumns()

Availability

Dreamweaver MX

Description

Executes the specified SQL statement and, regardless of whether any rows are returned, returns an array of objects that describe the columns that are specified in the SQL statement. When `MMDB.getColumns( )` executes a `SELECT` statement, the returned rows are parsed to build the JavaScript data object list.

Arguments

`connName`, *`sqlStatement`*

- *`connName`* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *`sqlStatement`* is the SQL statement that `MMDB.getColumns( )` executes.

Returns

An array of column objects, one object for each column. Each object defines the following three properties for the column with which it is associated.

property name	description
name	Name of the column (for example, `price`)
datatype	Data type of the column (for example, `small money`)
definedsize	Defined size of the column (for example, `8`)

Example

```
var connName  = componentRec.parent.parent.parent.name;
var sqlstatement = "select * from " + tableName + " where 1=0";
var columnNameObjs = MMDB.getColumns(connName,sqlstatement);
var columnName = columnNameObjs[i];
var tooltiptext = columnName.datatype;
tooltiptext+=" ";
tooltiptext+=columnName.definedsize;
```

MMDB.getColumnsOfTable()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of all the columns in the specified table.

Arguments

connName, *tableName*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *tableName* is the name of a table in the database that is specified by *connName*.

Returns

An array of strings where each string is the name of a column in the table.

Example

The statement `MMDB.getColumnsOfTable ("EmpDB","Employees");` returns the following strings:

```
["EmpID", "FirstName", "LastName"]
```

MMDB.getPrimaryKeys()

Availability

Dreamweaver MX

Description

Returns the column names that combine to form the primary key of the named table. A primary key serves as the unique identifier for a database row and consists of at least one column.

Arguments

connName, tableName

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *tableName* is the name of the table for which you want to retrieve the set of columns that comprises the primary key of that table.

Returns

An array of strings. The array contains one string for each column that comprises the primary key.

Example

```
var connName    = componentRec.parent.parent.parent.name;
var tableName   = componentRec.name;
var primaryKeys = MMDB.getPrimaryKeys(connName,tableName);
```

MMDB.getProcedures()

Availability

Dreamweaver MX

Description

Returns an array of procedure objects that are associated with a named connection.

Arguments

connName

connName is a connection name as specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

An array of procedure objects where each procedure object has the following set of three properties:

Property Name	Description
schema*	Name of the schema associated with the object. This property identifies the user that is associated with the stored procedure in the SQL database that `getProcedures()` accesses. The database that this function accesses depends on the type of connection. For ODBC connections, the ODBC data source defines the database. The DSN is specified by the `dsn` property in the connection object (`connName`) that you pass to `getProcedures()`. For OLE DB connections, the connection string names the database.
catalog	Name of the catalog associated with the object (owner qualifier). The value of the `catalog` property is defined by an attribute of the OLE DB driver. This driver attribute defines a default `user.database` to use when the OLE DB connection string does not specify a database.
procedure	Name of the procedure.

* Dreamweaver MX connects to and gets all the tables in the database whenever you modify a recordset. If the database has many tables, Dreamweaver might take a long time to retrieve them on certain systems. If your database contains a schema or catalog, you can use the schema or catalog to restrict the number of database items Dreamweaver gets at design time. You must first create a schema or catalog in your database application before you can apply it in Dreamweaver. Consult your database documentation or your system administrator.

Example

```
//get a list of procedures.
var procObjects   = MMDB.getProcedures(connectionName);
for (i = 0; i < procObjects.length; i++)
{
  var thisProcedure = procObjects[i]
  thisSchema =  Trim(thisProcedure.schema)
  if (thisSchema.length == 0)
  {
  thisSchema = Trim(thisProcedure.catalog)
  }
  if (thisSchema.length > 0)
  {
  thisSchema += "."
  }

  var procName = String(thisSchema + thisProcedure.procedure);
 }
```

MMDB.getSPColumnList()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of result set columns that are generated by a call to the specified stored procedure.

Arguments

connName, *statement*, *paramValuesArray*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *statement* is the name of the stored procedure that returns the result set when it executes.
- *paramValuesArray* is an array that contains a list of design-time parameter test values. Specify the parameter values in the order in which the stored procedure expects them. You can use the `MMDB.getSPParamsAsString()` function to get the parameters of the stored procedure.

Returns

An array of strings that represents the list of columns. This function returns an error if the SQL statement or the connection string is invalid.

Example

The following code could return a list of result set columns that are generated from the executed stored procedure, `getNewEmployeesMakingAtLeast`:

```
var paramValueArray = new Array("2/1/2000", "50000")
var columnArray = MMDB.getSPColumnList("EmpDB", ¬
"getNewEmployeesMakingAtLeast", paramValueArray)
```

The following values are returned:

```
columnArray[0] = "EmpID", columnArray[1] = "LastName", ¬
columnArray[2] ="startDate", columnArray[3] = "salary"
```

MMDB.getSPColumnListNamedParams()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of result set columns that are generated by a call to the specified stored procedure.

Arguments

connName, *statement*, *paramNameArray*, *paramValuesArray*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *statement* is the name of the stored procedure that returns the result set when it executes.

- *paramNameArray* is an array that contains a list of parameter names. You can use the MMDB.getSPParamsAsString() function to get the parameters of the stored procedure.
- *paramValuesArray* is an array that contains a list of design-time parameter test values. You can specify if the procedure requires parameters when it executes. If you have provided parameter names in *paramNameArray*, specify the parameter values in the same order as their corresponding parameter names appear in *paramNameArray*. If you did not provide *paramNameArray*, specify the values in the order in which the stored procedure expects them.

Returns

An array of strings that represents the list of columns. This function returns an error if the SQL statement or the connection string is invalid.

Example

The following code could return a list of result set columns that are generated from the executed stored procedure, getNewEmployeesMakingAtLeast:

```
var paramNameArray = new Array("startDate", "salary")
var paramValueArray = new Array("2/1/2000", "50000")
var columnArray = MMDB.getSPColumnListNamedParams("EmpDB", ¬
"getNewEmployeesMakingAtLeast", paramNameArray, paramValueArray)
```

The following values are returned:

```
columnArray[0] = "EmpID", columnArray[1] = "LastName",¬
columnArray[2] ="startDate", columnArray[3] = "salary"
```

MMDB.getSPParameters()

Availability

Dreamweaver MX

Description

Returns an array of parameter objects for a named procedure.

Arguments

connName, procName

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *procName* is the name of the procedure.

Returns

An array of parameter objects, each specifying the following set of properties:

Property Name	Description
name	Name of the parameter (for example, `@@lolimit`)
datatype	Datatype of the parameter (for example, `smallmoney`)
direction	Direction of the parameter: `1`– The parameter is used for input only. `2`– The parameter is used for output only. In this case, you pass the parameter by reference and the method places a value in it. You can use the value after the method returns. `3`– The parameter is used for both input and output. `4`– The parameter holds a return value.

Example

```
var paramNameObjs = MMDB.getSPParameters(connName,procName);
for (i = 0; i < paramNameObjs.length; i++)
{
  var paramObj = paramNameObjs[i];
  var tooltiptext = paramObj.datatype;
  tooltiptext+=" ";
  tooltiptext+=GetDirString(paramObj.directiontype);
}
```

MMDB.getSPParamsAsString()

Availability

Dreamweaver UltraDev 1

Description

Gets a comma-delimited string that contains the list of parameters that the stored procedure takes.

Arguments

`connName`, *`procName`*

- *`connName`* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *`procName`* is the name of the stored procedure.

Returns

A comma-delimited string that contains the list of parameters that the stored procedure requires. The parameters' names, direction, and data type are included, separated by semicolons (;).

Example

The code `MMDB.getSPParamsAsString ("EmpDB","getNewEmployeesMakingAtLeast")` could return a string of form name `startDate;direction:in;datatype:date, salary;direction:in;datatype:integer`

In this example, the stored procedure, `getNewEmployeesMakingAtLeast`, has two parameters: `startDate` and `Salary`. For `startDate`, the direction is `in` and the data type is `date`. For `salary`, the direction is `in` and the data type is `date`.

MMDB.getTables()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of all the tables that are defined for the specified database. Each table object has three properties: `table`, `schema`, and `catalog`. `Table` is the name of the table, `schema` is the name of the schema that contains the table, and `catalog` is the catalog that contains the table.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

An array of objects where each object has three properties: `table`, `schema`, and `catalog`.

Example

The statement `MMDB.getTables ("EmpDB");` might produce an array of two objects. The first object's properties might be similar to the following example:

```
object1[table:"Employees", schema:"personnel", catalog:"syscat"]
```

The second object's properties might be similar to the following example:

```
object2[table:"Departments", schema:"demo", catalog:"syscat2"]
```

MMDB.getViews()

Availability

Dreamweaver UltraDev 4

Description

Gets a list of all the views that are defined for the specified database. Each view object has `catalog`, `schema`, and `view` properties. `Catalog` or `schema` is used for restricting or filtering the number of views that pertain to an individual schema name or catalog name that is defined as part of the connection information.

Arguments

connName

`connName` is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.

Returns

An array of view objects where each object has three properties: `catalog`, `schema`, and `view`.

Example

The following example returns the views for a given connection value, `CONN_LIST.getValue()`:

```
var viewObjects = MMDB.getViews(CONN_LIST.getValue())
for (i = 0; i < viewObjects.length; i++)
{
  thisView = viewObjects[i]
  thisSchema = Trim(thisView.schema)
  if (thisSchema.length == 0)
  {
    thisSchema = Trim(thisView.catalog)
  }
  if (thisSchema.length > 0)
  {
    thisSchema += "."
  }
  views.push(String(thisSchema + thisView.view))
}
```

MMDB.showResultset()

Availability

Dreamweaver UltraDev 1

Description

Displays a dialog box that has the results of executing the specified SQL statement. The dialog box displays a tabular grid where the header shows the column information and data of the result set that is generated by the executed stored procedure. If the connection string or the SQL statement is invalid, an error appears. You can use this function to verify the validity of the SQL statement.

Arguments

connName, *SQLstatement*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *SQLstatement* is the SQL `SELECT` statement.

Returns

Nothing. This function returns an error if the SQL statement or the connection string is invalid.

Example

The following code displays the results of the executed SQL statement:

```
MMDB.showResultset("EmpDB","Select EmpName,EmpFirstName,Age ¬
from Employees")
```

MMDB.showSPResultset()

Availability

Dreamweaver UltraDev 1

Description

Displays a dialog box that has the results of executing the specified stored procedure. The dialog box displays a tabular grid where the header shows the column information and data of the result set that is generated by the executed stored procedure. If the connection string or the stored procedure is invalid, an error appears. You can use this function to verify the validity of the stored procedure.

Arguments

connName, *procName*, *paramValuesArray*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *procName* is the name of the stored procedure to execute.
- *paramValuesArray* is an array that contains a list of design-time parameter test values. Specify the parameter values in the order in which the stored procedure expects them. You can use the `MMDB.getSPParamsAsString()` function to get the parameters of the stored procedure.

Returns

Nothing. This function returns an error if the SQL statement or the connection string is invalid.

Example

The following code displays the results of the executed stored procedure:

```
var paramValueArray = new Array("2/1/2000", "50000")
MMDB.showSPResultset("EmpDB", "getNewEmployeesMakingAtLeast", ¬
paramValueArray)
```

MMDB.showSPResultsetNamedParams()

Availability

Dreamweaver UltraDev 1

Description

Displays a dialog box that has the results of executing the specified stored procedure. The dialog box displays a tabular grid where the header shows the column information and data of the result set that is generated by the executed stored procedure. If the connection string or the stored procedure is invalid, an error appears. You can use this function to verify the validity of the stored procedure. This function differs from `MMDB.showSPResultset()` because you can specify the parameter values by name instead of the order in which the stored procedure expects them.

Arguments

connName, *procName*, *paramNameArray*, *paramValuesArray*

- *connName* is a connection name that is specified in the Connection Manager. It identifies the connection string that Dreamweaver should use to make a database connection to a live data source.
- *procName* is the name of the stored procedure that returns the result set when it executes.

- *paramNameArray* is an array that contains a list of parameter names. You can use the `MMDB.getSPParamsAsString()` function to get the parameters of the stored procedure.
- *paramValuesArray* is an array that contains a list of design-time parameter test values.

Returns

Nothing. This function returns an error if the SQL statement or the connection string is invalid.

Example

The following code displays the results of the executed stored procedure:

```
var paramNameArray = new Array("startDate", "salary")
var paramValueArray = new Array("2/1/2000", "50000")
MMDB.showSPResultsetNamedParams("EmpDB","getNewEmployees¬
MakingAtLeast", paramNameArray, paramValueArray)
```

CHAPTER 28
The Database Connectivity API

As a developer, you can create a new connection types and corresponding dialog boxes for new or existing server models. However, when a user sets up a site to start building pages, he or she creates a new connection object after selecting a particular type of connection that you created.

The user can select your new connection type in several ways:

- On the Application Building panel, the user can click the plus (+) button and select Recordset. In the Recordset dialog box, the user can expand the Connection list box.
- On the Database tab of the Databases panel, the user can click plus (+) button and select Data Source Name.

To develop a new connection type:

1 Create the layout for the connection dialog box.

Create an HTML file that lays out the user interface (UI) for your connection dialog box. Name this file using the name of the connection (for example myConnection.htm). For information on how to create a dialog box, see "Adding Custom Server Behaviors" in Book 8, Making Pages Dynamic, in *Getting Started with Dreamweaver MX*.

Make sure this HTML file includes the JavaScript implementation file that you define in Step "Create a JavaScript file that implements at least the following elements:" on page 338, as shown in the following example:

```
<head>
  <script SRC="../myConnectionImpl.js"></script>
</head>
```

Store this HTML file, which defines your connection dialog box, in the Configuration/Connections/*server-model*/*platform* folder.

server-model	is the folder that is associated with the document type (such as asp_js) of the currently open page.
platform	is either `Win` or `Mac`.

For example, the default ADO connection dialog box for an ASP JavaScript document on a Windows platform is stored in the ASP_Js/Win folder and is named Connection_ado_conn_string.htm.

Note: At runtime, Dreamweaver MX dynamically builds the list of connection types that are available to the user from the collection of dialog boxes that are in the ASP_Js/Win folder.

In the Configuration/ServerModels folder, there are .htm files that define each server model. Inside each of these HTML files is a function named `getServerModelFolderName()`, which returns the name of the folder that is associated with the server model. The following example shows the function for the ASP JavaScript document type:

```
function getServerModelFolderName()
{
  return "ASP_JS";
}
```

You can also look at the MMDocumentTypes.xml file, which is located in the Configuration/DocumentTypes folder, to determine the mapping between server models and document types.

2 Create a JavaScript file that implements at least the following elements:

Element	Description	Examples
A set of variables	Each defines a specific connection property	Type of connection, data source name, and so on
A set of buttons	Each button appears in the connection dialog box	Test, Help, and so on (OK and Cancel are automatically included)

Connectivity functions	Together, these functions define the Connectivity API	`findConnection()`
		`applyConnection()`
		`inspectConnection()`

You can choose any name for this implementation file but it must have a .js extension (for example, myConnectionImpl.js). You can store this implementation file on either your local or a remote computer. You might want to store your implementation file in the appropriate subfolder within the Configuration/Connections folder.

Note: The HTML file that you defined in Step "Create the layout for the connection dialog box." on page 337 must include this connection type implementation file.

Unless you need to define connection parameters other than the ones provided in the standard connection_includefile.edml file, these two steps are the minimum to create a new connection dialog box.

Note: The title of the dialog box that the user sees is in the `<title>` tag, which is specified in the HTML document.

The functions listed in the next section let you create a connection dialog box. Along with implementing the calls for generating include files for the user, you might need to register your connectivity type within the server model section of the connection XML file.

For information about the Database Connectivity API that is associated with creating a new connection, see ""Database connection functions" on page 312".

The Connection API

To create a new type of connection, including the dialog box with which users interact, you must implement the following three functions: `findConnection()`, `inspectConnection()`, and `applyConnection()`. You write these three functions and include them in the .js implementation file that is associated with your new connection type (see Step "Create a JavaScript file that implements at least the following elements:" on page 338 above).

The `applyConnection()` function returns an HTML source within an include file. You can see examples of the HTML source in the "The generated include file" on page 341. The `findConnection()` function takes the HTML source and extracts its properties. You can implement `findConnection()` to use the search patterns in XML files to extract the information that returns from `applyConnection()`. For an example of such an implementation, see the following two JavaScript files:

`connection_ado_conn_string.js`	Located in Configuration/Connections/ASP_Js folder
`connection_common.js`	Located in Configuration/Connections/Shared folder

When the user opens a site, Dreamweaver goes through each file in the Connections folder, opens it, and passes the contents to `findConnection()`. If the contents of a file match the criteria for a valid connection, `findConnection()` returns a connection object. Dreamweaver then lists all the connection objects in the Database Explorer panel.

When the user opens a connection dialog box and chooses to create a new connection or duplicate or edit an existing connection, Dreamweaver calls `inspectConnection()` and passes back the same connection object that `findConnection()` created. In this way, Dreamweaver can populate the dialog box with the connection information.

When the user clicks OK in a connection dialog box, Dreamweaver calls `applyConnection()` to build the HTML, which is placed in the connection include file that is located in the Configuration/Connections folder. The `applyConnection()` function returns an empty string that indicates there is an error in one of the fields and the dialog box should not be closed. The include file has the default file extension type for the current server model.

When the user adds to the page a server behavior that uses the connection, such as a record set or a stored procedure, Dreamweaver adds a statement to the page that includes the connection include file.

findConnection()

Availability

Dreamweaver UltraDev 4

Description

Dreamweaver calls this function to detect a connection in the specified HTML source and to parse the connection parameters. If the contents of this source file match the criteria for a valid connection, `findConnection()` returns a connection object; otherwise, this function returns a null value.

Argument

htmlSource

`htmlSource` is the HTML source for a connection.

Returns

A connection object that provides values for a particular combination of the properties that are listed in the following table. The properties for which this function returns a value depends on the document type.

Property	Description
`name`	Name of the connection
`type`	If `useHTTP` is `false`, indicates which DLL to use for connecting to database at runtime
`string`	Runtime connection string. For ADO, it is a string of connection parameters; for JDBC, it is a connection URL
`dsn`	Data source name used for ODBC or Cold Fusion runtime connections
`driver`	Name of a JDBC driver used at runtime
`username`	Name of the user used for the runtime connection
`password`	Password used for the runtime connection
`designtimeString`	Design-time connection string (see `string`)
`designtimeDsn`	Design-time data source name (see `dsn`)
`designtimeDriver`	Name of a JDBC driver used at design time
`designtimeUsername`	Name of the user used for the design-time connection
`designtimePassword`	Password used for the design-time connection
designtimeType	Design time connection type
usesDesigntimeInfo	When `false`, Dreamweaver uses runtime properties at design time; otherwise, Dreamweaver uses design-time properties
`useHTTP`	String containing either `true` or `false`; which specifies whether to use HTTP connection at design time or use DLL
`includePattern`	Regular expression used to find the file include statement on the page during Live Data and Preview In Browser
`variables`	Object with a property for each page variable which is set to its corresponding value. This object is used during Live Data and Preview In Browser
`catalog`	String containing a database identifier that restricts the amount of metadata that appears
`schema`	String containing a database identifier that restricts the amount of metadata that appears
`filename`	Name of the dialog box used to create the connection

If a connection is not found in htmlSource, a null value returns.

Note: Developers can add custom properties (for example, metadata) to the HTML source, which `applyConnection()` returns along with the standard properties.

inspectConnection()

Availability

Dreamweaver UltraDev 4

Description

Dreamweaver calls this function, when the user edits an existing connection, to initialize the dialog box data for defining a connection. In this way, Dreamweaver can populate the dialog box with the appropriate connection information.

Argument

parameters

parameters is the same object that `findConnection()` returns.

Returns

Nothing.

applyConnection()

Availability

Dreamweaver UltraDev 4

Description

Dreamweaver calls this function when the user clicks OK in the connection dialog box. The `applyConnection()` function generates the HTML source for a connection. Dreamweaver writes the HTML to the Configuration/Connections/*connection-name.ext* include file, where *connection-name* is the name of your connection (see Step "Create the layout for the connection dialog box." on page 337), and *ext* is the default extension that is associated with the server model.

Arguments

None.

Returns

The HTML source for a connection. Dreamweaver also closes the connection dialog box. If a field validation error occurs, `applyConnection()` displays an error message and returns an empty string to indicate that the dialog box should remain open.

The generated include file

The include file that `applyConnection()` generates declares all the properties of a connection. The filename for the include file is the connection name that has the file extension defined for the server model that is associated with the current site.

Note: Connections are shared, so set the `allowMultiple` value to `false`. This ensures that the connection file is included in the document only once and that the server script remains in the page if any other server behaviors use it.

The following sections illustrate some sample include files that `applyConnection()` generates for various default server models.

Note: To create a new connection include file format, you need to define a new `.edml` mapping file, which should be similar to connection_includefile.edml, as shown in "The definition file for your connection type" on page 343.

ASP JavaScript

The ASP and JavaScript include file should be named MyConnection1.asp, where MyConnection1 is the name of the connection. The following sample is an include file for an ADO connection string:

```
<%
  // Filename="Connection_ado_conn_string.htm"
  // Type="ADO"
  // HTTP="true"
  // Catalog=""
  // Schema=""
  var MM_MyConnection1_STRING = "dsn=pubs";
%>
```

The server behavior file includes this connection by using the relative file include statement, as shown in the following example:

```
  <!--#include file="../Connections/MyConnection1.asp"-->
```

ColdFusion

When you use UltraDev 4 ColdFusion, Dreamweaver MX relies on a ColdFusion include file to get a list of data sources.

Note: For regular Dreamweaver MX ColdFusion, Dreamweaver MX ignores any include files and, instead, makes use of RDS to retrieve the list of data sources from ColdFusion.

The UltraDev 4 ColdFusion include file should be named MyConnection1.cfm, where MyConnection1 is the name of your connection. The following example shows the include file for a ColdFusion connection to a product table.

```
<!-- FileName="Connection_cf_dsn.htm" "dsn=products" -->
<!-- Type="ADO" -->
<!-- Catalog="" -->
<!-- Schema="" -->
<!-- HTTP="false" -->
<CFSET MM_MyConnection1_DSN      = "products">
<CFSET MM_MyConnection1_USERNAME  = "">
<CFSET MM_Product_USERNAME        = "">
<CFSET MM_MyConnection1_PASSWORD  = "">
```

The server behavior file includes this connection by using the `cfinclude` statement, as shown in the following example:

```
<cfinclude template="Connections/MyConnection1.cfm">
```

JSP

The JSP include file should be named MyConnection1.jsp, where MyConnection1 is the name of your connection. The following sample is the include file for a JDBC connection to a database:

```
<%
  // Filename="Connection_jdbc_conn1.htm"
  // Type="JDBC"
  // HTTP="false"
  // Catalog=""
  // Schema=""
  String MM_MyConnection1_DRIVER    = "com.inet.tds.TdsDriver";
  String MM_MyConnection1_USERNAME  = "testadmin";
  String MM_MyConnection1_PASSWORD  = "velcro";
  String MM_MyConnection1_URL       = "jdbc:server:test-3:1433?database=pubs";
%>
```

The server behavior file includes this connection by using the relative file include statement, as shown in the following example:

```
<%@ include file="Connections/MyConnection1.jsp" %>
```

The definition file for your connection type

For each server model, there is a connection_includefile.edml file that defines the connection type and maps the properties that are defined in the include file to elements in the Dreamweaver MX interface.

Dreamweaver provides, by default, seven definition files, one for each of the predefined server models, as listed in the following table.

Server Model	Subfolder within the Configuration/Connections folder
ASP JavaScript	ASP_Js
ASP.NET CSharp	ASP.NET_Csharp
ASP.NET VBScript	ASP.NET_VB
ASP VBScript	ASP_Vbs
ColdFusion	ColdFusion
JavaServer Page	JSP
PHP MySql	PHP_MySql

Dreamweaver uses the `quickSearch` and `searchPattern` parameters to recognize connection blocks and the `insertText` parameter to create connection blocks. For more information on XML and regular expression search patterns, see “Server Behaviors” on page 145.

Note: If you change the format of your include file or define an include file for a new server model, you need to map the connection parameters with the Dreamweaver UI, Live Data, and Preview In Browser. The following sample XML file, which is associated with the default ASP JS server model, maps all connection page variables with their respective live values before sending the page to the server. For more information on XML and regular expression search patterns, see "Server Behaviors" on page 145.

```
<participant name="connection_includefile" version="5.0">
  <quickSearch>
    <![CDATA[// HTTP=]]></quickSearch>
  <insertText location="">
<![CDATA[<%
// FileName="@@filename@@"
// Type="@@type@@" @@designtimeString@@
// DesigntimeType="@@designtimeType@@"
// HTTP="@@http@@"
// Catalog="@@catalog@@"
// Schema="@@schema@@"
var MM_@@cname@@_STRING = @@string@@
%>
]]>
  </insertText>
  <searchPatterns whereToSearch="directive">
    <searchPattern paramNames="filename">
      <![CDATA[/\/\/\s*FileName="([^"]*)"/]]></searchPattern>
    <searchPattern paramNames="type,designtimeString">
      <![CDATA[/\/\/\s+Type="(\w*)"([^\r\n]*)/]]></searchPattern>
    <searchPattern paramNames="designtimeType" isOptional="true">
      <![CDATA[/\/\/\s*DesigntimeType="(\w*)"/]]></searchPattern>
    <searchPattern paramNames="http">
      <![CDATA[/\/\/\s*HTTP="(\w+)"/]]></searchPattern>
    <searchPattern paramNames="catalog">
      <![CDATA[/\/\/\s*Catalog="(\w*)"/]]></searchPattern>
    <searchPattern paramNames="schema">
      <![CDATA[/\/\/\s*Schema="(\w*)"/]]></searchPattern>
    <searchPattern paramNames="cname,string">
      <![CDATA[/var\s+MM_(\w*)_STRING\s*=\s*([^\r\n]+)/]]></searchPattern>
  </searchPatterns>
</participant>
```

Tokens in an .edml file—such as `@@filename@@` in this example—map values in the include file to properties of a connection object. You set the properties of connection objects in the .js implementation file.

All the default connection dialog boxes that come with Dreamweaver MX use the connection_includefile.edml mapping file. To let Dreamweaver MX find this file, its name is set in the .js implementation file as shown in the following example:

```
var PARTICIPANT_FILE = "connection_includefile";
```

When you create a custom connection type, you can use any mapping file in your custom dialog boxes. If you create a mapping file, you can use a name other than connection_includefile for your .edml file. If you use a different name, you need to use this name in your .js implementation file when you specify the value that is assigned to the `PARTICIPANT_FILE` variable, as shown in the following example:

```
var PARTICIPANT_FILE = "myConnection_mappingfile";
```

CHAPTER 29
The JavaBeans API

This chapter explains the APIs for JavaBeans, the `MMJB*()` functions are JavaScript hooks that invoke Java introspection calls for JavaBeans support. These functions get class names, methods, properties, and events from the JavaBeans, which can appear in your Dreamweaver MX user interface. To use these JavaScript functions and let Dreamweaver access your JavaBeans, the JavaBeans must reside in the Configuration/Classes folder.

Note: `packageName.className` is a single string input.

MMJB.getProperties()

Availability

Dreamweaver UltraDev 4

Description

Introspects the JavaBeans class and returns its properties.

Arguments

`packageName.className`

`packageName.className` is the name of the class, which is part of the class path. It must be a `.jar` or `.zip` Java archive that resides in your system class path or a `.class` file that is installed in the Configuration/Classes folder.

Returns

A string array of the JavaBeans properties; an error returns an empty array.

MMJB.getMethods()

Availability

Dreamweaver UltraDev 4

Description

Introspects the JavaBeans class and returns its methods.

Arguments

`packageName.className`

`packageName.className` is the package name of the class, which is part of the class path. It must be a Java `.jar` or `.zip` Java archive.

Returns

A string array of the JavaBeans methods; an error returns an empty array.

MMJB.getEvents()

Availability

Dreamweaver UltraDev 4

Description

Introspects the JavaBeans class and returns its events.

Arguments

packageName.className

packageName.className is the package name of the class, which is part of the class path. It must be a Java `.jar` or `.zip` Java archive.

Returns

A string array of the JavaBeans events; an error returns an empty array.

MMJB.getIndexedProperties()

Availability

Dreamweaver UltraDev 4

Description

Introspects the JavaBeans class and returns its indexed properties, which are design patterns that behave the same way as collections.

Arguments

packageName.className

packageName.className is the package name of the class, which is part of the class path. It must be a Java `.jar` or `.zip` Java archive.

Returns

A string array of the JavaBeans' indexed properties; an error returns an empty array.

MMJB.getClasses()

Availability

Dreamweaver UltraDev 4

Description

Reads all the JavaBeans class names from the Configuration/Classes folder.

Arguments

None.

Returns

A string array of class names that are located in Configuration/Classes folder; an error returns an empty array.

MMJB.getClassesFromPackage()

Availability

Dreamweaver UltraDev 4

Description

Reads all the JavaBeans classes from the package.

Arguments

packageName.pathName

packageName.pathName is the path to the package. It must be a Java `.jar` or `.zip` Java archive. For example, `C:/jdbcdrivers/Una2000_Enterprise.zip`.

Returns

A string array of class names inside the particular `.jar` or `.zip` Java file; an error returns an empty array.

MMJB.getErrorMessage()

Availability

Dreamweaver UltraDev 4

Description

Gets the last error message from Dreamweaver that occurred while using the MMJB interface.

Arguments

None.

Returns

A string of the Dreamweaver message from the last error.

CHAPTER 30
The Source Control Integration API

The Source Control Integration API lets you write shared libraries to extend the Macromedia Dreamweaver MX Check in/Check out feature using source control systems (such as Sourcesafe, CVS, or Sitespring).

You must support a minimum set of API functions that you must support for Dreamweaver to integrate with a source control system.

Your library resides in the Configuration\SourceControl folder.

When Dreamweaver starts, it loads each library. Dreamweaver determines which features the library supports by calling `GetProcAddress()` for each API function. If an address does not exist, Dreamweaver assumes the library does not support the API. If the address exists, Dreamweaver uses the library's version of the function to support the functionality. When a Dreamweaver user defines or edits a site and then chooses the Web Server SCS tab, the choices that correspond to the DLLs that loaded from the Configuration/SourceControl folder appear (in addition to the standard items) on the tab.

To add custom items to the Site > Source Control menu, add the following code in the Site menu in the menus.xml file:

```
<menu name="Source Control" id="DWMenu_MainSite_Site_Source¬
Control"><menuitem dynamic name="None"file="Menus/MM/¬
File_SCSItems.htm" id="DWMenu_MainSite_Site_NewFeatures_¬
Default" />
</menu>
```

Integration with Dreamweaver

When a Dreamweaver user chooses server connection, file transfer, or Design Notes features, Dreamweaver calls the DLL's version of the corresponding API function (`Connect()`, `Disconnect()`, `Get()`, `Put()`, `Checkin()`, `Checkout()`, `Undocheckout()`, and `Synchronize()`). The DLL is responsible for handling the request, including displaying dialog boxes that gather information or let the user interact with the DLL. The DLL is also responsible for displaying information or error messages.

The source control system can optionally support Design Notes and Check In/Check Out. The Dreamweaver user enables Design Notes in source control systems by choosing the Design Notes tab in the Edit Sites dialog box and checking the box that enables the feature; this is the same way to enable Design Notes with FTP and LAN. If the source control system does not support Design Notes and the user wants to use this feature, Dreamweaver transports Design Note (.mno) files to maintain the Design Notes (as it does with FTP and LAN).

Check In/Check Out is treated differently than the Design Notes feature; if the source control system supports it, the user cannot override its use from the Design Notes dialog box. If the user tries to override the source control system, an error message appears.

Adding source control system functionality

You can add source control system functionality to Dreamweaver by writing a `GetNewFeatures` handler that returns a set of menu items and corresponding C functions. For example, if you write a Sourcesafe library and want to let Dreamweaver users see the history of a file, you can write a `GetNewFeatures` handler that returns the History menu item and the C function name of `history`. Then, in Windows, when the user right-clicks a file, the History menu item is one of the items on the menu. If a user chooses the History menu item, Dreamweaver calls the corresponding function, passing the selected file(s) to the DLL. The DLL displays the History dialog box so the user can interact with it in the same way as Sourcesafe.

The Source Control Integration API required functions

The Source Control Integration API has required and optional functions. The functions listed in this section are required.

bool SCS_GetAgentInfo()

Description

Asks the DLL to return its name and description, which appear in the Edit Sites dialog box. The name appears in the Server Access pop-up menu (for example, sourcesafe, webdav, perforce) and description just below the pop-up menu.

Arguments

`char name[32]`, *`char version[32]`*, *`char description[256]`*, *`const char*dwAppVersion`*

- *`name`* is the name of the source control system. The name appears in the combo box for selecting a source control system in the Source Control tab of the Edit Sites dialog box. The name can be a maximum of 32 characters.
- *`version`* is a string that indicates the version of the DLL. Version appears in the Source Control tab of the Edit Sites dialog box. The version can be a maximum of 32 characters.
- *`description`* is a string that indicates the description of the source control system. Description appears in the Source Control tab of the Edit Sites dialog box. The description can be a maximum of 256 characters.
- *`dwAppVersion`* is a string that indicates the version of Dreamweaver that is calling the DLL. The DLL can use this string to determine the version and language of Dreamweaver.

Returns

`true` if successful; `false` otherwise.

bool SCS_Connect()

Description

Connects the user to the source control system. If the DLL does not have login information, the DLL is responsible for displaying a dialog box to prompt the user for the information and for storing the data for later use.

Arguments

*void **connectionData*, *const char siteName[64]*

- *connectionData* is a handle to the data that the agent wants Dreamweaver to pass to it when calling other API functions.
- *siteName* is a string that points to the name of the site. The site name can be a maximum of 64 characters.

Returns

`true` if successful; `false` otherwise.

bool SCS_Disconnect()

Description

Disconnects the user from the source control system.

Arguments

*void *connectionData*

connectionData is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.

Returns

`true` if successful; `false` otherwise.

bool SCS_IsConnected()

Description

Determines the state of the connection.

Arguments

*void *connectionData*

connectionData is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.

Returns

`true` if connected; `false` otherwise.

int SCS_GetRootFolderLength()

Description

Returns the length of the name of the root folder.

Arguments

*void *connectionData*

- *connectionData* is a pointer to the agent's data that was passed into Dreamweaver during the `Connect()` call.

Returns

An integer that indicates the length of the name of the root folder. If the function returns `< 0`, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

bool SCS_GetRootFolder()

Description

Returns the name of the root folder.

Arguments

*void *connectionData*, *char remotePath[]*, *const int folderLen*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is a buffer where the full remote path of the root folder is stored.
- *folderLen* is an integer that indicates the length of *remotePath*. This is the value that *GetRootFolderLength* returns.

Returns

true if successful; false otherwise.

int SCS_GetFolderListLength()

Description

Returns the number of items in the passed-in folder.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is the full path and name of the remote folder that the DLL checks for the number of items.

Returns

An integer that indicates the number of items in the current folder. If the function returns < 0, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

bool SCS_GetFolderList()

Description

Returns a list of files and folders in the passed-in folder, including pertinent information such as modified date, size, and whether the item is a folder or file.

Arguments

*void *connectionData*, *const char *remotePath*, *itemInfo itemList*[], *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is the path name of the remote folder that the DLL checks for the number of items.

- *itemList* is a preallocated list of itemInfo structures:

name	char[256]	name of file or folder
isFolder	bool	true if folder, false if file
month	int	month component of mod date 1-12
day	int	day component of mod date 1-31
year	int	year component of mod date 1900+
hour	int	hour component of mod date 0-23
minutes	int	minute component of mod date 0-59
seconds	int	second component of mod date 0-59
type	char[256]	type of file (if not set by DLL, DW will use file extension to determine type, as it does now)
size	int	in bytes

- *numItems* is the number of items that are allocated for the itemList (returned from GetFolderListLength).

Returns

true if successful; false otherwise.

bool SCS_Get()

Description

Gets a list of files or folders and stores them locally.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char *localPathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of the remote files or folders to retrieve, which is specified as complete paths and names.
- *localPathList* is a mirrored list of local file or folder path names.
- *numItems* is the number of items in each list.

Returns

true if successful; false otherwise.

bool SCS_Put()

Description

Puts a list of local files or folders into the source control system.

Arguments

*void *connectionData*, *const char *localPathList[]*, *const char *remotePathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *localPathList* is the list of local file or folder path names to put into the source control system.
- *remotePathList* is a mirrored list of remote file or folder path names.
- *numItems* is the number of items in each list.

Returns

true if successful; false otherwise.

bool SCS_NewFolder()

Description

Creates a new folder.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is the full path name of the remote folder the DLL creates.

Returns

true if successful; false otherwise.

bool SCS_Delete()

Description

Deletes a list of files or folders from the source control system.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to delete.
- *numItems* is the number of items in *remotePathList*.

Returns

true if successful; false otherwise.

bool SCS_Rename()

Description

Renames or moves a file or folder, depending on the values that are specified for *oldRemotePath* and *newRemotePath*. For example, if *oldRemotePath* equals "$/folder1/file1" and *newRemotePath* equals "$/folder1/renamefile1", file1 is renamed renamefile1 and is located in folder1.

If *oldRemotePath* equals "$/folder1/file1" and *newRemotePath* equals "$/folder1/subfolder1/file1", file1 is moved to the subfolder1 directory.

To find out if an invocation of this function is a move or a rename, check the parent paths of the two input values; if they are the same, the operation is a rename.

Arguments

*void *connectionData*, *const char *oldRemotePath*, *const char *newRemotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *oldRemotePath* is a remote file or folder path name to rename.
- *newRemotePath* is the remote path name of the new name for the file or folder.

Returns

true if successful; false otherwise.

bool SCS_ItemExists()

Description

Determines whether a file or folder exists on the server.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is a remote file or folder path name.

Returns

true if exists, false otherwise.

The Source Control Integration API optional functions

The Source Control Integration API has required and optional functions. The functions in this section are optional.

bool SCS_GetConnectionInfo()

Description

Displays a dialog box to let the user change or set the connection information for this site. Does not make the connection. This function is called when the user clicks the Settings button in the Remote Info section of the Edit Sites dialog box.

Arguments

*void **connectionData*, *const char siteName[64]*

- *connectionData* is a handle to data that the agent wants Dreamweaver to pass it when calling other API functions.
- *siteName* is a string that points to the name of the site. The name cannot exceed 64 characters.

Returns

true if successful; false otherwise.

bool SCS_SiteDeleted()

Description

Notifies the DLL that the site has been deleted or that the site is no longer tied to this source control system. It indicates that the source control system can delete its persistent information for this site.

Arguments

const char siteName[64]

siteName is a string that points to the name of the site. The name cannot exceed 64 characters.

Returns

`true` if successful; `false` otherwise.

bool SCS_SiteRenamed()

Description

Notifies the DLL when the user has renamed the site so that it can update its persistent information about the site.

Arguments

const char oldSiteName[64], *const char newSiteName[64]*

- *oldSiteName* is a string that points to the original name of the site before it was renamed. The name cannot exceed 64 characters.
- *newSiteName* is a string that points to the new name of the site after it was renamed. The name cannot exceed 64 characters.

Returns

`true` if successful; `false` otherwise.

int SCS_GetNumNewFeatures()

Description

Returns the number of new features to add to Dreamweaver (for example, File History, Differences, and so on).

Arguments

None.

Returns

An integer that indicates the number of new features to add to Dreamweaver. If the function returns `< 0`, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

bool SCS_GetNewFeatures()

Description

Returns a list of menu items to add to the Dreamweaver main and context menus. For example, the Sourcesafe DLL can add History and File Differences to the main menu.

Arguments

char menuItemList[][32], *scFunction functionList[], scFunction enablerList[]*, *const int numNewFeatures*

- *menuItemList* is a string list that is populated by the DLL; it specifies the menu items to add to the main and context menus. Each string can contain a maximum of 32 characters.
- *functionList* is populated by the DLL; it specifies the routines in the DLL to call when the user chooses the corresponding menu item.
- *enablerList* is populated by the DLL; it specifies the routines in the DLL to call when Dreamweaver needs to determine whether the corresponding menu item is enabled.
- *numNewFeatures* is the number of items being added by the DLL; this value is retrieved from the `GetNumNewFeatures()` call.

The following function signature defines the functions and enablers that passed to the `SCS_GetNewFeatures()` call in the `functionlist` and `enablerList` arguments.

```
bool (*scFunction)(void *connectionData, const char *remotePathList[],
  const char *localPathList[], const int numItems)
```

Returns

`true` if successful; `false` otherwise.

bool SCS_GetCheckoutName()

Description

Returns the check-out name of the current user. If it is unsupported by the source control system and this feature is enabled by the user, this function uses the Dreamweaver internal Check in/Check out functionality, which transports .lck files to and from the source control system.

Arguments

*void *connectionData*, *char checkOutName[64]*, *char emailAddress[64]*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *checkOutName* is the check-out name of the current user.
- *emailAddress* is the e-mail address of the current user.

Returns

`true` if successful; `false` otherwise.

bool SCS_Checkin()

Description

Checks a list of local files or folders into the source control system. The DLL is responsible for making the file read-only. If it is unsupported by the source control system and this feature is enabled by the user, this function uses the Dreamweaver internal Check in/Check out functionality, which transports .lck files to and from the source control system.

Arguments

*void *connectionData*, *const char *localPathList[]*, *const char *remotePathList[]*, *bool successList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *localPathList* is a list of local file or folder path names to check in.
- *remotePathList* is a mirrored list of remote file or folder path names.
- *successList* is a list of Boolean values that are populated by the DLL to let Dreamweaver know which of the corresponding files are successfully checked in.
- *numItems* is the number of items in each list.

Returns

true if successful; false otherwise.

bool SCS_Checkout()

Description

Checks out a list of local files or folders from the source control system. The DLL is responsible for granting the privileges that let the file be writable. If it is unsupported by the source control system and this feature is enabled by the user, this function uses the Dreamweaver internal Check in/Check out functionality, which transports .lck files to and from the source control system.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char *localPathList[]*, *bool successList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to check out.
- *localPathList* is a mirrored list of local file or folder path names.
- *successList* is a list of Boolean values that are populated by the DLL to let Dreamweaver know which of the corresponding files are successfully checked out.
- *numItems* is the number of items in each list.

Returns

true if successful; false otherwise.

bool SCS_UndoCheckout()

Description

Undoes the check-out status of a list of files or folders. The DLL is responsible for making the file read-only. If it is unsupported by the source control system and this feature is enabled by the user, this function uses the Dreamweaver internal Check in/Check out functionality, which transports .lck files to/from the source control system.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char *localPathList[]*, *bool successList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePathList* is a list of remote file or folder path names on which to undo the check out.
- *localPathList* is a mirrored list of local file or folder path names.
- *successList* is a list of Boolean values that are populated by the DLL to let Dreamweaver know which corresponding files' check outs are successfully undone.
- *numItems* is the number of items in each list.

Returns

`true` if successful; `false` otherwise.

int SCS_GetNumCheckedOut()

Description

Returns the number of people who have a file checked out.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is the remote file or folder path name to check to see how many users have it checked out.

Returns

An integer that indicates the number of people who have the file checked out. If the function returns `< 0`, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

bool SCS_GetFileCheckoutList()

Description

Returns a list of people who have a file checked out. If the list is empty, no one has the file checked out.

Arguments

*void *connectionData*, *const char *remotePath*, *char checkOutList[][64]*, *char emailAddressList[][64]*, *const int numCheckedOut*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is the remote file or folder path name to check how many users have it checked out.
- *checkOutList* is a list of strings that corresponds to the users who have the file checked out. Each user string cannot exceed a maximum length of 64 characters.

- *emailAddressList* is a list of strings that corresponds to the users' e-mail addresses. Each e-mail address string cannot exceed a maximum length of 64 characters.
- *numCheckedOut* is the number of people who have the file checked out. This is returned from GetNumCheckedOut().

Returns

true if successful; false otherwise.

int SCS_GetErrorMessageLength()

Description

Returns the length of the DLL's current internal error message. This allocates the buffer that passes into the GetErrorMessage() function. This function should be called only if an API function returns false or <0, which indicates a failure of that API function.

Arguments

*void *connectionData*

connectionData is a pointer to the agent's data that was passed into Dreamweaver during the Connect() call.

Returns

An integer that represents the length of the error message.

bool SCS_GetErrorMessage()

Description

Returns the last error message. If you implement getErrorMessage(), Dreamweaver calls it each time one of your API functions returns false.

If a routine returns -1 or false, it indicates an error message should be available.

Arguments

*void *connectionData*, *char errorMsg[]*, *const int *msgLength*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *errorMsg* is a preallocated string for the DLL to fill in with the error message.
- *msgLength* is the length of the *errorMsg* buffer passed in.

Returns

true if successful; false otherwise.

int SCS_GetNoteCount()

Description

Returns the number of Design Note keys for the specified remote file or folder path. If unsupported by the source control system, Dreamweaver gets this information from the companion Design Note (.mno) file.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is the remote file or folder path name that the DLL checks for the number of attached Design Notes.

Returns

An integer that indicates the number of Design Notes that are associated with this file. If the function returns `< 0`, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

int SCS_GetMaxNoteLength()

Description

Returns the length of the largest Design Note for the specified file or folder. If it is unsupported by the source control system, Dreamweaver gets this information from the companion Design Note (.mno) file.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is the remote file or folder path name that the DLL checks for the maximum Design Note length.

Returns

An integer that indicates the size of the longest Design Note that is associated with this file. If the function returns `< 0`, Dreamweaver considers it an error and tries to retrieve the error message from the DLL, if supported.

bool SCS_GetDesignNotes()

Description

Retrieves key-value pairs from the meta information for the specified file or folder. If it is unsupported by the source control system, Dreamweaver retrieves the information from the corresponding Design Note (.mno) file.

Arguments

*void *connectionData*, *const char *remotePath*, *char keyList[][64]*, *char *valueList[]*, *bool showColumnList[]*, *const int noteCount*, *const int noteLength*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is the remote file or folder path name that the DLL checks for the number of items.
- *keyList* is a list of Design Note keys, such as `"Status"`.
- *valueList* is a list of Design Note values that correspond to the Design Note keys, such as `"Awaiting Signoff"`.

- *showColumnList* is a list of Boolean values that correspond to the Design Note keys, which indicate whether Dreamweaver can display the key as a column in the Site panel.
- *noteCount* is the number of Design Notes that are attached to a file or folder; the GetNoteCount() call returns this value.
- *noteLength* is the maximum length of a Design Note; this is the value that the GetMaxNoteLength() call returns.

Returns

true if successful; false otherwise.

bool SCS_SetDesignNotes()

Description

Stores the key-value pairs in the meta information for the specified file or folder. This replaces the set of meta information for the file. If it is unsupported by the source control system, Dreamweaver stores Design Notes in .mno files.

Arguments

*void *connectionData*, *const char *remotePath*, *const char keyList[][64]*, *const char *valueList[]*, *bool showColumnList[]*, *const int noteCount*, *const int noteLength*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is the remote file or folder path name that the DLL checks for the number of items.
- *keyList* is a list of Design Note keys, such as "Status".
- *valueList* is a list of Design Note values that corresponds to the Design Note keys, such as "Awaiting Signoff".
- *showColumnList* is a list of Boolean values that correspond to the Design Note keys, which indicate whether Dreamweaver can display the key as a column in the Site panel.
- *noteCount* is the number of Design Notes that are attached to a file or folder; this number lets the DLL know the size of the specified lists. If *noteCount* is 0, all the Design Notes are removed from the file.
- *noteLength* is the length of the largest Design note for the specified file or folder.

Returns

true if successful; false otherwise.

bool SCS_IsRemoteNewer()

Description

Checks each specified remote path to see if the remote copy is newer. If it is unsupported by the source control system, Dreamweaver uses its internal isRemoteNewer algorithm.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char*remotePathList[]*, *int remoteIsNewerList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to compare for newer status.
- *localPathList* is a mirrored list of local file or folder path names.
- *remoteIsNewerList* is a list of integers that are populated by the DLL to let Dreamweaver know which of the corresponding files is newer on the remote side. The following values are valid: 1 indicates the remote version is newer, -1 indicates the local version is newer, 0 indicates the versions are the same.
- *numItems* is the number of items in each list.

Returns

true if successful; false otherwise.

Enablers

If the optional enablers are not supported by the source control system or the application is not connected to the server, Dreamweaver determines when the menu items are enabled, based on the information it has about the remote files.

bool SCS_canConnect()

Description

Returns whether the Connect menu item should be enabled.

Arguments

None.

Returns

true if enabled, false otherwise.

bool SCS_canGet()

Description

Returns whether the Get menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char *localPathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to get.
- *localPathList* is a mirrored list of local file or folder path names.
- *numItems* is the number of items in each list.

Returns

true if enabled, false otherwise.

bool SCS_canCheckout()

Description

Returns whether the Checkout menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char *localPathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to check out.
- *localPathList* is a mirrored list of local file or folder path names.
- *numItems* is the number of items in each list.

Returns

true if enabled, false otherwise.

bool SCS_canPut()

Description

Returns whether the Put menu item should be enabled.

Arguments

*void *connectionData*, *const char *localPathList[]*, *const char *remotePathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *localPathList* is a list of local file or folder path names to put into the source control system.
- *remotePathList* is a mirrored list of remote file or folder path names to put into the source control system.
- *numItems* is the number of items in each list.

Returns

true if enabled, false otherwise.

bool SCS_canCheckin()

Description

Returns whether the Checkin menu item should be enabled.

Arguments

*void *connectionData*, *const char *localPathList[]*, *const char *remotePathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *localPathList* is a list of local file or folder path names to check in.

- *remotePathList* is a mirrored list of remote file or folder path names.
- *numItems* is the number of items in each list.

Returns

`true` if enabled, `false` otherwise.

bool SCS_CanUndoCheckout()

Description

Returns whether the Undo Checkout menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const char***localPathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePathList* is a list of remote file or folder path names to check out.
- *localPathList* is a list of the local file or folder path names to put to the source control system.
- *numItems* is the number of items in each list.

Returns

`true` if enabled, `false` otherwise.

bool SCS_canNewFolder()

Description

Returns whether the New Folder menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the `Connect()` call.
- *remotePath* is a remote file or folder path names that the user selected to indicate where the new folder will be created.

Returns

`true` if enabled, `false` otherwise.

bool SCS_canDelete()

Description

Returns whether the Delete menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePathList[]*, *const int numItems*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePathList* is a list of remote file or folder path names to delete.
- *numItems* is the number of items in each list.

Returns

true if enabled, false otherwise.

bool SCS_canRename()

Description

Returns whether the Rename menu item should be enabled.

Arguments

*void *connectionData*, *const char *remotePath*

- *connectionData* is a pointer to the agent's data that passed into Dreamweaver during the Connect() call.
- *remotePath* is the remote file or folder path names that can be renamed.

Returns

true if enabled, false otherwise.

bool SCS_BeforeGet()

Description

Dreamweaver calls this function before getting or checking out one or more files. This function lets your DLL perform one operation, such as adding a check-out comment, to a group of files.

Arguments

**connectionData*

**connectionData* is a pointer to the connection data.

Returns

A Boolean true if successful; false otherwise.

Example

To get a group of files, Dreamweaver makes calls to the DLL in the following order:

```
SCS_BeforeGet(connectionData);
SCS_Get(connectionData,remotePathList1,localPathList1,¬
successList1);
SCS_Get(connectionData,remotePathList2,localPathList2,¬
successList2);
SCS_Get(connectionData,remotePathList3,localPathList3,¬
successList3);
SCS_AfterGet(connectionData);
```

bool SCS_BeforePut()

Description

Dreamweaver calls this function before putting or checking in one or more files. This function lets your DLL perform one operation, such as adding a check-in comment, to a group of files.

Arguments

**connectionData*

**connectionData* is a pointer to the connection data.

Returns

A Boolean `true` if successful; `false` otherwise.

Example

To get a group of files, Dreamweaver makes calls to the DLL in the following order:

```
SCS_BeforePut(connectionData);
SCS_Put(connectionData,localPathList1,remotePathList1,¬
successList1);
SCS_Put(connectionData,localPathList2,remotePathList2,¬
successList2);
SCS_Put(connectionData,localPathList3,remotePathList3,¬
successList3);
SCS_AfterPut(connectionData);
```

bool SCS_AfterGet()

Description

Dreamweaver calls this function after getting or checking out one or more files. This function lets your DLL perform any operation after a batch get or check out, such as creating a summary dialog box.

Arguments

**connectionData*

**connectionData* is a pointer to the connection data.

Returns

A Boolean `true` if successful; `false` otherwise.

Example

See example in "bool SCS_BeforeGet()" on page 366.

bool SCS_AfterPut()

Description

Dreamweaver calls this function after putting or checking in one or more files. This function lets the DLL perform any operation after a batch put or check in, such as creating a summary dialog box.

Arguments

`*connectionData`

`*connectionData` is a pointer to the connection data.

Returns

`true` if successful; `false` otherwise.

Example

See example in "bool SCS_BeforePut()" on page 367.

Part IV
JavaScript API

Use any of the more than 600 core JavaScript functions available in Dreamweaver, which encapsulate the kinds of tasks users perform when creating or editing a document. You can use these functions to perform any task that the user can accomplish using menus, floating panels, property inspectors, the Site panel, or the Document window.

- Chapter 31, “The Dreamweaver JavaScript API”

CHAPTER 31
The Dreamweaver JavaScript API

The Macromedia Dreamweaver MX JavaScript API provides an extensive set of tools that are useful to extension developers. In addition to the standards-based Document Object Model (DOM) methods that are described in "The Dreamweaver Document Object Model" on page 41, Dreamweaver provides extension developers with more than 600 JavaScript functions that encapsulate the kinds of tasks that users perform when creating or editing a document in Dreamweaver. Almost any task that the user can accomplish with the menus, floating panels, Property inspectors, Site panel, or Document window can also be done using JavaScript.

Many of the JavaScript API functions require that you specify which document you are working on by getting the appropriate `dom`. The most commonly used function for getting a document object is `dreamweaver.getDocumentDOM()`, because it gets the `dom` of the current user document. Other Dreamweaver functions can also return the document `dom`. For descriptions of `dreamweaver.getDocumentDOM()`, `dreamweaver.newDocumentDOM()` and other functions that return a document `dom`, see "File manipulation functions" on page 447.

Note: In Dreamweaver 4 and later, `dw` can be used interchageably with `dreamweaver` when you write code, so all `dreamweaver` methods can be referred to either as `dreamweaver.functionName()` or `dw.functionName()`.

Understanding the objects in the API

Each time you call a JavaScript API method, it returns information from one of the following three objects:

- An object that represents the current document, another open document, or a document on disk
- The `site` object
- The `dreamweaver` object

Functions that work with the current document, another open document, or a document on disk are methods of the `DOM` object. Methods that work directly with DOM objects are listed as `dom.functionName()`. To work with DOM methods, you must first get the DOM of a document (see "dreamweaver.getDocumentDOM()" on page 453) and call the functions as methods of that DOM. Dreamweaver DOM objects have all the properties and methods of a document object, as described in "The Dreamweaver Document Object Model" on page 41.

Functions that refer to the Site panel or a selection in the Site panel are methods of the `site` object. For example, `site.put(remoteSite)` puts the currently selected files from the Site panel into the remote site.

Functions that refer to Dreamweaver are methods of the `dreamweaver` object. For example, `dreamweaver.closeDocument()` is a function of the `dreamweaver` object and causes Dreamweaver to close the current document. The `dreamweaver` object can be abbreviated as `dw`.

How this chapter is organized

The methods in the Dreamweaver JavaScript API are grouped functionally, then alphabetically by object, and then by method name. Each section describes methods of the `dom` object, the `site` object, or the `dreamweaver` object. For example, methods that deal with creating, applying, and deleting cascading style sheet (CSS) styles are grouped under CSS Styles functions; CSS `dom` methods are listed first, and then they are followed by CSS `dreamweaver` methods. Enablers are listed in the Enablers section. Deprecated functions are listed in a section at the end of the chapter. Optional arguments are enclosed in braces ({ }).

About enablers

The functions in the JavaScript API can perform any task that the user can perform using the Dreamweaver user interface. However, certain functions do not work under specific conditions. Calling a function through JavaScript when those conditions exist generates one or more JavaScript errors. Enablers check the current context to see whether conditions exist that would generate a JavaScript error if the associated function is called. For example, `site.canGet()` checks whether Dreamweaver can perform a Get operation on the `site` object (`site.get()`).

When a function in the Dreamweaver JavaScript API has an enabler, the enabler is listed with the function and documented in "Enablers" on page 409. Many functions do not require enablers, because the menu item that is associated with the function is always enabled, because the function is unrelated to menus, or because an enabler would duplicate the function of an existing API function. For example, functions that require the use of a current document often do not require an enabler because you can use `dw.getDocumentDOM()!=null` to test the current context.

Assets panel functions

Assets panel functions, which are programmed into the API as an asset panel, let you manage and use the elements in the Assets panel (templates, libraries, images, Macromedia Shockwave and Flash movies, URLs, colors, movies, and scripts).

dreamweaver.assetPalette.addToFavoritesFromDocument()

Availability

Dreamweaver 4

Description

Adds the element that is selected in the Document window to the Favorites list. This function handles only images, movies, Shockwave files, Flash files, text font colors, and URLs.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.addToFavoritesFromSiteAssets()

Availability

Dreamweaver 4

Description

Adds elements that are selected in the Site list to the Favorites list and gives each item a nickname in the Favorites list. This function does not remove the element from the Site list.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.addToFavoritesFromSiteWindow()

Availability

Dreamweaver 4

Description

Adds the elements that are selected in the Site Panel or Site Map to the Favorites list. This function handles only images, movies, scripts, Shockwave files, Flash files, and URLs (in the case of the Site Map). If other folders or files are selected, they are ignored.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.copyToSite()

Availability

Dreamweaver 4

Description

Copies selected elements to another Site and puts them in that Site's Favorites list. If the elements are files (other than colors or URLs), the actual file is copied into that Site.

Arguments

targetSite

targetSite is the name of the target Site, as returned from the `site.getSites()` call.

Returns

Nothing.

dreamweaver.assetPalette.edit()

Availability

Dreamweaver 4

Description

Edits selected elements with primary external editor or Custom Edit control. For colors, the color picker appears. For URLs, a dialog box appears and prompts the user for a URL and a nickname. Not available for the Site list of colors and URLs.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.assetPalette.canEdit()" on page 418

dreamweaver.assetPalette.getSelectedCategory()

Availability

Dreamweaver 4

Description

Returns the currently selected category, which can be one of the following categories: `"templates"`, `"library"`, `"images"`, `"movies"`, `"shockwave"`, `"flash"`, `"scripts"`, `"colors"`, or `"urls"`.

Arguments

None.

Returns

The currently selected category.

dreamweaver.assetPalette.getSelectedItems()

Availability

Dreamweaver 4

Description

Returns an array of the selected items in the Assets panel, either in the Site list or Favorites list.

Arguments

None.

Returns

An array of the following three strings for each selected item:

- *`name`* is the name/filename or nickname, as seen in the panel.
- *`value`* is the full file path, full URL, or color value, depending on the selected item.
- *`type`* is either `"folder"` or one of the following categories: `"templates"`, `"library"`, `"images"`, `"movies"`, `"shockwave"`, `"flash"`, `"scripts"`, `"colors"`, or `"urls"`.

Note: If nothing is selected in the Assets panel, this function returns an array of one empty string.

Example

If URLs is the category, and a folder MyFolderName and a URL MyFavoriteURL are both selected in the Favorites list, the function returns the following list:

```
items[0] = "MyFolderName"
items[1] = "//path/FolderName"
items[2] = "folder"
items[3] = "MyFavoriteURL"
items[4] = "http://www.MyFavoriteURL.com"
items[5] = "urls"
```

dreamweaver.assetPalette.getSelectedView()

Availability

Dreamweaver 4

Description

Indicates which list is currently shown in the Assets panel.

Arguments

None.

Returns

Returns either `"site"` or `"favorites"`.

dreamweaver.assetPalette.insertOrApply()

Availability

Dreamweaver 4

Description

Inserts selected elements or applies the element to the current selection. Applies templates, colors to selection, and URLs to selection; it also inserts URLs and other elements at the insertion point. If a document isn't open, the function is not available.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.assetPalette.canInsertOrApply()" on page 418

dreamweaver.assetPalette.locateInSite()

Availability

Dreamweaver 4

Description

Selects files that are associated with the selected elements in the local side of the Site panel. This function does not work for colors or URLs. It is available in the Site list and the Favorites list. If a folder is selected in the Favorites list, it is ignored.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.newAsset()

Availability

Dreamweaver 4

Description

Creates a new element for the current category in the Favorites list. For library and templates, this is a new blank library or template file that the user can name immediately. For colors, the color picker appears. For URLs, a dialog box appears and prompts the user for a URL and a nickname. It is not available for images, movies, Shockwave files, Flash files, or scripts.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.newFolder()

Availability

Dreamweaver 4

Description

Creates a new folder in the current category with the default name (untitled) and puts an text box around the default name. It is available only in the Favorites list.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.recreateLibraryFromDocument()

Availability

Dreamweaver 4

Description

Replaces the deprecated `libraryPalette` function, `recreateLibraryFromDocument()`. Creates an LBI file for the selected instance of a library item in the current document. This function is equivalent to clicking Recreate in the Property inspector.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.refreshSiteAssets()

Availability

Dreamweaver 4

Description

Scans Site, switches to the Site list, and populates the list

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.removeFromFavorites()

Availability

Dreamweaver 4

Description

Removes the selected elements from the Favorites list. This function does not delete the actual file on disk, except in the case of a library or template where the user is prompted before the file is deleted. It works only in the Favorites list or if the category is Library or Templates.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.renameNickname()

Availability

Dreamweaver 4

Description

Edits the folder name or the file's nickname by displaying an text box around the existing nickname. It is available only in the Favorites list or in the Library or Template category.

Arguments

None.

Returns

Nothing.

dreamweaver.assetPalette.setSelectedCategory()

Availability

Dreamweaver 4

Description

Switches to show a different category.

Arguments

categoryType

categoryType can be one of the following categories: `"templates"`, `"library"`, `"images"`, `"movies"`, `"shockwave"`, `"flash"`, `"scripts"`, `"colors"`, or `"urls"`.

Returns

Nothing.

dreamweaver.assetPalette.setSelectedView()

Availability

Dreamweaver 4

Description

Switches the display to show either the Site list or Favorites list.

Arguments

viewType

viewType can be `site` or `favorites`.

Returns

Nothing.

dreamweaver.referencePalette.getFontSize()

Availability

Dreamweaver 4

Description

Returns the current font size of the Reference panel display region.

Arguments

None.

Returns

The relative font size as `small`, `medium`, or `large`.

dreamweaver.referencePalette.setFontSize()

Availability

Dreamweaver 4

Description

Changes the font size that appears in the Reference panel.

Arguments

`fontSize`

`fontSize` is one of the following relative sizes: `small`, `medium`, or `large`.

Returns

Nothing.

Behavior functions

Behavior functions let you add behaviors to and remove them from an object, find out which behaviors are attached to an object, get information about the object to which a behavior is attached, and so on. Methods of the `dreamweaver.behaviorInspector` object either control or act on the selection in the Behaviors panel, not in the current document.

dom.addBehavior()

Availability

Dreamweaver 3

Description

Adds a new event/action pair to the selected element. This function is valid only for the active document.

Arguments

`event`, *`action`*, *`{eventBasedIndex}`*

- *`event`* is the JavaScript event handler that should attach the behavior to the element; for example, `onClick`, `onMouseOver`, or `onLoad`.
- *`action`* is the function call that `applyBehavior()` returns if the action is added using the Behaviors panel; for example, `"MM_popupMsg('Hello World')"`.
- *`eventBasedIndex`* is the position at which this action should be added. *`eventBasedIndex`* is a zero-based index; if two actions already are associated with the specified event, and you specify *`eventBasedIndex`* as 1, this action executes between the other two. If you omit this argument, the action is added after all existing actions for the specified event.

Returns

Nothing.

dom.getBehavior()

Availability

Dreamweaver 3

Description

Gets the action at the specified position within the specified event. This function acts on the current selection and is valid only for the active document.

Arguments

`event`, *`{eventBasedIndex}`*

- *`event`* is the JavaScript event handler through which the action is attached to the element; for example, `onClick`, `onMouseOver`, or `onLoad`.
- *`eventBasedIndex`* is the position of the action to get. For example, if two actions are associated with the specified event, 0 is first and 1 is second. If you omit this argument, all the actions for the specified event return.

Returns

A string that represents the function call (for example, `"MM_swapImage('document.Image1','document.Image1','foo.gif','#933292969950')"`) or an array of strings if *eventBasedIndex* is omitted.

dom.reapplyBehaviors()

Availability

Dreamweaver 3

Description

Checks to make sure that the functions that are associated with any behavior calls on the specified node are in the `HEAD` of the document and inserts them if they are missing.

Arguments

{elementNode}

elementNode is an element node within the current document. If you omit the argument, Dreamweaver checks all element nodes in the document for orphaned behavior calls.

Returns

Nothing.

dom.removeBehavior()

Availability

Dreamweaver 3

Description

Removes the action at the specified position within the specified event. This function acts on the current selection and is valid only for the active document.

Arguments

event, *{eventBasedIndex}*

- *event* is the event handler through which the action is attached to the element; for example, `onClick`, `onMouseOver`, or `onLoad`. If you omit this argument, all actions are removed from the element.
- *eventBasedIndex* is the position of the action to be removed. For example, if two actions are associated with the specified event, 0 is first and 1 is second. If you omit this argument, all the actions for the specified event are removed.

Returns

Nothing.

dreamweaver.getBehaviorElement()

Availability

Dreamweaver 2

Description

Gets the DOM object that corresponds to the tag to which the behavior is being applied. This function is applicable only in Behavior action files.

Arguments

None.

Returns

A DOM object or `null`. This function returns `null` under the following circumstances:

- When the current script is not executing within the context of the Behaviors panel
- When the Behaviors panel is being used to edit a behavior in a timeline
- When the currently executing script is invoked by `dreamweaver.popupAction()`
- When the Behaviors panel is attaching an event to a link wrapper and the link wrapper does not yet exist
- When this function appears outside of an action file

Example

`dreamweaver.getBehaviorElement()` can be used in the same way as "`dreamweaver.getBehaviorTag()`" `on page 382` to determine whether the selected action is appropriate for the selected HTML tag, except that it gives you access to more information about the tag and its attributes. For example, if you write an action that can be applied only to a hypertext link (`A HREF`) that does not target another frame or window, you can use `getBehaviorElement()` as part of the function that initializes the user interface for the Parameters dialog box.

```
function initializeUI(){
  var theTag = dreamweaver.getBehaviorElement();
  var CANBEAPPLIED = (theTag.tagName == "A" && ¬
  theTag.getAttribute("HREF") != null && ¬
  theTag.getAttribute("TARGET") == null);
  if (CANBEAPPLIED) {
    // display the action UI
  } else{
    // display a helpful message that tells the user
    // that this action can only be applied to a
    // hyperlink without an explicit target]
  }
}
```

dreamweaver.getBehaviorTag()

Availability

Dreamweaver 1.2

Description

Gets the source of the tag to which the behavior is being applied. This function is applicable only in action files.

Arguments

None.

Returns

A string that represents the source of the tag. This is the same string that passes as an argument (*`HTMLelement`*) to the `canAcceptBehavior()` function. If this function appears outside an action file, the return value is an empty string.

Example

If you write an action that can be applied only to a hypertext link (`A HREF`), you can use `getBehaviorTag()` as part of the function that initializes the user interface for the Parameters dialog box.

```
function initializeUI(){
  var theTag = dreamweaver.getBehaviorTag().toUpperCase();
  var CANBEAPPLIED = (theTag.indexOf('HREF') != -1));
  if (CANBEAPPLIED) {
    // display the action UI
  } else{
    // display a helpful message that tells the user
    // that this action can only be applied to a
    // hyperlink
  }
}
```

dreamweaver.popupAction()

Availability

Dreamweaver 2

Description

Presents the user with a Parameters dialog box for the specified behavior action. To the user, the effect is the same as selecting the action from the Actions pop-up menu in the Behaviors panel. This function lets extension files other than actions attach behaviors to objects in the user's document. It blocks other edits until the user dismisses the dialog box.

Note: This function can be called only within `objectTag()` or in any script in a command or the Property inspector file.

Arguments

`actionName`, *`{funcCall}`*

- *`actionName`* is the name of a file in the Configuration/Behaviors/Actions folder that contains a JavaScript behavior action; for example, `"Timeline/Play Timeline.htm"`.
- *`funcCall`* is a string that contains a function call for the action specified in `actionName`; for example, `"MM_playTimeline(...)"`. This argument, if specified, is supplied by the `applyBehavior()` function in the action file.

Returns

The function call for the behavior action. When the user clicks OK in the Parameters dialog box, the behavior is added to the current document (the appropriate functions are added to the `HEAD` of the document, HTML might be added to the top of the `BODY`, and other edits might be made to the document). The function call (for example, `"MM_playTimeline(...)"`) is not added to document but becomes the return value of this function.

dreamweaver.behaviorInspector.getBehaviorAt()

Availability

Dreamweaver 3

Description

Gets the event/action pair at the specified position in the Behaviors panel.

Arguments

`positionIndex`

Returns

An array of two items:

- An event handler
- A function call or JavaScript statement

Example

Because `positionIndex` is a zero-based index, if the Behaviors panel displays the list, a call to `dreamweaver.behaviorInspector.getBehaviorAt(2)` returns an array that contains two strings: `"onMouseOver"` and `"MM_changeProp('document.moon','document.moon','src','sun.gif', 'MG')"`.

dreamweaver.behaviorInspector.getBehaviorCount()

Availability

Dreamweaver 3

Description

Counts the number of actions that are attached to the currently selected element through event handlers.

Arguments

None.

Returns

An integer that represents the number of actions that are attached to the element. This number is equivalent to the number of actions that are visible in the Behaviors panel and includes Dreamweaver behavior actions and custom JavaScript.

Example

A call to `dreamweaver.behaviorInspector.getBehaviorCount()` for the selected link `<A HREF="javascript:setCookie()" onClick="MM_popupMsg('A cookie has been set.');parent.rightframe.location.href='aftercookie.html'">` returns 2.

dreamweaver.behaviorInspector.getSelectedBehavior()

Availability

Dreamweaver 3

Description

Gets the position of the selected action in the Behaviors panel.

Arguments

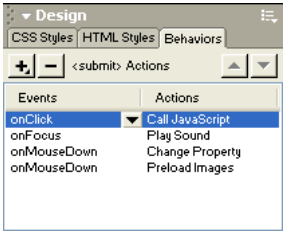
None.

Returns

An integer that represents the position of the selected action in the Behaviors panel, or `-1` if no action is selected.

Example

If the first action in the Behaviors panel is selected, as shown in the following example, a call to `dreamweaver.behaviorInspector.getSelectedBehavior()` returns 0.



dreamweaver.behaviorInspector.moveBehaviorDown()

Availability

Dreamweaver 3

Description

Moves a behavior action lower in sequence by changing its execution order within the scope of an event.

Arguments

`positionIndex`

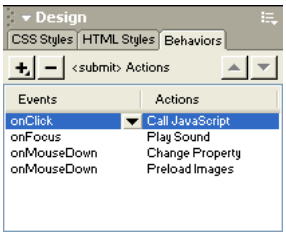
`positionIndex` is the position of the action in the Behaviors panel. The first action in the list is at position 0.

Returns

Nothing.

Example

Assuming the Behaviors panel setup shown in the following example, calling `dreamweaver.behaviorInspector.moveBehaviorDown(2)` swaps the positions of the Preload Images and the Change Property actions on the `onMouseDown` event. Calling `dreamweaver.behaviorInspector.moveBehaviorDown()` for any other position has no effect because the `onClick` and `onFocus` events each have only one associated behavior, and the behavior at position 3 is already at the bottom of the `onMouseDown` event group.



dreamweaver.behaviorInspector.moveBehaviorUp()

Availability

Dreamweaver 3

Description

Moves a behavior higher in sequence by changing its execution order within the scope of an event.

Arguments

positionIndex

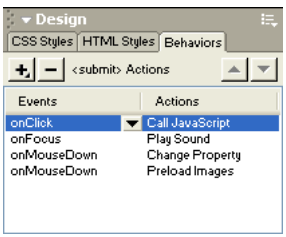
positionIndex is the position of the action in the Behaviors panel. The first action in the list is at position 0.

Returns

Nothing.

Example

Assuming the Behaviors panel setup that is shown in the following example, calling `dreamweaver.behaviorInspector.moveBehaviorUp(3)` swaps the positions of the Preload Images and the Change Property actions on the `onMouseOver` event. Calling `dreamweaver.behaviorInspector.moveBehaviorUp()` for any other position has no effect because the `onClick` and `onFocus` events each have only one associated behavior, and the behavior at position 2 is already at the top of the `onMouseDown` event group.



dreamweaver.behaviorInspector.setSelectedBehavior()

Availability

Dreamweaver 3

Description

Selects the action at the specified position in the Behaviors panel.

Arguments

positionIndex

positionIndex is the position of the action in the Behaviors panel. The first action in the list is at position 0. To deselect all actions, specify a *positionIndex* of –1. Specifying a position for which no action exists is equivalent to specifying –1.

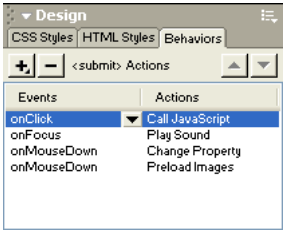
Returns

Nothing.

None.

Example

Assuming the Behaviors panel setup shown in the following example, calling `dreamweaver.behaviorInspector.setSelection(2)` selects the Change Property action that is associated with the `onMouseDown` event.



Clipboard functions

Clipboard functions are related to cutting, copying, and pasting. On the Macintosh, some Clipboard functions can also apply to text boxes in dialog boxes and floating panels. Functions that can operate in text boxes are implemented as methods of the `dreamweaver` object and as methods of the `dom` object. The `dreamweaver` version of the function operates on the selection in the active window: the current Document window, the Code inspector, or the Site panel. On the Macintosh, the function can also operate on the selection in a text box if it is the current field. The `dom` version of the function always operates on the selection in the specified document.

dom.clipCopy()

Availability

Dreamweaver 3

Description

Copies the selection, including any HTML markup that defines the selection, to the Clipboard.

Arguments

None.

Returns

Nothing.

dom.clipCopyText()

Availability

Dreamweaver 3

Description

Copies the selected text to the Clipboard, ignoring any HTML markup.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canClipCopyText()" on page 410

dom.clipCut()

Availability

Dreamweaver 3

Description

Removes the selection, including any HTML markup that defines the selection, to the Clipboard.

Arguments

None.

Returns

Nothing.

dom.clipPaste()

Availability

Dreamweaver 3

Description

Pastes the contents of the Clipboard into the current document at the current insertion point or in place of the current selection. If the Clipboard contains HTML, it is interpreted as such.

Arguments

None.

Returns

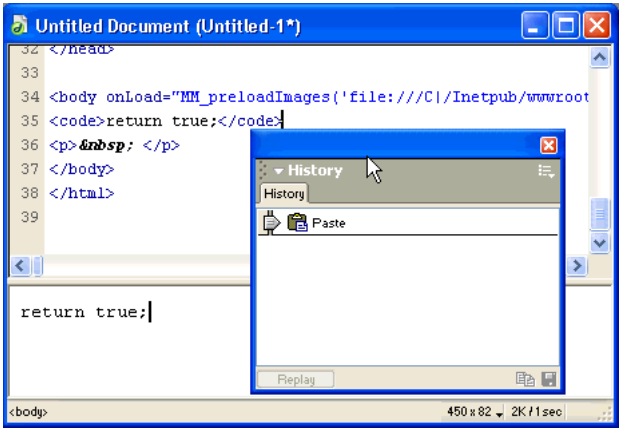
Nothing.

Enabler

"dom.canClipPaste()" on page 410

Example

If the Clipboard contains `<code>return true;</code>`, a call to `dw.getDocumentDOM().clipPaste()` results in the following illustration:



dom.clipPasteText()

Availability

Dreamweaver 3

Description

Pastes the contents of the Clipboard into the current document at the insertion point or in place of the current selection. It replaces any linefeeds in the Clipboard content with `BR` tags. If the Clipboard contains HTML, it is not interpreted; angle brackets are pasted as `<` and `>`.

Arguments

None.

Returns

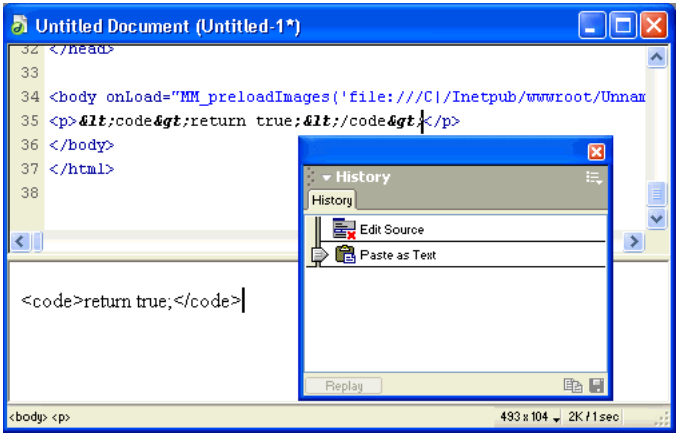
Nothing.

Enabler

"dom.canClipPasteText()" on page 410

Example

If the Clipboard contains `<code>return true;</code>`, a call to `dw.getDocumentDOM().clipPasteText()` results in the following illustration:



dreamweaver.clipCopy()

Availability

Dreamweaver 3

Description

Copies the current selection from the active Document window, dialog box, floating panel, or Site panel to the Clipboard.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canClipCopy()" on page 418

dreamweaver.clipCut()

Availability

Dreamweaver 3

Description

Removes the selection from the active Document window, dialog box, floating panel, or Site panel to the Clipboard.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canClipCut()" on page 419

dreamweaver.clipPaste()

Availability

Dreamweaver 3

Description

Pastes the contents of the Clipboard into the current document, dialog box, floating panel, or Site panel.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canClipPaste()" on page 419

dreamweaver.getClipboardText()

Availability

Dreamweaver 3

Description

Gets all the text that is stored on the Clipboard.

Arguments

{bAsText}

{bAsText} is a Boolean value that specifies whether the Clipboard content is retrieved as text. If *bAsText* is `true`, the Clipboard content is retrieved as text. If *bAsText* is `false`, the behavior is the same as in Dreamweaver 3. This argument defaults to `false`.

Returns

A string that contains the contents of the Clipboard, if the Clipboard contains text (which can be HTML); otherwise, nothing.

Example

If `dreamweaver.getClipboardText()` returns `"text <b>bold</b> text"`, then `dreamweaver.getClipboardText(true)` returns `"text bold text"`.

Code hints functions

Code Hints are menus that Macromedia Dreamweaver MX pops up when you type certain character patterns in the Code view. Code Hints offer a typing shortcut by providing a list of strings that potentially complete the string you are typing. If the string you are typing appears in the menu, you can scroll to it and press Enter or Return to complete your entry. For example, when you type <, a pop-up menu shows a list of tag names. Instead of typing the rest of the tag name, you can select the tag from the menu to include it in your text.

Dreamweaver loads Code Hints menus from the CodeHints.xml file in the Configuration folder. You can add Code Hints menus to Dreamweaver MX by defining them in the CodeHints.xml file. After Dreamweaver MX loads the contents of CodeHints.xml, you can also add new Code Hints menus dynamically through JavaScript. For example, JavaScript code populates the list of session variables in the Bindings panel. You can use the same code to add a Code Hints menu, so when a user types "`Session.`" in Code view, Dreamweaver MX displays a menu of session variables. For information on using JavaScript to add or modify a Code Hints menu, see "Code hints functions" on page 397.

Dreamweaver cannot express some types of Code Hints menus through the XML file or the JavaScript API. Both the CodeHints.xml file and the JavaScript API expose a useful subset of the Code Hints engine, but some Dreamweaver functionality is not accessible. For example, there is no JavaScript hook to pop up a color picker, so Dreamweaver cannot express the Attribute Values menu using JavaScript. You can only pop up a menu of text items from which you can insert text. Also, when you insert text, the insertion pointer is placed after the inserted string.

The CodeHints.xml file

The CodeHints.xml file contains the following entities:

- A list of all the menu groups

  Dreamweaver displays the list of menu groups when you select the Code Hints category from the Preferences dialog box. You can activate the Preferences dialog box by selecting Preferences from the Edit menu. Dreamweaver MX provides the following menu groups or types of Code Hints menus: Tag Names, Attribute Names, Attribute Values, Function Arguments, Object Methods and Variables, and HTML Entities.

- The description for each menu group

  The description appears in the Preferences dialog box for the Code Hints category when you select the menu group in the list. The description for the selected entry appears below the menu group list.

- Code Hints menus

  A menu consists of a pattern that triggers the Code Hints menu, and a list of menu items. For example, a pattern such as "`&`" could trigger a menu such as "`&`", "`>`", "`<`".

The following example shows the format of the CodeHints.xml file.

```
<codehints>
<menugroup name="HTML Entities" enabled="true" id="CodeHints_HTML_Entities">
  <description>
  <![CDATA[ When you type a '&', a drop-down menu shows
    a list of HTML entities. The list of HTML entities
    is stored in Configuration/CodeHints.xml. ]]>
  </description>

  <menu pattern="&amp;">
      <menuitem value="&amp;amp;" texticon="&amp;"/>
      <menuitem value="&amp;lt;" icon="lessThan.gif"/>
  </menu>
</menugroup>

<menugroup name="Tag Names" enabled="true" id="CodeHints_Tag_Names">
  <description>
      <![CDATA[ When you type '<', a drop-down menu shows
      all possible tag names.  You can edit the list of tag
      names using the
      <a href="javascript:dw.popupTagLibraryEditor()"> Tag Library Editor
      </a>]]>
    </description>
</menugroup>

<menugroup name="Function Arguments" enabled="true"
    id="CodeHints_Function_Arguments">
  <description>
    ...
  </description>
  <function pattern="ArraySort(array, sort_type, sort_order)"
        doctypes="CFML"/>
  <function pattern="Response.addCookie(Cookie cookie)"
        doctypes="JSP"/>
</menugroup>
<codehints>
```

Code Hints tags

The CodeHints.xml file contains the following tags, which define Code Hints menus. You can use these tags to define additional Code Hints menus.

<codehints>

Description

The `codehints` tag is the root of the `CodeHints.xml` file.

Attributes

None.

Contents

One or more `menugroup` tags.

Container

None.

Example

```
<codehints>
```

<menugroup>

Description

Each `menugroup` tag corresponds to a type of menu. You can see the menu types that Dreamweaver MX defines by selecting the Code Hints category from the Preferences dialog box. Select Preferences from the Edit menu to display the Preferences dialog box.

You can create a new menu group or add to an existing group. Menu groups are logical collections of menus that the user might want to enable or disable, using the Preferences dialog box.

Attributes

`name, enabled, id`

`name` is the localized name that appears in the list of menu groups in the Code Hints category of the Preferences dialog box.

`enabled` indicates whether the menu group is currently checked or enabled. A menu group that is enabled appears with a check mark next to it in the Code Hints category of the Preferences dialog box. Assign a value of `true` to enable the menu group. Assign the value `false` to disable a menu group.

`id` is a nonlocalized identifier that refers to the menu group.

Contents

`description`, `menu`, and `function` tags.

Container

`codehints` tag.

Example

```
<menugroup name="Session Variables" enabled="true" id="Session_Code_Hints">
```

<description>

Description

The `description` tag contains text that Dreamweaver displays when you select the menu group from the Preferences dialog box. The description text displays below the list of menu groups. The description text might optionally contain a single `<a>` tag where the `href` attribute must be a JavaScript URL that Dreamweaver executes if the user clicks the link. Use the XML CDATA construct to enclose any special or illegal characters in the string so that Dreamweaver will treat them as text.

Attributes

None.

Contents

Description text.

Container

`menugroup` tag.

Example

```
<description>
<![CDATA[ To add or remove tags and attributes, use the <a
  href="javascript:dw.tagLibrary.showTagLibraryEditor()">Tag Library Editor</
  a>.
  ]]>
</description>
```

<menu>

Description

Describes a single pop-up menu. Dreamweaver pops up the menu whenever the user types the last character of the string in the pattern attribute. For example, the menu that shows the contents of a Session variable might have a pattern attribute that is equal to "`Session.`".

Attributes

`pattern doctypes casesensitive`

`pattern` specifies the pattern of typed characters that cause Dreamweaver to pop up the Code Hints menu. If the first character of the pattern is a letter, number, or underscore, Dreamweaver displays the menu only if the character that precedes the pattern in the document is not a letter, number, or underscore. For example, if the pattern is "`Session.`", Dreamweaver does not pop up the menu if the user types "`my_Session.`".

`doctypes` specifies that the menu is active only for the specified document types. This attribute lets you specify different lists of function names for ASP-JavaScript (ASP-JS), Java Server Pages (JSP), ColdFusion, and so on. You can specify `doctypes` as a comma-separated list of document type IDs. See the Dreamweaver Configuration/Documenttypes/MMDocumentTypes.xml file for a list of Dreamweaver document types.

`casesensitive` specifies whether the pattern is case-sensitive. The possible values for `casesensitive` are `true`, `false`, or a subset of the comma-separated list that you specify for the `doctypes` attribute. The list of document types lets you specify that the pattern is case-sensitive for some document types but not for others. The value defaults to `false` if you omit this attribute. If `casesensitive` is `true`, the Code Hints menu will pop up only if the text that the user types exactly matches the pattern specified by the pattern attribute. If `casesensitive` is `false`, the menu pops up even if the pattern is lowercase and the text is uppercase.

Contents

`menuitem` tag.

Container

`menugroup` tag.

Example

```
<menu pattern="CGI." doctypes="ColdFusion">
```

<menuitem>

Description

Specifies the text for an item in a Code Hints pop-up menu. The `menuitem` also specifies the value to insert into the text when you select the item.

Attributes

`label value {icon} {texticon}`

`label` is the string that Dreamweaver displays in the pop-up menu.

`value` is the string that Dreamweaver inserts in the document when you select the menu item. When the user selects the item from the menu and presses Enter or Return, Dreamweaver replaces all the text that the user typed since the menu popped up. The user typed the pattern-matching characters before the menu popped up, so Dreamweaver does not insert them again. For example, if you want to insert `&amp`, which is the HTML entity for `&`, you could define the following `menu` and `menuitem` tags:

```
<menu pattern="&amp;">
<menuitem label="&amp;amp;" value="amp;" texticon="&amp;"/>
```

The value attribute does not include the ampersand (&) character because the user typed it before the menu popped up.

`icon` is an optional attribute that specifies the path to an image file that Dreamweaver displays as an icon to the left of the menu text. The location is expressed as a URL, relative to the Configuration folder.

`texticon` is an optional attribute that specifies a text string to appear in the icon area instead of an image file. This attribute is used for the HTML Entities menu.

Contents

None.

Container

`menu` tag.

Example

```
<menuitem label="CONTENT_TYPE" value="&quot;CONTENT_TYPE&quot;)" icon="shared/
  mm/images/hintMisc.gif" />
```

<function>

Description

Replaces the menu tag for specifying function arguments and object methods for a Code Hints pop-up menu. When you type a function or method name in Code view, Dreamweaver pops up a menu of function arguments. Each time you type a comma, Dreamweaver updates the menu to display only the remaining arguments.

For object methods, when you type the object name Dreamweaver pops up a menu of the methods that are defined for that object.

The set of recognized functions is stored in the Dreamweaver Configuration/CodeHints.xml file.

Attributes

`pattern doctypes`

`pattern` specifies the name of the function and its argument list. For methods, the pattern attribute describes the name of the object, the name of the method, and the method's arguments. For a function name, the Code Hints menu pops up when the user types `functionname(`. The menu shows the list of arguments for the function. For an object method, the Code Hints menu pops up when the user types `objectname.` (including the period). This menu shows the methods that have been specified for the object. After that, the Code Hints menu pops up a list of the arguments for the method in the same way it does for a function.

`doctypes` specifies that the menu is active only for the specified document types. This attribute lets you specify different lists of function names for ASP-JavaScript (ASP-JS), Java Server Pages (JSP), ColdFusion, and so on. You can specify `doctypes` as a comma-separated list of document type IDs. See the Dreamweaver Configuration/Documenttypes/MMDocumentTypes.xml file for a list of Dreamweaver document types.

`casesensitive` specifies whether the pattern is case-sensitive. The possible values for `casesensitive` are `true`, `false` or a subset of the comma-separated list that you specify for the `doctypes` attribute. The list of document types lets you specify that the pattern is case-sensitive for some document types but not for others. The value defaults to `false` if you omit this attribute. If `casesensitive` is `true`, the Code Hints menu pops up only if the text that the user types exactly matches the pattern specified by the pattern attribute. If `casesensitive` is `false`, the menu pops up even if the pattern is lowercase and the text is uppercase.

Contents

None.

Container

`menugroup` tag.

Example

```
// function example
<function pattern="CreateDate(year, month, day)" DOCTYPES="ColdFusion" />
// object method example
<function pattern="application.getAttribute(String name)" DOCTYPES="JSP" />
```

Code hints functions

The JavaScript Code hints API consists of four functions.

dw.codeHints.addMenu()

Availability

Dreamweaver MX

Description

This function dynamically defines a new `menu` tag in the CodeHints.xml file. If there is an existing menu tag that has the same pattern and document type, this function adds items to the existing menu.

Arguments

```
menuGroupId, pattern, labelArray, {valueArray}, {iconArray}, {doctypes},
  {casesensitive}
```

`menuGroupId` is the ID attribute for one of the `<menugroup>` tags.

`pattern` is the pattern attribute for the new `<menu>` tag.

`labelArray` is an array of strings. Each string is the text for a single menu item in the pop-up menu.

`valueArray` is an array of strings, which should be the same length as `labelArray`. When a user chooses an item from the pop-up menu, the string in this array is inserted in the user's document. If the string to be inserted is always the same as the menu label, this argument may be null.

`iconArray` is either a string or an array of strings. If it is a string, it specifies the URL for a single image file that Dreamweaver uses for all items in the menu. If it is an array of strings, it must be the same length as `labelArray`. Each string is a URL, relative to the Dreamweaver Configuration folder, for an image file that Dreamweaver uses as an icon for the corresponding menu item. If this argument is null, Dreamweaver displays the menu without icons.

`doctypes` is an optional argument that specifies that this menu is active for only certain document types. You can specify `doctypes` as a comma-separated list of document type IDs. See the Dreamweaver Configuration/Documenttypes/MMDocumentTypes.xml file for a list of Dreamweaver document types.

`casesensitive` specifies whether the pattern is case-sensitive. The possible values for `casesensitive` are the Boolean values `true` or `false`. The value defaults to `false` if you omit this argument. If `casesensitive` is `true`, the Code Hints menu pops up only if the text that the user types exactly matches the pattern specified by the pattern attribute. If `casesensitive` is `false`, the menu pops up even if the pattern is lowercase and the text is uppercase.

Returns

Nothing.

Example

If the user creates a record set called "`myRs`", the following code would create a menu for `myRS`:

```
dw.codeHints.addMenu(
  "CodeHints_object_methods",  // menu is enabled if object methods are enabled
  "myRS.",                     // pop up menu if user types "myRS."
  new Array("firstName", "lastName"),  // items in drop-down menu for myRS
  new Array("firstName", "lastName"),  // text to actually insert in document
  null,              // no icons for this menu
  "ASP_VB, ASP_JS"); // specific to the ASP doc types
```

dw.codeHints.addFunction()

Availability

Dreamweaver MX

Description

Dynamically defines a new `function` tag. If there is an existing `function` tag with the same pattern and document type, this function replaces the existing `function` tag.

Arguments

`menuGroupId, pattern, {doctypes}, {casesensitive}`

`menuGroupId` is the ID string attribute of a `menugroup` tag.

`pattern` is a string that specifies the pattern attribute for the new `function` tag.

`doctypes` is an optional argument that specifies that this function is active for only certain document types. You can specify `doctypes` as a comma-separated list of document type IDs. See the Dreamweaver Configuration/Documenttypes/MMDocumentTypes.xml file for a list of Dreamweaver document types.

`casesensitive` specifies whether the pattern is case-sensitive. The possible values for `casesensitive` are the Boolean values `true` or `false`. The value defaults to `false` if you omit this argument. If `casesensitive` is `true`, the Code Hints menu pops up only if the text that the user types exactly matches the pattern specified by the pattern attribute. If `casesensitive` is `false`, the menu pops up even if the pattern is lowercase and the text is uppercase.

Returns

Nothing.

Example

```
dw.codeHints.addFunction(
  "CodeHints_Object_Methods",
  "out.newLine()",
  "JSP")
```

dw.codeHints.resetMenu()

Availability

Dreamweaver MX

Description

This function resets the specified menu tag or function tag to its state immediately after CodeHints.xml is read. In other words, a call to this function erases the effect of previous calls to `addMenu()` and `addFunction()`.

Arguments

`menuGroupId, pattern, {doctypes}`

`menuGroupId` is the ID string attribute of a `menugroup` tag.

`pattern` is a string that specifies the pattern attribute for the new `menu` or `function` tag to be reset.

`doctypes` is an optional argument that specifies that this menu is active for only certain document types. You can specify `doctypes` as a comma-separated list of document type IDs. See the Dreamweaver Configuration/Documenttypes/MMDocumentTypes.xml file for a list of Dreamweaver document types.

Returns

Nothing.

Example

Your JavaScript code might build a Code Hints menu that contains user-defined session variables. Each time the list of session variables is changed, that code needs to update the menu. Before the code can load the new list of session variables into the menu, it needs to remove the old list. Calling this function removes the old session variables.

dw.codeHints.showCodeHints()

Availability

Dreamweaver MX

Description

Dreamweaver calls this function when the user invokes the Edit > Show Code Hints menu item. The function pops up the Code Hints menu at the current selection location in Code view.

Arguments

None.

Returns

Nothing.

Example

```
dw.codeHints.showCodeHints()
```

Command functions

Command functions help you make the most of the files in the Configuration/Commands folder. They manage the Command menu and call commands from other types of extension files.

dreamweaver.editCommandList()

Availability

Dreamweaver 3

Description

Opens the Edit Command List dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.runCommand()

Availability

Dreamweaver 3

Description

Executes the specified command; it works the same as choosing the command from a menu. If a dialog box is associated with the command, it appears and the command script blocks other edits until the user dismisses the dialog box. This function provides the ability to call a command from another extension file.

Note: This function can be called only within the `objectTag()` function in objects; it can be called from or in any script in a command, Property inspector file, or menu command.

Arguments

commandFile, *{commandArg1}*, *{commandArg2}*,...*{commandArgN}*

- *commandFile* is a filename in the Configuration/Commands folder.
- The second and remaining arguments pass to the `receiveArguments` function in *commandFile*.

Returns

Nothing.

Example

You can write a custom Property inspector for tables that let users get to the Format Table command from a button on the inspector by calling the following function from the button's `onClick` event handler:

```
function callFormatTable(){
  dreamweaver.runCommand('Format Table.htm');
}
```

Components functions

Server Components functions let you access the currently selected node of the Server Components tree control that appears in the Components panel. Using these functions, you can also refresh the view of the component tree.

dreamweaver.serverComponents.getSelectedNode()

Availability

Dreamweaver MX

Description

Returns the currently selected `ComponentRec` property in the Server Components tree control.

Arguments

None.

Returns

`ComponentRec` property.

dreamweaver.serverComponents.refresh()

Availability

Dreamweaver MX

Description

Refreshes the view of the component tree.

Arguments

None.

Returns

Nothing.

Conversion functions

Conversion functions convert tables to layers, layers to tables, and Cascading Style Sheets (CSS) styles to HTML markup. Each function exactly duplicates the behavior of one of the conversion commands in the File or Modify menu.

dom.convertLayersToTable()

Availability

Dreamweaver 3

Description

Opens the Convert Layers to Table dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canConvertLayersToTable()" on page 411

dom.convertTablesToLayers()

Availability

Dreamweaver 3

Description

Opens the Convert Tables to Layers dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canConvertTablesToLayers()" on page 411

dom.convertTo30()

Availability

Dreamweaver 3

Description

Opens the "Convert to 3.0 Browser Compatible" dialog box.

Arguments

None.

Returns

Nothing.

CSS Styles functions

CSS styles functions handle the application, removal, creation, and deletion of CSS styles. Methods of the `dreamweaver.cssStylePalette` object either control or act on the selection in the CSS Styles panel, not in the current document.

dom.applyCSSStyle()

Availability

Dreamweaver 4

Description

Applies the specified style to the specified element. This function is valid only for the active document.

Arguments

`elementNode`, *`styleName`*, *`[classOrID]`*, *`[bForceNesting]`*

- *`elementNode`* is an element node in the DOM. If *`elementNode`* is `NULL` or an empty string (`''`), the function acts on the current selection.
- *`styleName`* is the name of a CSS style.
- *`[classOrID]`* is the attribute with which the style should be applied (either `"class"` or `"id"`). If *`elementNode`* is `NULL` or an empty string and no tag exactly surrounds the selection, the style is applied using `SPAN` tags. If the selection is an insertion point, Dreamweaver uses heuristics to determine to which tag the style should be applied.
- *`[bForceNesting]`* is a Boolean value, which indicates whether nesting is allowed. If the *`bForceNesting`* flag is specified, Dreamweaver inserts a new `SPAN` tag instead of trying to modify the existing tags in the document. This argument defaults to `false` if it is not specified.

Returns

Nothing.

Example

The following code applies the `red` style to the selection, either by surrounding the selection with `SPAN` tags or by applying a `CLASS` attribute to the tag that surrounds the selection:

```
var theDOM = dreamweaver.getDocumentDOM('document');
theDOM.applyCSSStyle('','red');
```

dom.removeCSSStyle()

Availability

Dreamweaver 3

Description

Removes the `CLASS` or `ID` attribute from the specified element, or removes the `SPAN` tag that completely surrounds the specified element. This function is valid only for the active document.

Arguments

elementNode, *{classOrID}*

- *elementNode* is an element node in the DOM. If *elementNode* is specified as an empty string (" "), the function acts on the current selection.
- *classOrID* is the attribute that should be removed (either "class" or "id"). If *classOrID* is not specified, it defaults to "class". If no CLASS attribute is defined for *elementNode*, then the SPAN tag surrounding *elementNode* is removed.

Returns

Nothing.

dreamweaver.cssStylePalette.applySelectedStyle()

Availability

Dreamweaver MX

Description

Applies the selected style to the current active document or to its attached style sheet, depending on the selection in the CSS Styles panel.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.cssStylePalette.canApplySelectedStyle()" on page 424

dreamweaver.cssStylePalette.attachStyleSheet()

Availability

Dreamweaver 4

Description

Displays a dialog box that lets users attach a style sheet to the current active document or to one of its attached style sheets, depending on the selection in the CSS Styles panel.

Arguments

None.

Returns

Nothing.

dreamweaver.cssStylePalette.deleteSelectedStyle()

Availability

Dreamweaver 3

Description

Deletes the style that is currently selected in the CSS Styles panel from the document.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.cssStylePalette.canDeleteSelectedStyle()" on page 425

dreamweaver.cssStylePalette.duplicateSelectedStyle()

Availability

Dreamweaver 3

Description

Duplicates the style that is currently selected in the CSS Styles panel and displays the Duplicate Style dialog box to let the user assign a name or selector to the new style.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.cssStylePalette.canDuplicateSelectedStyle()" on page 425

dreamweaver.cssStylePalette.editSelectedStyle()

Availability

Dreamweaver 3

Description

Opens the Style Definition dialog box for the style that is currently selected in the CSS Styles panel.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.cssStyle.canEditSelectedStyle()" on page 425

dreamweaver.cssStylePalette.editStyleSheet()

Availability

Dreamweaver 3

Description

Opens the Edit Style Sheet dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.cssStylePallette.canEditStyleSheet()" on page 425

dreamweaver.cssStylePalette.getSelectedStyle()

Availability

Dreamweaver 3; `fullSelector` available in Dreamweaver MX

Description

Gets the name of the style that is currently selected in the CSS Styles panel.

Arguments

`fullSelector`

`fullSelector` is a Boolean value that indicates whether the full selector or only the class should be returned. If nothing is specified, only the class name returns. For instance, `p.class1` is a selector that means the style is applied to any `p` tag of `class1`, but it does not apply for instance to a `div` tag of `class1`. Without `fullSelector`, `getSelectedStyle` returns only the class name, `class1`, for the selector. `fullSelector` tells the function to return `p.class1` instead of `class1`.

Returns

When `fullSelector` is `true`, returns either the full selector, or an empty string when the stylesheet node is selected.

When `fullSelector` is `false` or omitted, a string that represents the class name of the selected style. If the selected style does not have a class or a stylesheet node is selected, an empty string returns.

Example

If the style `red` is selected, a call to `dw.cssStylePalette.getSelectedStyle()` returns `"red"`.

dreamweaver.cssStylePalette.getSelectedTarget()

Availability

Dreamweaver 3

Description

Gets the selected element in the Apply To pop-up menu at the top of the CSS Styles panel.

Arguments

None.

Returns

The object to which the style should be applied, or `NULL` if the target is the current selection.

Example

Before applying a style, use `dreamweaver.cssStylePalette.getSelectedTarget()` to ensure that if the user has changed the target, you have the one that is currently selected.

For example:

```
var currDOM = dreamweaver.getDocumentDOM();
currDOM.applyCSSStyle(dreamweaver.cssStylePalette.getSelectedTarget(), ¬
"codeRed");
```

dreamweaver.cssStylePalette.getStyles()

Availability

Dreamweaver 3

Description

Gets a list of all the class styles in the active document.

Arguments

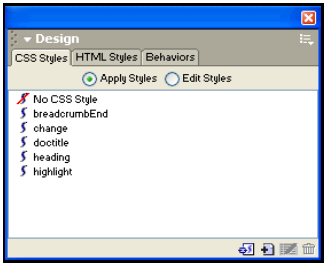
None.

Returns

An array of strings that represent the names of all the class styles in the document.

Example

Assuming the CSS Styles panel setup that is shown in the following example, a call to `dreamweaver.cssStylePalette.getStyles()` returns an array that contains these strings: `"BreadcrumbEnd"`, `"change"`, `"doctitle"`, `"heading"`, and `"highlight"`.



dreamweaver.cssStylePalette.newStyle()

Availability

Dreamweaver 3

Description

Opens the New Style dialog box.

Arguments

None.

Returns

Nothing.

Data source functions

Data source files are stored in the Configuration/DataSources folder. Each server model has its own folder: ASP.Net/C#, ASP.Net/VisualBasic, ASP/JavaScript, ASP/VBScript, ColdFusion, JSP, and PHP/MySQL. Within each server model subfolder are HTML and EDML files that are associated with the data sources for that server model.

For more information about using data sources in Dreamweaver, see "Data Sources" on page 191.

dreamweaver.dbi.getDataSources

Availability

Dreamweaver UltraDev 4

Description

Calls the `findDynamicSources()` function for each file in the Configuration/DataSources folder. You can use this function to generate a list of all the data sources in the user's document. This function iterates through all the files in the Configuration/DataSources folder, calls the `findDynamicSources()` function in each file, concatenates all the returned arrays, and returns the concatenated array of data sources.

Arguments

None.

Returns

The array that this function returns contains a concatenated list of all the data sources in the user's document. Each element in the array is an object, and each object has the following properties:

- The `title` property is the label string that appears to the right of the icon for each parent node. The `title` property is always defined.
- The `imageFile` property is the path of a file that contains the icon (a GIF image) that represents the parent node in the Bindings panel or Dynamic Data dialog box or Dynamic Text dialog box. The `imageFile` property is always defined.
- The `allowDelete` property is an optional property. If this property is set to `false`, when the user clicks on this node in the Bindings panel, the minus (-) button is disabled. If it is set to `true`, the minus (-) button is enabled. If the property is not defined, the minus (-) button is enabled when the user clicks on the item (as if the property is set to `true`).

- The `dataSource` property is the simple name of the file in which the `findDynamicSources()` function is defined. For example, the `findDynamicSources()` function in Configuration/DataSources/ASP_Js/Session.htm would set the `dataSource` property to `session.htm`. This is always defined.
- The `name` property is the name of the server behavior associated with the data source, `dataSource`, if one exists. The `name` property is always defined, but can be an empty string (`""`) if no server behavior is associated with the data source (such as a session variable).

Enablers

Enabler functions determine whether to enable menu items based on whether Dreamweaver can perform specific operations in the current context. The function specifications describe the general circumstances under which each function returns a value of `true`. However, the descriptions are not intended to be comprehensive and might exclude some cases in which the function would return a value of `false`.

dom.canAlign()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Align Left, Align Right, Align Top, or Align Bottom operation.

Arguments

None.

Returns

A Boolean value that indicates whether two or more layers or hotspots are selected.

dom.canApplyTemplate()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Apply To Page operation. This function is valid only for the active document.

Arguments

None.

Returns

A Boolean value that indicates whether the document is not a library item or a template, and that the selection is not within the `NOFRAMES` tag.

dom.canArrange()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Bring to Front or Move to Back operation.

Arguments

None.

Returns

A Boolean value that indicates whether a hotspot is selected.

dom.canClipCopyText()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Copy as Text operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is a range (that is, not an insertion point).

dom.canClipPaste()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Paste operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Clipboard contains any content that can be pasted into Dreamweaver.

dom.canClipPasteText()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Paste as Text operation.

Arguments

None.

Returns

A Boolean value that indicates whether the clipboard contains any content that can be pasted into Dreamweaver as text.

dom.canConvertLayersToTable()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Convert Layers to Table operation.

Arguments

None.

Returns

A Boolean value that indicates whether all content in the `BODY` of the document is contained within layers.

dom.canConvertTablesToLayers()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Convert Tables to Layers operation.

Arguments

None.

Returns

A Boolean value that indicates whether all the content in the `BODY` of the document is contained within tables, and the document is not based on a template.

dom.canDecreaseColspan()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Decrease Colspan operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current cell has a `COLSPAN` attribute, and whether that attribute's value is greater than or equal to 2.

dom.canDecreaseRowspan()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Decrease Rowspan operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current cell has a `ROWSPAN` attribute, and whether that attribute's value is greater than or equal to 2.

dom.canDeleteTableColumn()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Delete Column operation.

Arguments

None.

Returns

A Boolean value that indicates whether the insertion point is inside a cell, or if a cell or column is selected.

dom.canDeleteTableRow()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Delete Row operation.

Arguments

None.

Returns

A Boolean value that indicates whether the insertion point is inside a cell, or if a cell or row is selected.

dom.canEditNoFramesContent()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Edit No Frames Content operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current document is a frameset or within a frameset.

dom.canIncreaseColspan()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Increase Colspan operation.

Arguments

None.

Returns

A Boolean value that indicates whether there are any cells to the right of the current cell.

dom.canIncreaseRowspan()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Increase Rowspan operation.

Arguments

None.

Returns

A Boolean value that indicates whether there are any cells below the current cell.

dom.canInsertTableColumns()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Insert Column(s) operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is inside a table. This function returns `false` if the selection is an entire table.

dom.canInsertTableRows()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Insert Row(s) operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is inside a table. This function returns `false` if the selection is an entire table.

dom.canMakeNewEditableRegion()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a New Editable Region operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current document is a template (.dwt) file.

dom.canMarkSelectionAsEditable()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Mark Selection as Editable operation.

Arguments

None.

Returns

A Boolean value that indicates whether there is a selection, and whether the current document is a template (.dwt) file.

dom.canMergeTableCells()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Merge Cells operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is an adjacent grouping of table cells.

dom.canPlayPlugin()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Play operation. This function is valid only for the active document.

Arguments

None.

Returns

A Boolean value that indicates whether the selection can be played with a plug-in.

dom.canRedo()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Redo operation.

Arguments

None.

Returns

A Boolean value that indicates whether any steps remain to redo.

dom.canRemoveEditableRegion()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Unmark Editable Region operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current document is a template.

dom.canSelectTable()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Select Table operation.

Arguments

None.

Returns

A Boolean value that indicates whether the insertion point or selection is within a table.

dom.canSetLinkHref()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can change the link around the current selection or create one if necessary.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is an image, text, or an insertion point inside a link. A text selection is defined as a selection for which the text Property inspector would appear.

dom.canShowListPropertiesDialog()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can show the List Properties dialog box.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is within an `LI` tag.

dom.canSplitFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Split Frame [Left | Right | Up | Down] operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is within a frame.

dom.canSplitTableCell()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Split Cell operation.

Arguments

None.

Returns

A Boolean value that indicates whether the insertion point is inside a table cell or the selection is a table cell.

dom.canStopPlugin()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Stop operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is currently being played with a plug-in.

dom.canUndo()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Undo operation.

Arguments

None.

Returns

A Boolean value that indicates whether any steps remain to undo.

dom.hasTracingImage()

Availability

Dreamweaver 3

Description

Checks whether the document has a tracing image.

Arguments

None.

Returns

A Boolean value that indicates whether the document has a tracing image.

dreamweaver.assetPalette.canEdit()

Availability

Dreamweaver 4

Description

Enables menu items in the Assets panel for editing.

Arguments

None.

Returns

Returns `true` if the asset can be edited; `false` otherwise. Returns `false` for colors and URLs in the Site list, and returns `false` for a multiple selection of colors and URLs in the Favorites list.

dreamweaver.assetPalette.canInsertOrApply()

Availability

Dreamweaver 4

Description

Checks if the selected elements can be inserted or applied. Returns `true` or `false` so the menu items can be enabled or disabled for insertion or application.

Arguments

None.

Returns

Returns `false` if the current page is a template, and the current category is Templates. Returns `false` if no document is open. Returns `false` if a library item is selected in the document and the current category is Library. Otherwise returns `true`.

dreamweaver.canClipCopy()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Copy operation.

Arguments

None.

Returns

A Boolean value that indicates whether there is any content selected that can be copied to the Clipboard.

dreamweaver.canClipCut()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Cut operation.

Arguments

None.

Returns

A Boolean value that indicates whether there is any content selected that can be cut to the Clipboard.

dreamweaver.canClipPaste()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Paste operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Clipboard contains any content that can be pasted into the current document or the active pane in the Site panel; or, on the Macintosh, an text field in a floating panel or dialog box.

dreamweaver.canDeleteSelection()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can delete the current selection. Depending on the window that has focus, the deletion may occur in the Document window or the Site panel; or, on the Macintosh, in an text field in a dialog box or floating panel.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is a range (that is, not an insertion point).

dreamweaver.canExportCSS()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Export CSS Styles operation.

Arguments

None.

Returns

A Boolean value that indicates whether the document contains any class styles that are defined in the `HEAD`.

dreamweaver.canExportTemplateDataAsXML()

Availability

Dreamweaver MX

Description

Checks whether Dreamweaver can export the current document as XML.

Arguments

None.

Returns

`true` if you can perform an export on the current document; `false` otherwise.

Example

```
if(dreamweaver.canExportTemplateDataAsXML())
{
  dreamweaver.exportTemplateDataAsXML("file:///c|/dw_temps/mytemplate.txt")
}
```

dreamweaver.canFindNext()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Find Next operation.

Arguments

None.

Returns

A Boolean value that indicates whether a search pattern has already been established.

dreamweaver.canOpenInFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Open in Frame operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection or insertion point is within a frame.

dreamweaver.canPlayRecordedCommand()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Play Recorded Command operation.

Arguments

None.

Returns

A Boolean value that indicates whether there is an active document and a previously recorded command that can be played.

dreamweaver.canPopupEditTagDialog()

Availability

Dreamweaver MX

Description

Checks whether the current selection is a tag and whether the Edit Tag menu item is active.

Arguments

None.

Returns

The name of the currently selected tag, or `null` if no tag is selected.

dreamweaver.canRedo()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Redo operation in the current context.

Arguments

None.

Returns

A Boolean value that indicates whether any operations can be undone.

dreamweaver.canRevertDocument()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Revert (to the last-saved version) operation.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree (the value that `dreamweaver.getDocumentDOM()` returns).

Returns

A Boolean value that indicates whether the document is in an unsaved state and a saved version of the document exists on a local drive.

dreamweaver.canSaveAll()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Save All operation.

Arguments

None.

Returns

A Boolean value that indicates whether one or more unsaved documents are open.

dreamweaver.canSaveDocument()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Save operation on the specified document.

Arguments

documentObject

documentObject is the root of a document's DOM (the same value that `dreamweaver.getDocumentDOM()` returns).

Returns

A Boolean value that indicates whether the document has any unsaved changes.

dreamweaver.canSaveDocumentAsTemplate()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Save As Template operation on the specified document.

Arguments

documentObject

documentObject is the root of a document's DOM (the same value that `dreamweaver.getDocumentDOM()` returns).

Returns

A Boolean value that indicates whether the document can be saved as a template.

dreamweaver.canSaveFrameset()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Save Frameset operation on the specified document.

Arguments

documentObject

documentObject is the root of a document's DOM (the same value that `dreamweaver.getDocumentDOM()` returns).

Returns

A Boolean value that indicates whether the document is a frameset with unsaved changes.

dreamweaver.canSaveFramesetAs()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Save Frameset As operation on the specified document.

Arguments

documentObject

documentObject is the root of a document's DOM (the same value that `dreamweaver.getDocumentDOM()` returns).

Returns

A Boolean value that indicates whether the document is a frameset.

dreamweaver.canSelectAll()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Select All operation.

Arguments

None.

Returns

A Boolean value that indicates whether a Select All operation can be performed.

dreamweaver.canShowFindDialog()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Find operation.

Arguments

None.

Returns

A Boolean value that indicates whether a Site panel or a Document window is open. This function returns `false` when the selection is in the `HEAD`.

dreamweaver.canUndo()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Undo operation in the current context.

Arguments

None.

Returns

A Boolean value that indicates whether any operations can be undone.

dreamweaver.cssStylePalette.canApplySelectedStyle()

Availability

Dreamweaver MX

Description

Checks the current active document to see whether the selected style can be applied.

Arguments

None.

Returns

A Boolean value: `true` if the selected style has a class selector; `false` otherwise.

dreamweaver.cssStylePalette.canDeleteSelectedStyle()

Availability

Dreamweaver MX

Description

Checks the current selection to determine whether the selected style can be deleted.

Arguments

None.

Returns

A Boolean value: `true` if the selection can be deleted; `false` otherwise.

dreamweaver.cssStylePalette.canDuplicateSelectedStyle()

Availability

Dreamweaver MX

Description

Checks the current active document to see whether the selected style can be duplicated.

Arguments

None.

Returns

A Boolean value: `true` if the selected style can be duplicated; `false` otherwise.

dreamweaver.cssStyle.canEditSelectedStyle()

Availability

Dreamweaver MX

Description

Checks the current active document to see whether the selected style can be edited.

Arguments

None.

Returns

A Boolean value: `true` if the selected style is editable; `false` otherwise.

dreamweaver.cssStylePallette.canEditStyleSheet()

Availability

Dreamweaver MX

Description

Checks the current selection to see whether it contains style sheet elements that can be edited.

Arguments

None.

Returns

A Boolean value: `true` if the selection is a stylesheet node or a style definition within a stylesheet node and the stylesheet is neither "hidden" nor "This Document".

A Boolean value `false` if the selection is "hidden" or in "This Document".

dreamweaver.isRecording()

Availability

Dreamweaver 3

Description

Reports whether Dreamweaver is currently recording a command.

Arguments

None.

Returns

A Boolean value that indicates whether Dreamweaver is recording a command.

dreamweaver.htmlStylePalette.canEditSelection()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can edit, delete, or duplicate the selection in the HTML Styles panel.

Arguments

None.

Returns

A Boolean value: `false` if no style is selected or if one of the "clear" styles is selected.

dreamweaver.resultsPalette.canClearItems()

Availability

Dreamweaver MX

Description

Checks whether you can clear the contents of the Results panel currently in focus.

Arguments

None

Return Value

A Boolean value: `true` if the contents can be cleared; `false` otherwise.

dreamweaver.resultsPalette.canClipCopy()

Availability
Dreamweaver MX

Description
Checks whether the current Results window can display a copied message in its contents.

Arguments
None.

Return Value
A Boolean value: `true` if the contents can be displayed; `false` otherwise.

dreamweaver.resultsPalette.canClipCut()

Availability
Dreamweaver MX

Description
Checks whether the current Results window can display a cut message in its contents.

Arguments
None.

Return Value
A Boolean value: `true` if the contents can be displayed; `false` otherwise.

dreamweaver.resultsPalette.canClipPaste()

Availability
Dreamweaver MX

Description
Checks whether the current Results window can display a paste message in its contents.

Arguments
None.

Return Value
A Boolean value: `true` if the contents can be displayed; `false` otherwise.

dreamweaver.resultsPalette.canOpenInBrowser()

Availability
Dreamweaver MX

Description
Checks whether the current report can be displayed in a browser.

Arguments
None.

Return Value

A Boolean value: `true` if the contents can be displayed; `false` otherwise.

dreamweaver.resultsPalette.canOpenInEditor()

Availability

Dreamweaver MX

Description

Checks whether the current report can be displayed in an editor.

Arguments

None.

Return Value

A Boolean value: `true` if the contents can display; `false` otherwise.

dreamweaver.resultsPalette.canSave()

Availability

Dreamweaver MX

Description

Checks whether the Save dialog box can launch for the current panel. Currently, the Site Reports, Target Browser Check, Validation, and Link Checker panels support the Save dialog box.

Arguments

None.

Return Value

A Boolean value: `true` if the Save dialog box can appear; `false` otherwise.

dreamweaver.resultsPalette.canSelectAll()

Availability

Dreamweaver MX

Description

Checks whether a Select All message can be sent to the current window in focus.

Arguments

None.

Return Value

A Boolean value: `true` if the Select All message can be sent; `false` otherwise.

dreamweaver.snippetpalette.canEditSnippet()

Availability

Dreamweaver MX

Description

Checks whether you can edit the currently selected item and returns `true` or `false` so that you can enable or disable menu items for editing.

Arguments

None.

Return Value

A Boolean value.

dw.snippetpalette.canInsert()

Availability

Dreamweaver MX

Description

Checks whether you can insert or apply the selected element and returns `true` or `false` so you can enable or disable menu items for inserting or applying

Arguments

None.

Return Values

A Boolean value.

dreamweaver.tagInspector.tagBeforeEnabled()

Availability

Dreamweaver MX

Description

Checks whether the `dreamweaver.treeViewPalette.tagBefore()` function can be called.

Arguments

None.

Returns

`true` if exactly one tag is selected.

dreamweaver.tagInspector.tagInsideEnabled()

Availability

Dreamweaver MX

Description

Checks whether the `dreamweaver.treeViewPalette.tagInside()` function can be called.

Arguments

None.

Returns

`true` if exactly one tag is selected.

dreamweaver.tagInspector.tagAfterEnabled()

Availability

Dreamweaver MX

Description

Checks whether the `dreamweaver.treeViewPalette.tagAfter()` function can be called.

Arguments

None.

Returns

`true` if exactly one tag is selected.

dreamweaver.tagInspector.deleteTagsEnabled()

Availability

Dreamweaver MX

Description

Checks whether the `dreamweaver.TreeViewPalette.deleteTags()` function can be called.

Arguments

None.

Returns

`true` if one or more tags are selected.

dreamweaver.tagInspector.editTagNameEnabled()

Availability

Dreamweaver MX

Description

Checks whether the `dreamweaver.treeViewPalette.editTagName()` function can be called.

Arguments

None.

Returns

`true` if exactly one tag is selected.

dreamweaver.timelineInspector.canAddFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Add Frame operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Timelines panel has any animation bars or behaviors.

dreamweaver.timelineInspector.canAddKeyFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Add Keyframe operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection in the Timelines panel is part of an animation bar.

dreamweaver.timelineInspector.canChangeObject()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Change Object operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection in the Timelines panel is part of an animation bar.

dreamweaver.timelineInspector.canRemoveBehavior()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Remove Behavior operation.

Arguments

None.

Returns

A Boolean value that indicates whether the selection in the Timelines panel is a behavior.

dreamweaver.timelineInspector.canRemoveFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Remove Frame operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Timelines panel has any animation bars or behaviors.

dreamweaver.timelineInspector.canRemoveKeyFrame()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Remove Keyframe operation.

Arguments

None.

Returns

A Boolean value that indicates whether the current frame in the Timelines panel is a keyframe.

dreamweaver.timelineInspector.canRemoveObject()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Remove Object operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Timelines panel has any animation bars.

site.browseDocument()

Availability

Dreamweaver 4

Description

Opens all selected documents in a browser window; same as using the Preview in Browser command.

Arguments

browserName

browserName is the name of a browser as defined in the Preview in Browser preferences. If omitted, this argument defaults to the user's primary browser.

Returns

Nothing.

site.canAddLink()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform an Add Link to [Existing File | New File] operation.

Arguments

None.

Returns

A Boolean value that indicates that the selected document in the site map is an HTML file.

site.canChangeLink()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Change Link operation.

Arguments

None.

Returns

A Boolean value that indicates that an HTML or Flash file links to the selected file in the site map.

site.canCheckIn()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Check In operation.

Arguments

siteOrURL

siteOrURL must be the `site` keyword, which indicates that the function should act on the selection in the Site panel or the URL for a single file.

Returns

A Boolean value that indicates whether all the following conditions are true:

- A remote site has been defined.
- If a Document window has focus, the file has been saved in a local site; or, if the Site panel has focus, one or more files or folders are selected.
- Check In/Check Out is turned on.

site.canCheckOut()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Check Out operation on the specified file or files.

Arguments

siteOrURL

siteOrURL must be the `site` keyword, which indicates that the function should act on the selection in the Site panel or the URL for a single file.

Returns

A Boolean value that indicates whether all the following conditions are true:

- A remote site has been defined.
- If a Document window has focus, the file is part of a local site and is not already checked out; or, if the Site panel has focus, one or more files or folders are selected and at least one of the selected files is not already checked out.
- Check In/Check Out is turned on.

site.canCloak()

Availability

Dreamweaver MX

Description

Determines whether Dreamweaver can perform a cloaking operation.

Arguments

siteOrURL

siteOrURL must be the `site` keyword, which indicates that `canCloak()` should act on the selection in the Site panel or the URL of a particular folder, which indicates that `canCloak()` should act on the specified folder and all its contents.

Returns

`true` if Dreamweaver can perform the cloaking operation on the current site or the specified folder.

site.canConnect()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can connect to the remote site.

Arguments

None.

Returns

A Boolean value that indicates whether the current remote site is an FTP site.

site.canFindLinkSource()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Find Link Source operation.

Arguments

None.

Returns

A Boolean value that indicates that the selected link in the site map is not the home page.

site.canGet()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Get operation.

Arguments

`siteOrURL`

`siteOrURL` must be the `site` keyword, which indicates that the function should act on the selection in the Site panel or the URL for a single file.

Returns

If the argument is site, a Boolean value that indicates whether one or more files or folders is selected in the Site panel and a remote site has been defined. If the argument is a URL, a Boolean value that indicates whether the document belongs to a site for which a remote site has been defined.

site.canLocateInSite()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Locate in Local Site or Locate in Remote Site operation (depending on the argument).

Arguments

localOrRemote, *siteOrURL*

- *localOrRemote* must be either `local` or `remote`.
- *siteOrURL* must be the `site` keyword, which indicates that the function should act on the selection in the Site panel or the URL for a single file.

Returns

One of the following values:

- If the first argument is `local` and the second argument is a URL, a Boolean value that indicates whether the document belongs to a site.
- If the first argument is `remote` and the second argument is a URL, a Boolean value that indicates whether the document belongs to a site for which a remote site has been defined, and, if the server type is Local/Network, whether the drive is mounted.
- If the second argument is `site`, a Boolean value that indicates whether both panes contain site files (not the site map) and whether the selection is in the opposite pane from the argument.

site.canMakeEditable()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Turn Off Read Only operation.

Arguments

None.

Returns

A Boolean value that indicates whether one or more of the selected files is locked.

site.canMakeNewFileOrFolder()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a New File or New Folder operation in the Site panel.

Arguments

None.

Returns

A Boolean value that indicates whether any files are visible in the selected pane of the Site panel.

site.canOpen()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can open the files or folders that are currently selected in the Site panel.

Arguments

None.

Returns

A Boolean value that indicates whether any files or folders are selected in the Site panel.

site.canPut()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Put operation.

Arguments

siteOrURL

siteOrURL must be the `site` keyword, which indicates that the function should act on the selection in the Site panel, or the URL for a single file.

Returns

If the argument is `site`, a Boolean value that indicates whether any files or folders are selected in the Site panel and a remote site has been defined. If the argument is a URL, a Boolean value that indicates whether the document belongs to a site for which a remote site has been defined.

site.canRecreateCache()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Recreate Site Cache operation.

Arguments

None.

Returns

A Boolean value that indicates whether the Use Cache To Speed Link Updates option is enabled for the current site.

site.canRefresh()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Refresh [Local | Remote] operation.

Arguments

localOrRemote

localOrRemote must be either `local` or `remote`.

Returns

A value of `true` if *localOrRemote* is `local`; otherwise, returns a Boolean value that indicates whether a remote site has been defined.

site.canRemoveLink()

Availability

Dreamweaver 3

Description

Checks whether Dreamweaver can perform a Remove Link operation.

Arguments

None.

Returns

A Boolean value that indicates that an HTML or Flash file links to the selected file in the site map.

site.canSetLayout()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Layout operation.

Arguments

None.

Returns

A Boolean value that indicates whether the site map is visible.

site.canSelectAllCheckedOutFiles()

Availability

Dreamweaver 4

Description

Determines whether the current working site has Check In/Check Out enabled.

Arguments

None.

Returns

A Boolean value: `true` if the site allows Check In/Check Out; `false` otherwise.

site.canSelectNewer()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Select Newer [Remote | Local] operation.

Arguments

`localOrRemote`

`localOrRemote` must be either `local` or `remote`.

Returns

A Boolean value that indicates whether the document belongs to a site for which a remote site has been defined.

site.canShowPageTitles()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Show Page Titles operation.

Arguments

None.

Returns

A Boolean value that indicates whether the site map is visible.

site.canSynchronize()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a Synchronize operation.

Arguments

None.

Returns

A Boolean value that indicates whether a remote site has been defined.

site.canUncloak()

Availability

Dreamweaver MX

Description

Determines whether Dreamweaver can perform an uncloaking operation.

Arguments

siteOrURL

siteOrURL must be the keyword `site`, which indicates that `canUncloak()` should act on the selection in the Site panel or the URL of a particular folder, which indicates that `canUncloak()` should act on the specified folder and all its contents.

Returns

A value of `true` if Dreamweaver can perform the uncloaking operation on the current site or the specified folder.

site.canUndoCheckOut()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform an Undo Check Out operation.

Arguments

siteOrURL

siteOrURL must be the `site` keyword, which indicates that the function should act on the selection in the Site panel or the URL for a single file.

Returns

A Boolean value that indicates whether the specified file or at least one of the selected files is checked out.

site.canViewAsRoot()

Availability

Dreamweaver 3

Description

Determines whether Dreamweaver can perform a View as Root operation.

Arguments

None.

Returns

A Boolean value that indicates whether the specified file is an HTML or Flash file.

External application functions

External application functions handle operations that are related to the Macromedia Flash MX application and to the browsers and external editors that are defined in the Preview in Browser and External Editors preferences. These functions let you get information about these external applications and open files with them.

dreamweaver.browseDocument()

Availability

Dreamweaver 2; enhanced in 3 and 4.

Description

Opens the specified URL in the specified browser.

Arguments

fileName {,browser}

- *fileName* is the name of the file to open, which is expressed as an absolute URL.
- *browser*, added in Dreamweaver 3, specifies a browser. This argument can be the name of a browser, as defined in the Preview in Browser preferences or either `'primary'` or `'secondary'`. If omitted, the URL opens in the user's primary browser.

Returns

Nothing.

Example

The following function uses `dreamweaver.browseDocument()` to open the Hotwired home page in a browser:

```
function goToHotwired(){
  dreamweaver.browseDocument('http://www.hotwired.com/');
}
```

In Dreamweaver 4, you can expand this operation to open the document in Microsoft Internet Explorer using the following code:

```
function goToHotwired(){
  var prevBrowsers = dw.getBrowserList();
  var theBrowser = "";
  for (var i=1; i < prevBrowsers.length; i+2){
    if (prevBrowsers[i].indexOf('Iexplore.exe') != -1){
      theBrowser = prevBrowsers[i];
      break;
    }
  }
  dw.browseDocument('http://www.hotwired.com/',theBrowser);
}
```

For more information on `dw.getBrowserList()`, see "dreamweaver.getBrowserList()" on page 442.

dreamweaver.getBrowserList()

Availability

Dreamweaver 3

Description

Gets a list of all the browsers in the Preview in Browser submenu

Arguments

None.

Returns

An array that contains a pair of strings for each browser in the list. The first string in each pair is the name of the browser, and the second string is its location on the user's computer, which is expressed as a file:// URL. If no browsers appear in the submenu, the function returns nothing.

dreamweaver.getExtensionEditorList()

Availability

Dreamweaver 3

Description

Gets a list of editors for the specified file from the External Editors preferences

Arguments

fileURL

fileURL can be a complete file:// URL, a filename, or a file extension (including the period).

Returns

An array that contains a pair of strings for each editor in the list. The first string in each pair is the name of the editor, and the second string is its location on the user's computer, which is expressed as a file:// URL. If no editors appear in the preferences, the function returns an array of one empty string.

Example

A call to `dreamweaver.getExtensionEditorList(".gif")` might return an array that contains the following strings:

- `"Fireworks 3"`
- `"file:///C|/Program Files/Macromedia/Fireworks 3/Fireworks 3.exe"`

dreamweaver.getExternalTextEditor()

Availability

Dreamweaver 4

Description

Gets the name of the currently configured external text editor.

Arguments

None.

Returns

A string that contains the name of the text editor that is suitable for presentation in the UI, not the full path.

dreamweaver.getFlashPath()

Availability

Dreamweaver MX

Description

Gets the full path to the Flash MX application in the form of a file URL.

Arguments

None.

Returns

An array that contains two elements. Element [0] is a string that contains the name of the Flash MX editor. Element [1] is a string that contains the path to the Flash application on the local computer, which is expressed as a file:// URL. If Flash is not installed, it returns nothing.

Example

```
var myDoc = dreamweaver.getDocumentDOM();

if ( dreamweaver.validateFlash() ) {
  var flashArray = dreamweaver.getFlashPath();
  dreamweaver.openWithApp( myDoc.myForm.swfFilePath, flashArray[1] );
}
```

dreamweaver.getPrimaryBrowser()

Availability

Dreamweaver 3

Description

Gets the path to the primary browser.

Arguments

None.

Returns

A string that contains the path on the user's hard drive to the primary browser, which is expressed as a file:// URL. If no primary browser is defined, it returns nothing.

dreamweaver.getPrimaryExtensionEditor()

Availability

Dreamweaver 3

Description

Gets the primary editor for the specified file.

Arguments

fileURL

Returns

An array that contains a pair of strings. The first string in the pair is the name of the editor, and the second string is its location on the user's computer, which is expressed as a file:// URL. If no primary editor is defined, the function returns an array of one empty string.

dreamweaver.getSecondaryBrowser()

Availability

Dreamweaver 3

Description

Gets the path to the secondary browser.

Arguments

None.

Returns

A string that contains the path on the user's hard disk to the secondary browser, which is expressed as a file:// URL. If no secondary browser is defined, it returns nothing.

dreamweaver.openHelpURL()

Availability

Dreamweaver MX

Description

Opens the specified Help file in the operating system Help viewer.

Dreamweaver MX displays help content in the standard operating system help viewer instead of a browser. Help content is in HTML, but it is packaged for Windows HTML Help or Help Viewer for Mac OS 9 and OS X.

The following four types of files comprise the full help content. See your operating system documentation for more information on help files.

- Help book

  A help book consists of the HTML help files, images, and indexes. In Windows, the help book is a file that has a name with a .chm extension. On the Macintosh, the help book is a folder.

  The Help book files reside in the Dreamweaver MX Help folder.

- help.xml

  The help.xml file maps book IDs to Help book names. For example, the following XML code maps the book ID for Dreamweaver MX Help to the filename that contains that help.

```
<?xml version = "1.0" ?>
<help-books>
<book-id id="DW_Using" win-mapping="UsingDreamweaver.chm" mac-
mapping="Dreamweaver Help"/>
</help-books>
```

  Each `book-id` entry has the following attributes:

  `id` is the book id that is used in the help.map and HelpDoc.js files

  `win-mapping` is the Windows book name, which is `"UsingDreamweaver.chm"` in this example.

  `mac-mapping` is the Macintosh book name, which is `"Dreamweaver Help"` in this example.

- help.map

  The help.map file maps a Help content ID to a specific help book. Dreamweaver MX uses the help.map file to locate specific Help content when it calls Help internally.

- helpDoc.js

The helpDoc.js file lets you map variable names that you can use in place of the actual book ID and page string. The helpDoc.js file maps a help content ID to an HTML page in a specific help book. Dreamweaver MX uses the helpDoc.js file when it calls help from JavaScript.

Arguments

`bookID` Required. Specifies the name of the file from which Dreamweaver displays help, which is expressed as a book ID in the help.xml file, followed by a colon, followed by the page in the book.

Returns

`true` if successful; `false` if Dreamweaver cannot open the specified file in the help viewer.

Example

```
openHelpURL("DW_Using:index.htm");
```

dreamweaver.openWithApp()

Availability

Dreamweaver 3

Description

Opens the specified file with the specified application.

Arguments

`fileURL`, *`appURL`*

- *`fileURL`* is the path to the file to open, which is expressed as a file:// URL.
- *`appURL`* is the path to the application that is to open the file, which is expressed as a file:// URL.

Returns

Nothing.

dreamweaver.openWithBrowseDialog()

Availability

Dreamweaver 3

Description

Opens the Select External Editor dialog box to let the user choose the application with which to open the specified file.

Arguments

fileURL

Returns

Nothing.

dreamweaver.openWithExternalTextEditor()

Availability

Dreamweaver 3

Description

Opens the current document in the external text editor that is specified in the External Editors preferences.

Arguments

None.

Returns

Nothing.

dreamweaver.openWithImageEditor()

Availability

Dreamweaver 3

Description

Opens the named file with the specified image editor.

Note: This function invokes a special Fireworks integration mechanism that returns information to the active document if Fireworks is specified as the image editor. To prevent errors if no document is active, this function should never be called from the Site panel.

Arguments

fileURL, *appURL*

- *fileURL* is the path to the file to open, which is expressed as a file:// URL.
- *appURL* is the path to the application with which to open the file, which is expressed as a file:// URL.

Returns

Nothing.

dreamweaver.validateFlash()

Availability

Dreamweaver MX

Description

Determines whether Flash MX (or a later version) is installed on the local computer.

Arguments

None.

Returns

A Boolean value. This function returns `true` if Flash MX (or a later version) is installed on the local computer; otherwise, it returns `false`.

File manipulation functions

File manipulation functions handle creating, opening, and saving documents (including XML and XHTML), converting existing HTML documents into XHTML, and exporting cascading style sheets (CSS) to external files. These functions accomplish such tasks as browsing for files or folders, creating files based on templates, closing documents, and getting information about recently opened files.

dom.cleanupXHTML()

Availability

Dreamweaver MX

Description

Similar to `convertToXHTML()` but cleans up an existing XHTML document. This function can run on a selection within the document. You can run the `cleanupXHTML()` method to clean up the syntax in an entire XHTML document or in the current selection of a document.

Arguments

`bWholeDoc`

`bWholeDoc` holds a Boolean value. If the value is `true`, `cleanupXHTML()` cleans up the entire document; otherwise, this function cleans up only the selection.

Returns

An array of six integers that quantify the number of the following elements:

- XHTML errors that Dreamweaver fixed
- `map` elements that do not have an `id` attribute and could not be fixed
- `script` elements that do not have a `type` attribute and could not be fixed
- `style` elements that do not have a `type` attribute and could not be fixed
- `img` elements that do not have an `alt` attribute and could not be fixed
- `area` elements that do not have an `alt` attribute and could not be fixed

dom.convertToXHTML()

Availability

Dreamweaver MX

Description

Parses the HTML into a DOM tree, inserts missing items that are required for XHTML, cleans up the tree, and then writes the tree as clean XHTML. The missing directives, declarations, elements and attributes that `convertToXHTML()` adds to the DOM tree, as necessary, include the following items:

- An XML directive
- A `doctype` declaration
- The `xmlns` attribute in the `html` element
- A `head` section
- A `title` element
- A `body` section

During the conversion, `dom.convertToXHTML()` converts pure HTML tags and attributes to lowercase, writes HTML tags and attributes with correct XHTML syntax, and adds missing HTML attributes where it can. This function treats third-party tags and attributes according to the settings in the Preferences dialog box.

If the document is a template, `dom.convertToXHTML()` alerts the user but does not perform the conversion.

Arguments

None.

Returns

An array of six integers that quantify the following items:

- XHTML errors that Dreamweaver fixed
- `map` elements that do not have an `id` attribute and cannot be fixed
- `script` elements that do not have a `type` attribute and cannot be fixed
- `style` elements that do not have a `type` attribute and cannot be fixed
- `img` elements that do not have an `alt` attribute and cannot be fixed
- `area` elements that do not have an `alt` attribute and cannot be fixed

Example

In normal use, an extension first calls `dreamweaver.openDocument()` or `dreamweaver.getDocumentDOM()` to get a reference to the document. The extension then calls `dom.getIsXHTMLDocument()` to determine whether the document is already in XHTML form. If it is not, the extension calls `dom.convertToXHTML()` to convert the document into XHTML. Then the extension calls `dreamweaver.saveDocument()` to save the converted file with a new filename.

dom.getIsXHTMLDocument()

Availability

Dreamweaver MX

Description

Checks a document (specifically, the `<!DOCTYPE>` declaration) to see whether it is XHTML.

Arguments

None.

Returns

`true` if the document is XHTML; `false`, otherwise

dreamweaver.browseForFileURL()

Availability

Dreamweaver 1, enhanced in 2, 3, and 4

Description

Opens the specified type of dialog box with the specified label in the title bar.

Arguments

openSelectOrSave *{*, *titleBarLabel*} *{*, *bShowPreviewPane*} ¬
{, *bSupressSiteRootWarnings*} *{*, *arrayOfExtensions*}

- *openSelectOrSave* indicates the type of dialog box: `open`, `select`, or `save`.
- *titleBarLabel* (added in Dreamweaver 2) is the label that should appear in the title bar of the dialog box. If this argument is omitted, Dreamweaver uses the default label that the operating system supplies.
- *bShowPreviewPane* (added in Dreamweaver 2) is a Boolean value that indicates whether to display the Image Preview Pane in the dialog box. If this argument is `true`, the dialog box filters for image files; if omitted, it defaults to `false`.
- *bSupressSiteRootWarnings* (added in Dreamweaver 3) is a Boolean value that indicates whether to suppress warnings about the selected file being outside the site root. If this argument is omitted, it defaults to `false`.
- *arrayOfExtensions* (added in Dreamweaver 4) is an array of strings for specifying the Files of type list menu default appearance at the bottom of the dialog box. The proper syntax is `menuEntryText|.xxx[;.yyy;.zzz]|CCCC|`, where *menuEntryText* is the name of the file type to appear. The extensions can be specified as *.xxx[;.yyy;.zzz]* or *CCCC*, where *.xxx* specifies the file extension for the file type (optionally, *.yyy* and *.zzz* specify multiple file extensions) and `CCCC` is the four-character file type constant for the Macintosh.

Returns

A string that contains the name of the file, which is expressed as a file:// URL.

dreamweaver.browseForFolderURL()

Availability

Dreamweaver 3

Description

Opens the Choose Folder dialog box with the specified label in the title bar.

Arguments

{titleBarLabel} {, directoryToStartIn}

- *titleBarLabel* is the label that should appear in the title bar of the dialog box. If it is omitted, *titleBarLabel* defaults to Choose Folder.
- *directoryToStartIn* is the path where the directory should start, which is expressed as a file:// URL.

Returns

A string that contains the name of the folder, which is expressed as a file:// URL.

Example

The following code returns the URL of a folder:

```
return dreamweaver.browseForFolderURL('Select a Folder', ¬
dreamweaver.getSiteRoot());
```

dreamweaver.closeDocument()

Availability

Dreamweaver 3

Description

Closes the specified document.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree (the value that `dreamweaver.getDocumentDOM()` returns). If *documentObject* refers to the active document, the Document window might not close until the script that calls this function finishes executing.

Returns

Nothing.

dreamweaver.createDocument()

Availability

Dreamweaver 2

Description

Depending on the argument that you pass to this function, it opens a new document either in the same window or in a new window. The new document becomes the active document.

Note: This function can be called only from menus.xml, a command, or the Property inspector file. If a behavior action or object tries to call this function, Dreamweaver displays an error message.

Arguments

{bOpenInSameWindow}

bOpenInSameWindow is a Boolean value that indicates whether to open the new document in the current window. If *bOpenInSameWindow* is `false` or omitted, or the function is called on the Macintosh, the new document opens in a separate window.

Returns

The document object for the newly created document. This is the same value that `dreamweaver.getDocumentDOM()` returns.

dreamweaver.createXHTMLDocument()

Availability

Dreamweaver MX

Description

Depending on the argument that you pass to this function, it opens a new XHTML document either in the same window or in a new window. The new document becomes the active document. It is similar to `dreamweaver.createDocument()`.

When Dreamweaver creates a new XHTML document, Dreamweaver reads a file named `default.xhtml`, which is located in the Configurations/Templates folder, and, using the content of that file, creates an output file that contains the following skeleton declarations:

```
<?xml version="1.0">
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=" />
</head>

<body bgcolor="#FFFFFF" text="#000000">

</body>
</html>
```

The default DTD declaration is `XHTML 1.0 Transitional`, rather than `Strict`. If the user adds a frameset to the document, Dreamweaver switches the DTD to `XHTML 1.0 Frameset`. `Content-Type` is `text/html`, and `charset` is intentionally left out of the default.xhtml file but is filled in before the user views the new document. The `<?xml>` directive is not required if the document uses UTF-8 or UTF-16 character encoding; if it is present, it might be rendered by some older browsers. However, because this directive should be in an XHTML document, by default, Dreamweaver uses it (for both new and converted documents). Users can manually delete the directive. The `<?xml>` directive includes the encoding attribute, which matches the `charset` in the `Content-Type` attribute.

Arguments

`{bOpenInSameWindow}`

`bOpenInSameWindow` is a Boolean value that indicates whether to open the new document in the current window. If this value is `false` or omitted, or the function is called on the Macintosh, the new document opens in a separate window.

Returns

The document object for the newly created document, which is the same value that `dreamweaver.getDocumentDOM()` returns.

dreamweaver.createXMLDocument()

Availability

Dreamweaver MX

Description

Creates and opens a new XML file, which is empty except for the XML directive.

Arguments

None.

Returns

The DOM of the new XML file.

Example

The following example creates a new document, which is empty except for the XML directive:

```
var theDOM = dreamweaver.createXMLDocument("document");
```

dreamweaver.exportCSS()

Availability

Dreamweaver 3

Description

Opens the Export Styles as a CSS File dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canExportCSS()" on page 420

dreamweaver.exportTemplateDataAsXML()

Availability

Dreamweaver MX

Description

Exports the current document to the specified file as XML. This function operates on the document that has focus, which must be a template. If you do not specify a filename argument, Dreamweaver MX opens a dialog box to request the export file string.

Arguments

{filePath}

filePath Optional. A string that specifies the filename to which Dreamweaver exports the template. Express *filepath* as a URL file string, such as, `"file:///c|/temp/mydata.txt"`.

Returns

Nothing.

Enabler

"dreamweaver.canExportTemplateDataAsXML()" on page 420

Example

```
if(dreamweaver.canExportTemplateDataAsXML())
{
  dreamweaver.exportTemplateDataAsXML("file:///c|/dw_temps/mytemplate.txt")
}
```

dreamweaver.getDocumentDOM()

Availability

Dreamweaver 2

Description

Provides access to the tree of objects for the specified document. After the tree of objects returns to the caller, the caller can edit the tree to change the contents of the document.

Arguments

sourceDoc

sourceDoc must be "document", "parent", "parent.frames[*number*]", "parent.frames['*frameName*']", or a URL. The sourceDoc value defaults to "document" if you do not supply a value.

- document specifies the document that has focus and contains the current selection.
- parent specifies the parent frameset (if the currently selected document is in a frame).
- parent.frames[*number*] and parent.frames['*frameName*'] specify a document that is in a particular frame within the frameset that contains the current document.

If the argument is a relative URL, it is relative to the extension file. In Dreamweaver 4, *sourceDoc* defaults to document if omitted.

Note: If the argument is "document", the caller must be applyBehavior(), deleteBehavior(), objectTag(), or any function in a command or Property inspector file in order to perform edits to the document.

Returns

The JavaScript document object at the root of the tree.

Examples

The following example uses dreamweaver.getDocumentDOM() to access the current document:

```
var theDOM = dreamweaver.getDocumentDOM("document");
```

In the following example, the current document DOM identifies a selection and pastes it at the end of another document:

```
var currentDOM = dreamweaver.getDocumentDOM('document');
currentDOM.setSelection(100,200);
currentDOM.clipCopy();
var otherDOM = dreamweaver.openDocument(dreamweaver.¬
getSiteRoot() + "html/foo.htm");
otherDOM.endOfDocument();
otherDOM.clipPaste();
```

Note: openDocument() is used because dom methods normally operate only on open documents. Running a function on a document that isn't open causes a Dreamweaver error. dom methods that can operate only on the active document or on closed documents indicate this fact in their descriptions.

dreamweaver.newDocumentDOM()

Availability

Dreamweaver MX

Description

Provides access to the editable tree for a new, empty document. This works in the same way as `getDocumetDOM()`, except that it points to a new rather than an existing document and does not open the document.

Arguments

None.

Returns

Pointer to new, empty document.

Example

The following code generates a new, empty document:

```
var theDOM = dreamweaver.newDocumentDOM("document");
```

dreamweaver.getRecentFileList()

Availability

Dreamweaver 3

Description

Gets a list of all the files in the recent files list at the bottom of the File menu.

Arguments

None.

Returns

An array of strings that represent the paths of the most recently accessed files. Each path is expressed as a file:// URL. If there are no recent files, the function returns nothing.

dreamweaver.importXMLIntoTemplate()

Availability

Dreamweaver 3

Description

Imports a file of XML text into the current template document. This function operates on the document that has focus, which must be a template. If you do not specify a filename argument, Dreamweaver opens a dialog box to request the import file string.

Arguments

{filePath}

filePath Optional. A string that specifies the filename to which Dreamweaver exports the template. Express *filepath* as a URL file string, such as `"file:///c|/temp/mydata.txt"`.

Returns

Nothing.

dreamweaver.newFromTemplate()

Availability

Dreamweaver 3

Description

Creates a new document from the specified template. If no argument is supplied, the Select Template dialog box appears.

Arguments

{templateURL,} bmaintain

- *templateURL* is the path to a template in the current site, which is expressed as a file: // URL.
- *bmaintain* is a Boolean value, `true` or `false`, that indicates whether to maintain the link to the original template.

Returns

Nothing.

dreamweaver.openDocument()

Availability

Dreamweaver 2

Description

Opens a document for editing in a new Dreamweaver window and gives it the focus. For a user, the effect is the same as choosing File > Open and selecting a file. If the specified file is already open, the window that contains the document comes to the front. The window that contains the specified file becomes the currently selected document. In Dreamweaver 2, if check in/check out is enabled, the file is checked out before it opens. In Dreamweaver 4, you must use "`dreamweaver.openDocumentFromSite()` on page 455 to get this behavior.

Note: This function cannot be called from Behavior action or object files because it causes an error.

Arguments

fileName

fileName is the name of the file to be opened, which is expressed as a URL. If the URL is relative, it is relative to the file that contains the script that called this function.

Returns

The document object for the specified file, which is the same value that `dreamweaver.getDocumentDOM()` returns.

dreamweaver.openDocumentFromSite()

Availability

Dreamweaver 3

Description

Opens a document for editing in a new Dreamweaver window and gives it the focus. For a user, the effect is the same as double-clicking a file in the Site panel. If the specified file is already open, the window that contains the document comes to the front. The window that contains the specified file becomes the currently selected document.

Note: This function cannot be called from Behavior action or object files because it causes an error.

Arguments

fileName

fileName is the filename to open, which is expressed as a URL. If the URL is relative, it is relative to the file that contains the script that called this function.

Returns

The document object for the specified file, which is the same value that `dreamweaver.getDocumentDOM()` returns.

dreamweaver.openInFrame()

Availability

Dreamweaver 3

Description

Opens the Open In Frame dialog box. When the user selects a document, it opens into the active frame.

Arguments

None.

Returns

Nothing.

Enabler

"`dreamweaver.canOpenInFrame()`" on page 420

dreamweaver.releaseDocument()

Availability

Dreamweaver 2

Description

Explicitly releases a previously referenced document from memory.

Documents that are referenced by `dreamweaver.getObjectTags()`, `dreamweaver.getObjectRefs()`, `dreamweaver.getDocumentPath()`, or `dreamweaver.getDocumentDOM()` are automatically released when the script that contains the call finishes executing. If the script opens many documents, you must use this function to explicitly release documents before finishing the script to avoid running out of memory.

Note: This function is relevant only for documents that were referenced by a URL, that are not currently open in a frame or document window, and that are not extension files. Extension files are loaded into memory at startup and are not released until you quit Dreamweaver.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

dreamweaver.revertDocument()

Availability

Dreamweaver 3

Description

Reverts the specified document to the previously saved version.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

Enabler

"`dreamweaver.canRevertDocument()`" on page 422

dreamweaver.saveAll()

Availability

Dreamweaver 3

Description

Saves all open documents, opening the Save As dialog box for any documents that have not previously been saved.

Arguments

None.

Returns

Nothing.

Enabler

"`dreamweaver.canSaveAll()`" on page 422

dreamweaver.saveDocument()

Availability

Dreamweaver 2

Description

Saves the specified file on a local drive.

Note: In Dreamweaver 2, if the file is read-only, Dreamweaver tries to check it out. If the document is still read-only after this attempt, or if it cannot be created, an error message appears.

Arguments

documentObject [, *fileURL*]

- *documentObject* is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.
- *fileURL* is a URL that represents a location on a local drive. If the URL is relative, it is relative to the extension file. In Dreamweaver 2, this argument is required. If *fileURL* is omitted in Dreamweaver 4, the file is saved to its current location if it has been previously saved; otherwise, a Save dialog box appears.

Returns

A Boolean value that indicates success (`true`) or failure (`false`).

Enabler

"dreamweaver.canSaveDocument()" on page 422

dreamweaver.saveDocumentAs()

Availability

Dreamweaver 3

Description

Opens the Save As dialog box.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

dreamweaver.saveDocumentAsTemplate()

Availability

Dreamweaver 3

Description

Opens the Save As Template dialog box.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

Enabler

"dreamweaver.canSaveDocumentAsTemplate()" on page 423

dreamweaver.saveFrameset()

Availability

Dreamweaver 3

Description

Saves the specified frameset, or opens the Save As dialog box if the frameset has not previously been saved.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

Enabler

"dreamweaver.canSaveFrameset()" on page 423

dreamweaver.saveFramesetAs()

Availability

Dreamweaver 3

Description

Opens the Save As dialog box for the frameset file that includes the specified DOM.

Arguments

documentObject

documentObject is the object at the root of a document's DOM tree, which is the value that `dreamweaver.getDocumentDOM()`) returns.

Returns

Nothing.

Enabler

"dreamweaver.canSaveFramesetAs()" on page 423

Find/replace functions

Find/replace functions handle find and replace operations. They cover basic functionality, such as finding the next instance of a search pattern, and complex replacement operations that require no user interaction.

dreamweaver.findNext()

Availability

Dreamweaver 3

Description

Finds the next instance of the search string that was specified previously by "dreamweaver.setUpFind()" on page 462 by "dreamweaver.setUpComplexFind()" on page 461, or by the user in the Find dialog box, and selects the instance in the document.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canFindNext()" on page 420

dreamweaver.replace()

Availability

Dreamweaver 3

Description

Verifies that the current selection matches the search criteria that was specified previously by "dreamweaver.setUpFindReplace()" on page 463, by "dreamweaver.setUpComplexFindReplace()" on page 461, or by the user in the Replace dialog box; and then replaces it with the content that is specified in that query.

Arguments

None.

Returns

Nothing.

dreamweaver.replaceAll()

Availability

Dreamweaver 3

Description

Replaces each section of the current document that matches the search criteria that was specified previously by "dreamweaver.setUpFindReplace()" on page 463, by "dreamweaver.setUpComplexFindReplace()" on page 461, or by the user in the Replace dialog box, with the content that is specified in that query.

Arguments

None.

Returns

Nothing.

dreamweaver.setUpComplexFind()

Availability

Dreamweaver 3

Description

Prepares for an advanced text or tag search by loading the specified XML query.

Arguments

xmlQueryString

xmlQueryString is a string of XML code that begins with `<dwquery>` and ends with `</dwquery>`. (To get a string of the proper format, set up the query in the Find dialog box, click the Save Query button, open the query file in a text editor, and copy everything from the beginning of the `<dwquery>` tag to the end of the `</dwquery>` tag.)

Returns

Nothing.

Example

The first line of the following example sets up a tag search and specifies that the scope of the search should be the current document. The second line performs the search operation.

```
dreamweaver.setUpComplexFind('<dwquery><queryparams matchcase="false" ¬
ignorewhitespace="true" useregexp="false"/><find>¬
<qtag qname="a"><qattribute qname="href" qcompare="="
qvalue="#">¬
  </qattribute><qattribute qname="onMouseOut" qcompare="=" qvalue="" ¬
qnegate="true"></qattribute></qtag></find></dwquery>');
dw.findNext();
```

dreamweaver.setUpComplexFindReplace()

Availability

Dreamweaver 3

Description

Prepares for an advanced text or tag search by loading the specified XML query.

Arguments

xmlQueryString

xmlQueryString is a string of XML code that begins with `<dwquery>` and ends with `</dwquery>`. (To get a string of the proper format, set up the query in the Find dialog box, click the Save Query button, open the query file in a text editor, and copy everything from the beginning of the `<dwquery>` tag to the end of the `</dwquery>` tag.)

Returns

Nothing.

Example

The first statement in the following example sets up a tag search and specifies that the scope of the search should be four files. The second statement performs the search and replace operation.

```
dreamweaver.setUpComplexFindReplace('<dwquery><queryparams ¬
matchcase="false" ignorewhitespace="true" useregexp="false"/>¬
<find><qtag qname="a"><qattribute qname="href" qcompare="=" ¬
  qvalue="#"></qattribute><qattribute qname="onMouseOut" ¬
  qcompare="=" qvalue="" qnegate="true"></qattribute></qtag>¬
  </find><replace action="setAttribute" param1="onMouseOut" ¬
  param2="this.style.color='#000000';this.style.¬
fontWeight='normal'"/></dwquery>');
dw.replaceAll();
```

dreamweaver.setUpFind()

Availability

Dreamweaver 3

Description

Prepares for a text or HTML source search by defining the search parameters for a subsequent `dw.findNext()` operation.

Arguments

`searchObject`

`searchObject` is an object for which the following properties can be defined:

- *`searchString`* is the text for which to search.
- *`searchSource`* is a Boolean value that indicates whether to search the HTML source.
- *`{matchCase}`* is a Boolean value that indicates whether the search is case-sensitive. If this property is not explicitly set, it defaults to `false`.
- *`{ignoreWhitespace}`* is a Boolean value that indicates whether white space differences should be ignored. *`ignoreWhitespace`* defaults to `false` if *`useRegularExpressions`* is `true` and `true` if *`useRegularExpressions`* is `false`.
- *`{useRegularExpressions}`* is a Boolean value that indicates whether the `searchString` uses regular expressions. If this property is not explicitly set, it defaults to `false`.

Returns

Nothing.

Example

The following code demonstrates three ways to create a *`searchObject`* object:

```
var searchParams;
searchParams.searchString = 'bgcolor="#FFCCFF"';
searchParams.searchSource = true;
dreamweaver.setUpFind(searchParams);

var searchParams = {searchString: 'bgcolor="#FFCCFF"', searchSource: true};
dreamweaver.setUpFind(searchParams);

dreamweaver.setUpFind({searchString: 'bgcolor="#FFCCFF"', searchSource: ¬
true});
```

dreamweaver.setUpFindReplace()

Availability

Dreamweaver 3

Description

Prepares for a text or HTML source search by defining the search parameters and the scope for a subsequent `dreamweaver.replace()` or `dw.replaceAll()` operation.

Arguments

`searchObject`

`searchObject` is an object for which the following properties can be defined:

- *`searchString`* is the text for which to search.
- *`replaceString`* is the text with which to replace the selection.
- *`searchSource`* is a Boolean value that indicates whether to search the HTML source.
- *`{matchCase}`* is a Boolean value that indicates whether the search is case-sensitive. If this property is not explicitly set, it defaults to `false`.
- *`{ignoreWhitespace}`* is a Boolean value that indicates whether white space differences should be ignored. *`ignoreWhitespace`* defaults to `false` if *`useRegularExpressions`* is `true` and `true` if *`useRegularExpressions`* is `false`.
- *`{useRegularExpressions}`* is a Boolean value that indicates whether the `searchString` uses regular expressions. If this property is not explicitly set, it defaults to `false`.

Returns

Nothing.

Example

The following code demonstrates three ways to create a *`searchObject`* object:

```
var searchParams;
searchParams.searchString = 'bgcolor="#FFCCFF"';
searchParams.replaceString = 'bgcolor="#CCFFCC"';
searchParams.searchSource = true;
dreamweaver.setUpFindReplace(searchParams);

var searchParams = {searchString: 'bgcolor="#FFCCFF"', replaceString:
  'bgcolor="#CCFFCC"', searchSource: true};
dreamweaver.setUpFindReplace(searchParams);

dreamweaver.setUpFindReplace({searchString: 'bgcolor="#FFCCFF"', ¬
replaceString: 'bgcolor="#CCFFCC"', searchSource: true});
```

dreamweaver.showFindDialog()

Availability

Dreamweaver 3

Description

Opens the Find dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canShowFindDialog()" on page 424

dreamweaver.showFindReplaceDialog()

Availability

Dreamweaver 3

Description

Opens the Replace dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canShowFindDialog()" on page 424

Frame and frameset functions

Frame and frameset functions cover two tasks: getting the names of the frames in a frameset and splitting a frame in two.

dom.getFrameNames()

Availability

Dreamweaver 3

Description

Gets a list of all the named frames in the frameset.

Arguments

None.

Returns

An array of strings where each string is the name of a frame in the current frameset. Any unnamed frames are skipped. If none of the frames in the frameset is named, an empty array returns.

Example

For a document that contains four frames (two of which are named), a call to `dreamweaver.getDocumentDOM().getFrameNames()` might return an array that contains the following strings:

- `"navframe"`
- `"main_content"`

dom.isDocumentInFrame()

Availability

Dreamweaver 4

Description

Identifies whether the current document is being viewed inside a frameset.

Arguments

None.

Returns

Returns `true` if the document is in a frameset; `false`, otherwise.

dom.saveAllFrames()

Availability

Dreamweaver 4

Description

If a document is a frameset or is inside a frameset, `dom.saveAllFrames()` saves all the frames and framesets from the Document window. If the given document is not in a frameset, `dom.saveAllFrames()` saves the document. Opens the Save As dialog box for any documents that have not been previously saved.

Arguments

None

Returns

Nothing.

dom.splitFrame()

Availability

Dreamweaver 3

Description

Splits the selected frame vertically or horizontally.

Arguments

`splitDirection`

`splitDirection` must be one of the following directions: `"up"`, `"down"`, `"left"`, or `"right"`.

Returns

Nothing.

Enabler

“dom.canSplitFrame()” on page 416

General editing functions

You handle general editing functions in the Document window. These functions insert text, HTML, and objects; apply, change, and remove font and character markup; modify tags and attributes; and more.

dom.applyCharacterMarkup()

Availability

Dreamweaver 3

Description

Applies the specified type of character markup to the selection. If the selection is an insertion point, it applies the specified character markup to any subsequently typed text.

Arguments

tagName

tagName is the tag name that is associated with the character markup. It must be one of the following strings: `"b"`, `"cite"`, `"code"`, `"dfn"`, `"em"`, `"i"`, `"kbd"`, `"samp"`, `"s"`, `"strong"`, `"tt"`, `"u"`, or `"var"`.

Returns

Nothing.

dom.applyFontMarkup()

Availability

Dreamweaver 3

Description

Applies the `FONT` tag and the specified attribute and value to the current selection.

Arguments

attribute, *value*

- *attribute* must be `"face"`, `"size"`, or `"color"`.
- *value* is the value that is to be assigned to the attribute; for example, `"Arial, Helvetica, sans-serif"`, `"5"`, or `"#FF0000"`.

Returns

Nothing.

dom.deleteSelection()

Availability

Dreamweaver 3

Description

Deletes the selection in the document.

Arguments

None.

Returns

Nothing.

dom.editAttribute()

Availability

Dreamweaver 3

Description

Displays the appropriate interface for editing the specified attribute. In most cases, this is a dialog box. This function is valid only for the active document.

Arguments

attribute

Returns

Nothing.

dom.exitBlock()

Availability

Dreamweaver 3

Description

Exits the current paragraph or heading block, leaving the cursor outside of all block elements.

Arguments

None.

Returns

Nothing.

dom.getCharSet()

Availability

Dreamweaver 4

Description

Returns the `charset` attribute in the meta tag of the document.

Arguments

None.

Returns

The encoding identity of the document. For example, in *Latin1* document, the function returns `iso-8859-1`.

dom.getFontMarkup()

Availability

Dreamweaver 3

Description

Gets the value of the specified attribute of the `FONT` tag for the current selection.

Arguments

`attribute`

`attribute` must be `"face"`, `"size"`, or `"color"`.

Returns

A string that contains the value of the specified attribute or an empty string if the attribute is not set.

dom.getLinkHref()

Availability

Dreamweaver 3

Description

Gets the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found and then calling `getAttribute('HREF')` on the link.

Arguments

None.

Returns

A string that contains the name of the linked file, which is expressed as a file:// URL.

dom.getLinkTarget()

Availability

Dreamweaver 3

Description

Gets the target of the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found, and then calling `getAttribute('TARGET')` on the link.

Arguments

None.

Returns

A string that contains the value of the `TARGET` attribute for the link or an empty string if no target is specified.

dom.getListTag()

Availability

Dreamweaver 3

Description

Gets the style of the selected list.

Arguments

None.

Returns

A string that contains the tag that is associated with the list (`"ul"`, `"ol"`, or `"dl"`) or an empty string if no tag is associated with the list. This value always returns in lowercase letters.

dom.getTextAlignment()

Availability

Dreamweaver 3

Description

Gets the alignment of the block that contains the selection.

Arguments

None.

Returns

A string that contains the value of the `ALIGN` attribute for the tag that is associated with the block or an empty string if the `ALIGN` attribute is not set for the tag. This value always returns in lowercase letters.

dom.getTextFormat()

Availability

Dreamweaver 3

Description

Gets the block format of the selected text.

Arguments

None.

Returns

A string that contains the block tag that is associated with the text (for example, `"p"`, `"h1"`, `"pre"`, and so on) or an empty string if no block tag is associated with the selection. This value always returns in lowercase letters.

dom.hasCharacterMarkup()

Availability

Dreamweaver 3

Description

Checks whether the selection already has the specified character markup.

Arguments

`markupTagName`

markupTagName is the name of the tag that you're checking. It must be one of the following strings: `"b"`, `"cite"`, `"code"`, `"dfn"`, `"em"`, `"i"`, `"kbd"`, `"samp"`, `"s"`, `"strong"`, `"tt"`, `"u"`, or `"var"`.

Returns

A Boolean value that indicates whether the entire selection has the specified character markup. The function returns `false` if only part of the selection has the specified markup.

dom.indent()

Availability

Dreamweaver 3

Description

Indents the selection using `BLOCKQUOTE` tags. If the selection is a list item, this function indents the selection by converting the selected item into a nested list. This nested list is of the same type as the outer list and contains one item, the original selection.

Arguments

None.

Returns

Nothing.

dom.insertHTML()

Availability

Dreamweaver 3

Description

Inserts HTML content into the document at the current insertion point.

Arguments

contentToInsert, *{bReplaceCurrentSelection}*

- *contentToInsert* is the content you want to insert.
- *bReplaceCurrentSelection* is a Boolean value that indicates whether the content should replace the current selection. If *bReplaceCurrentSelection* is `false`, the content is inserted after the current selection.

Returns

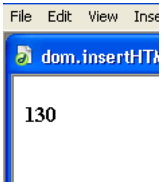
Nothing.

Example

The following code inserts `<b>130</b>` into the current document:

```
var theDOM = dw.getDocumentDOM();
theDOM.insertHTML('<b>130</b>');
```

The result appears in the Document window, as shown in the following figure:



dom.insertObject()

Availability

Dreamweaver 3

Description

Inserts the specified object, prompting the user for parameters if necessary.

Arguments

objectName

objectName is the name of an object in the Configuration/Objects folder.

Returns

Nothing.

Example

A call to `dreamweaver.getDocumentDOM().insertObject('Button')` inserts a form button into the active document after the current selection. If nothing is selected, this function inserts the button at the current insertion point.

Note: Although object files can be stored in separate folders, it's important that their filenames be unique. If a file called Button.htm exists in the Forms folder and also in the MyObjects folder, Dreamweaver cannot distinguish between them.

dom.insertText()

Availability

Dreamweaver 3

Description

Inserts text content into the document at the current insertion point.

Arguments

contentToInsert, *{bReplaceCurrentSelection}*

- *contentToInsert* is the content that you want to insert.
- *bReplaceCurrentSelection* is a Boolean value that indicates whether the content should replace the current selection. If *bReplaceCurrentSelection* is `false`, the content is inserted after the current selection.

Returns

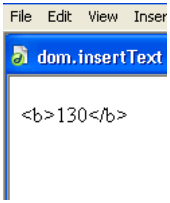
Nothing.

Example

The following code inserts `<b>130</b>` into the current document:

```
var theDOM = dreamweaver.getDocumentDOM();
theDOM.insertText('<b>130</b>');
```

The results appear in the Document window, as shown in the following figure:



dom.newBlock()

Availability

Dreamweaver 3

Description

Creates a new block with the same tag and attributes as the block that contains the current selection or creates a new paragraph if the cursor is outside of all blocks

Arguments

None.

Returns

Nothing.

Example

If the current selection is inside a center-aligned paragraph, a call to `dreamweaver.getDocumentDOM().newBlock()` inserts `<p align="center">` after the current paragraph.

dom.notifyFlashObjectChanged()

Availability

Dreamweaver 4

Description

Informs Dreamweaver that the current Flash object file has changed. Dreamweaver updates the Preview display, resizing it as necessary, and preserving the width-height ratio from the original size. For example, Flash Text uses this feature to update the text in the Layout view as the user changes its properties in the Command dialog box.

Arguments

None.

Returns

Nothing.

dom.outdent()

Availability

Dreamweaver 3

Description

Outdents the selection.

Arguments

None.

Returns

Nothing.

dom.removeCharacterMarkup()

Availability

Dreamweaver 3

Description

Removes the specified type of character markup from the selection.

Arguments

tagName

tagName is the tag name that is associated with the character markup. It must be one of the following strings: "b", "cite", "code", "dfn", "em", "i", "kbd", "samp", "s", "strong", "tt", "u", or "var".

Returns

Nothing.

dom.removeFontMarkup()

Availability

Dreamweaver 3

Description

Removes the specified attribute and its value from a FONT tag. If removing the attribute leaves only <FONT>, the FONT tag is also removed.

Arguments

attribute

attribute must be "face", "size", or "color".

Returns

Nothing.

dom.removeLink()

Availability

Dreamweaver 3

Description

Removes the hypertext link from the selection.

Arguments

None.

Returns

Nothing.

dom.resizeSelection()

Availability

Dreamweaver 3

Description

Resizes the selected object to the specified dimensions.

Arguments

newWidth, *newHeight*

Returns

Nothing.

dom.setAttributeWithErrorChecking()

Availability

Dreamweaver 3

Description

Sets the specified attribute to the specified value for the current selection, prompting the user if the value is the wrong type or if it is out of range. This function is valid only for the active document.

Arguments

attribute, *value*

Returns

Nothing.

dom.setLinkHref()

Availability

Dreamweaver 3

Description

Makes the selection a hypertext link or changes the URL value of the HREF tag that encloses the current selection.

Arguments

linkHREF

linkHREF is the URL (document-relative path, root-relative path, or absolute URL) comprising the link. If this argument is omitted, the Select HTML File dialog box appears.

Returns

Nothing.

Enabler

"dom.canSetLinkHref()" on page 416

dom.setLinkTarget()

Availability

Dreamweaver 3

Description

Sets the target of the link that surrounds the current selection. This function is equivalent to looping through the parents and grandparents of the current node until a link is found and then calling `setAttribute('TARGET')` on the link.

Arguments

{linkTarget}

linkTarget is a string that represents a frame name, window name, or one of the reserved targets (`"_self"`, `"_parent"`, `"_top"`, or `"_blank"`). If the argument is omitted, the Set Target dialog box appears.

Returns

Nothing.

dom.setListBoxKind()

Availability

Dreamweaver 3

Description

Changes the kind of the selected `SELECT` menu.

Arguments

kind

kind must be either `"menu"` or `"list box"`.

Returns

Nothing.

dom.showListPropertiesDialog()

Availability

Dreamweaver 3

Description

Opens the List Properties dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canShowListPropertiesDialog()" on page 416

dom.setListTag()

Availability

Dreamweaver 3

Description

Sets the style of the selected list.

Arguments

listTag

listTag is the tag that is associated with the list. It must be "ol", "ul", "dl", or an empty string.

Returns

Nothing.

dom.setTextAlignment()

Availability

Dreamweaver 3

Description

Sets the ALIGN attribute of the block that contains the selection to the specified value.

Arguments

alignValue

alignValue must be "left", "center", or "right".

Returns

Nothing.

dom.setTextFieldKind()

Availability

Dreamweaver 3

Description

Sets the format of the selected text field.

Arguments

fieldType

fieldType must be "input", "textarea", or "password".

Returns

Nothing.

dom.setTextFormat()

Availability

Dreamweaver 4

Description

Sets the block format of the selected text.

Arguments

blockFormat

blockFormat is a string that specifies one of the following formats: "" (for no format), "p", "h1", "h2", "h3", "h4", "h5", "h6", "pre".

Returns

Nothing.

dom.showFontColorDialog()

Availability

Dreamweaver 3

Description

Opens the Color Picker dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.deleteSelection()

Availability

Dreamweaver 3

Description

Deletes the selection in the active document or the Site panel; on the Macintosh, the text box that has focus in a dialog box or floating panel

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canDeleteSelection()" on page 419

dreamweaver.editFontList()

Availability

Dreamweaver 3

Description

Opens the Edit Font List dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.getFontList()

Availability

Dreamweaver 3

Description

Gets a list of all the font groups that appear in the text Property inspector and in the Style Definition dialog box.

Arguments

None.

Returns

An array of strings that represent each item in the font list.

Example

For the default installation of Dreamweaver, a call to `dreamweaver.getFontList()` returns an array that contains the following items:

- `"Arial, Helvetica, sans-serif"`
- `"Times New Roman, Times, serif"`
- `"Courier New, Courier, mono"`
- `"Georgia, Times New Roman, Times, serif"`
- `"Verdana, Arial, Helvetica, sans-serif"`

dreamweaver.getFontStyles()

Availability

Dreamweaver 4

Description

Returns the styles that a specified TrueType font supports.

Arguments

`fontName`

`fontName` is a string that contains the name of the font.

Returns

An array of three Boolean values that indicates what the font supports. The first value indicates whether the font supports *Bold*, the second indicates whether the font supports *Italic*, and the third indicates whether the font supports both *Bold* and *Italic*.

dreamweaver.getKeyState()

Availability

Dreamweaver 3

Description

Determines whether the specified modifier key is depressed.

Arguments

key

key must be one of the following values: `"Cmd"`, `"Ctrl"`, `"Alt"`, or `"Shift"`. In Windows, `"Cmd"` and `"Ctrl"` refer to the Control key; on the Macintosh, `"Alt"` refers to the Option key.

Returns

A Boolean value that indicates whether the key is depressed.

Example

The following code checks that both the Shift and Control keys (Windows) or Shift and Command keys (Macintosh) are down before performing an operation:

```
if (dw.getKeyState("Shift") && dw.getKeyState("Cmd")){
  // execute code
}
```

dreamweaver.getNaturalSize()

Availability

Dreamweaver 4

Description

Returns the width and height of a graphical object.

Arguments

url

url points to a graphical object for which the dimensions are wanted. Dreamweaver must support this object (GIF, JPEG, PNG, Flash, and Shockwave). The URL that is provided as the argument to `getNaturalSize()` must be an absolute URL that points to a local file; it cannot be a relative URL.

Returns

An array of two integers where the first integer defines the width of the object and the second defines the height.

dreamweaver.getSystemFontList()

Availability

Dreamweaver 4

Description

Returns a list of fonts for the system. This function can get either all fonts or TrueType fonts only. These fonts are needed for the Flash Text object.

Arguments

fontTypes

fontTypes is a string that contains either `"all"` or `"TrueType"`.

Returns

An array of strings that contain all the font names; returns `null` if no fonts are found.

Global application functions

Global application functions act on the entire application. They handle tasks such as quitting and accessing preferences.

dreamweaver.beep()

Availability

Dreamweaver MX

Description

Creates a system beep.

Arguments

None.

Returns

Nothing.

Example

```
beep(){
  if(confirm("Is your order complete?")
  {
    dreamweaver.beep();
    alert("Click OK to submit your order");
  }
}
```

dreamweaver.getShowDialogsOnInsert()

Availability

Dreamweaver 3

Description

Checks whether the Show Dialog When Inserting Objects option is turned on in the General preferences.

Arguments

None.

Returns

A Boolean value that indicates whether the option is on

dreamweaver.quitApplication()

Availability

Dreamweaver 3

Description

Quits Dreamweaver after the script that calls this function finishes executing.

Arguments

None.

Returns

Nothing.

dreamweaver.showAboutBox()

Availability

Dreamweaver 3

Description

Opens the About dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.showDynamicDataDialog()

Availability

Dreamweaver UltraDev 1

Description

Displays the Dynamic Data or Dynamic Text dialog box, and waits for the user to dismiss the dialog box. If the user clicks OK, the `showDynamicDataDialog()` function returns a string to be inserted into the user's document. (This string returned from the Data Sources API function, `generateDynamicDataRef()`, and passed to the Data Format API function, `formatDynamicDataRef()`; the return value from `formatDynamicDataRef()` is the one returned from `showDynamicDataDialog()`.)

Arguments

source, *title*

- *source* is a string that contains source code, which represents the dynamic data object. It is the same string that returned from a previous call to this function. The function uses the contents of *source* to initialize all the dialog box controls, so they appear exactly as when the user clicked OK to create this string.

  Dreamweaver passes this string to `inspectDynamicDataRef()` to determine if the string matches any of the nodes in the tree. If the string matches a node, that node is selected when the dialog box appears. You can also pass an empty string, which does not initialize the dialog box. For example, a dialog box is not initialized when used to create a new item.

- *title* is a string that contains the text to display in the title bar of the dialog box. This argument is optional. If it is not supplied, Dreamweaver displays Dynamic Data in the title bar.

Returns

A string that represents the dynamic data object, if the user clicks OK.

dreamweaver.showPreferencesDialog()

Availability

Dreamweaver 3

Description

Opens the Preferences dialog box.

Arguments

{whichTab}

The agument must be one of the following strings: `"general"`, `"external editors"`, `"floaters"`, `"fonts"`, `"highlighting"`, `"html colors"`, `"html format"`, `"html rewriting"`, `"invisible elements"`, `"layers"`, `"browsers"`, `"quick tag editor"`, `"site ftp"`, `"status bar"`, `"css styles"`, and `"translation"`. If Dreamweaver does not recognize the argument as a valid pane name, or if the argument is omitted, the dialog box opens to the last active pane.

Returns

Nothing.

dreamweaver.showTagChooser()

Availability

Dreamweaver MX

Description

Toggles the visibility of the Tag Chooser dialog box for users to insert tags into the Code view. The function shows the Tag Chooser dialog box on top of all other Dreamweaver windows. If the dialog box is not visible, the function opens the Tag Chooser, bring it to the front, and set focus to it. If the Tag Chooser is visible, the function hides the dialog box.

Arguments

None.

Returns

Nothing.

Global document functions

Global document functions act on an entire document. They check spelling, check target browsers, set page properties, and determine correct object references for elements in the document.

dom.checkSpelling()

Availability

Dreamweaver 3

Description

Checks the spelling in the document, opening the Check Spelling dialog box if necessary, and notifies the user when the check is complete.

Arguments

None.

Returns

Nothing.

dom.checkTargetBrowsers()

Availability

Dreamweaver 3

Description

Runs a target browser check on the document. To run a target browser check on a folder or group of files, see "site.checkTargetBrowsers()" on page 561.

Arguments

None.

Returns

Nothing.

dom.runValidation

Availability

Dreamweaver MX

Description

Runs the Validator on a single, specified document (similar to "site.runValidation()" on page 575).

Arguments

None.

Returns

Nothing.

Enabler

"canAcceptCommand()" on page 62.

dom.showPagePropertiesDialog()

Availability

Dreamweaver 3

Description

Opens the Page Properties dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.doURLDecoding()

Availability

Dreamweaver MX

Description

Uses the internal Dreamweaver URL decoding mechanism to decode special characters and symbols in URL strings. For example, this function decodes `%20` to a space character and the name `&quot` to ".

Arguments

`inStr`

`inStr` is the string to decode.

Returns

A string that contains the decoded URL.

Example

```
outStr = dreamweaver.doURLDecoding("http://maps.yahoo.com/py/
  ddResults.py?Pyt=Tmap&tarname=&tardesc=&newname=&newdesc=&newHash=&newTHash
  =&newSts=&newTSts=&tlt=&tln=&slt=&sln=&newFL=Use+Address+Below&newaddr=2000
  +Shamrock+Rd&newcsz=Metroo+Park%2C+CA&newcountry=us&newTFL=Use+Address+Belo
  w&newtaddr=500+El+Camino&newtcsz=Santa+Clara%2C+CA&newtcountry=us&Submit=Ge
  t+Directions")
```

dreamweaver.getElementRef()

Availability

Dreamweaver 2

Description

Gets the Netscape Navigator or Internet Explorer object reference for a specific tag object in the DOM tree.

Arguments

NSorIE, *tagObject*

- *NSorIE* must be either "NS 4.0" or "IE 4.0". The DOM and rules for nested references differ in Netscape Navigator 4.0 and Internet Explorer 4.0. This argument specifies for which browser to return a valid reference.
- *tagObject* is a tag object in the DOM tree.

Returns

A string that represents a valid JavaScript reference to the object, such as `document.layers['myLayer']`.

- Dreamweaver returns correct references for Internet Explorer for A, AREA, APPLET, EMBED, DIV, SPAN, INPUT, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags.
- Dreamweaver returns correct references for Netscape Navigator for A, AREA, APPLET, EMBED, LAYER, ILAYER, SELECT, OPTION, TEXTAREA, OBJECT, and IMG tags, and for absolutely positioned DIV and SPAN tags. For DIV and SPAN tags that are not absolutely positioned, Dreamweaver returns "cannot reference <*tag*>".
- Dreamweaver does not return references for unnamed objects. If an object does not contain either a NAME or an ID attribute, Dreamweaver returns "unnamed <*tag*>". If the browser does not support a reference by name, Dreamweaver references the object by index (for example, `document.myform.applets[3]`).
- Dreamweaver returns references for named objects that are contained in unnamed forms and layers (for example, `document.forms[2].myCheckbox`).

dreamweaver.getPreferenceInt()

Availability

Dreamweaver MX

Description

Lets you retrieve an integer preference setting for an extension.

Arguments

section key default_value

- *section* a string that specifies the preferences section that contains the entry.
- *key* a string that specifies the entry of the value to be retrieved.
- *default_value* the default value that Dreamweaver returns if it cannot find the entry. Must be an unsigned integer in the range 0 through 65,535 or a signed value in the range -32,768 through 32,767.

Returns

Integer value of the specified entry in the specified section, or the default value if the function does not find the entry. Returns 0 if the value of the specified entry is not an integer.

Example

```
var snapDist = 5; //default value if entry not found
dreamweaver.setPreferenceInt("My Extension", "Snap Distance", snapDist);
```

dreamweaver.getPreferenceString()

Availability

Dreamweaver MX

Description

Lets you retrieve a string preference setting that you stored for an extension.

Arguments

section key default_value

- *section* a string that specifies the preferences section that contains the entry.
- *key* a string that specifies the value to be retrieved.
- *default_value* the default string value that Dreamweaver returns if it cannot find the entry.

Returns

The requested preference string, or if the string cannot be found, the default value.

Example

```
var txtEditor = getExternalTextEditor(); //set default text Editor value
txtEditor = dreamweaver.getPreferenceString("My Extension", "Text Editor",
  txtEditor);
```

dreamweaver.setPreferenceInt()

Availability

Dreamweaver MX

Description

Lets you set an integer preference setting for an extension, to be stored with Dreamweaver preferences when Dreamweaver is not running.

Arguments

section, *key*, *new_value*

- *section* a string that specifies the preferences section that contains the entry. If the section does not exist, Dreamweaver creates it.
- *key* a string that specifies the entry into which the value is to be written. If the entry does not exist, Dreamweaver creates it.
- *new_value* an integer that contains the integer preference value that is to be saved.

Returns

`true` if successful; `false` otherwise.

Example

```
var snapDist = getSnapDistance();
if(snapDist > 0)
{
  dreamweaver.setPreferenceInt("My Extension", "Snap Distance", snapDist);
}
```

dreamweaver.setPreferenceString()

Availability

Dreamweaver MX

Description

Allows you to write a string preference setting for an extension, to be stored with Dreamweaver preferences when Dreamweaver is not running.

Arguments

section, *key*, *new_value*

- *section* a string that specifies the preferences section that contains the entry. If the section does not exist, Dreamweaver creates it.
- *key* a string that specifies the entry into which the value is to be written. If the entry does not exist, Dreamweaver creates it.
- *new_value* a string that contains the preference value that is to be saved.

Returns

`true` if successful; `false` otherwise.

Example

```
var txtEditor = getExternalTextEditor();
dreamweaver.setPreferenceString("My Extension", "Text Editor", txtEditor);
```

History functions

History functions handle undoing, redoing, recording, and playing steps that appear in the History panel. A step is any repeatable change to the document or to a selection in the document. Methods of the `dreamweaver.historyPalette` object either control or act on the selection in the History panel, not in the current document.

dom.redo()

Availability

Dreamweaver 3

Description

Redoes the step that was most recently undone in the document.

Arguments

None.

Returns

Nothing.

Enabler

“dom.canRedo()” on page 415

dom.undo()

Availability

Dreamweaver 3

Description

Undoes the previous step in the document.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canUndo()" on page 417

dreamweaver.getRedoText()

Availability

Dreamweaver 3

Description

Gets the text associated with the editing operation that will be redone if the user selects Edit > Redo or presses Control+Y (Windows) or Command+Y (Macintosh).

Arguments

None.

Returns

A string that contains the text associated with the editing operation that will be redone.

Example

If the user's last action was to make the selection bold, a call to `dw.getRedoText()` returns `"Repeat Apply Bold"`.

dreamweaver.getUndoText()

Availability

Dreamweaver 3

Description

Gets the text associated with the editing operation that will be undone if the user selects Edit > Undo or presses Control+Z (Windows) or Command+Z (Macintosh).

Arguments

None.

Returns

A string that contains the text associated with the editing operation that will be undone.

Example

If the user's last action was to apply a CSS style to a selected range of text, a call to `dw.getUndoText()` returns `"Undo Apply <span>"`.

dreamweaver.playRecordedCommand()

Availability

Dreamweaver 3

Description

Plays the recorded command in the active document.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canPlayRecordedCommand()" on page 421

dreamweaver.redo()

Availability

Dreamweaver 3

Description

Redoes the step that was most recently undone in the active Document window, dialog box, floating panel, or Site panel.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canRedo()" on page 421

dreamweaver.startRecording()

Availability

Dreamweaver 3

Description

Starts recording steps in the active document; the previously recorded command is immediately discarded.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.isRecording()" on page 426 (must return `false`)

dreamweaver.stopRecording()

Availability

Dreamweaver 3

Description

Stops recording without prompting the user.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.isRecording()" on page 426 (must return `true`)

dreamweaver.undo()

Availability

Dreamweaver 3

Description

Undoes the previous step in the active Document window, dialog box, floating panel, or Site panel that has focus.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canUndo()" on page 424

dreamweaver.historyPalette.clearSteps()

Availability

Dreamweaver 3

Description

Clears all steps from the History panel and disables the Undo and Redo menu items.

Arguments

None.

Returns

Nothing.

dreamweaver.historyPalette.copySteps()

Availability

Dreamweaver 3

Description

Copies the specified history steps to the Clipboard. Dreamweaver warns the user of possible unintended consequences if the specified steps include an unrepeatable action.

Arguments

arrayOfIndices

arrayOfIndices is an array of position indices in the History panel.

Returns

A string that contains the JavaScript that corresponds to the specified history steps.

Example

The following code copies the first four steps in the History panel:

```
dreamweaver.historyPalette.copySteps([0,1,2,3]);
```

dreamweaver.historyPalette.getSelectedSteps()

Availability

Dreamweaver 3

Description

Determines which portion of the History panel is selected.

Arguments

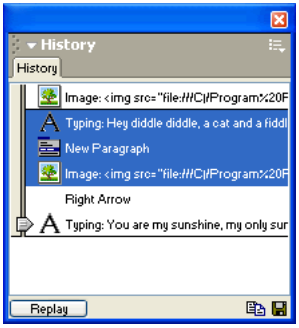
None.

Returns

An array that contains the position indices of all the selected steps. The first position is position 0 (zero).

Example

If the second, third, and fourth steps are selected in the History panel, as shown in the following illustration, a call to `dw.historyPalette.getSelectedSteps()` returns `[1,2,3]`.



dreamweaver.historyPalette.getStepCount()

Availability

Dreamweaver 3

Description

Gets the number of steps in the History panel.

Arguments

None.

Returns

An integer that represents the number of steps that are currently listed in the History panel.

dreamweaver.historyPalette.getStepsAsJavaScript()

Availability

Dreamweaver 3

Description

Gets the JavaScript equivalent of the specified history steps.

Arguments

arrayOfIndices

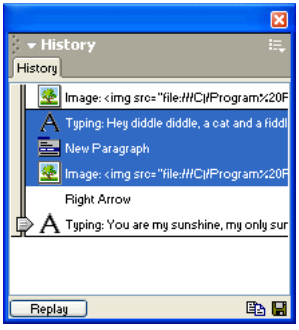
arrayOfIndices is an array of position indices in the History panel.

Returns

A string that contains the JavaScript that corresponds to the specified history steps.

Example

If the three steps shown in the following example are selected in the History panel, a call to `dreamweaver.historyPalette.getStepsAsJavaScript(dw.historyPalette.getSelectedSteps())` returns `"dw.getDocumentDOM().insertText('Hey diddle diddle, a cat and a fiddle, the cow jumped over the moon.');\ndw.getDocumentDOM().newBlock();\ndw.getDocumentDOM().insertHTML('<img src=\"../wdw99/50browsers/images/sun.gif\">', true);\n":`



dreamweaver.historyPalette.getUndoState()

Availability

Dreamweaver 3

Description

Gets the current undo state.

Arguments

None.

Returns

The position of the Undo marker in the History panel.

dreamweaver.historyPalette.replaySteps()

Availability

Dreamweaver 3

Description

Replays the specified history steps in the active document. Dreamweaver warns the user of possible unintended consequences if the specified steps include an unrepeatable action.

Arguments

arrayOfIndices

arrayOfIndices is an array of position indices in the History panel.

Returns

A string that contains the JavaScript that corresponds to the specified history steps.

Example

A call to `dreamweaver.historyPalette.replaySteps([0,2,3])` plays the first, third, and fourth steps in the History panel.

dreamweaver.historyPalette.saveAsCommand()

Availability

Dreamweaver 3

Description

Opens the Save As Command dialog box, which lets the user save the specified steps as a command. Dreamweaver warns the user of possible unintended consequences if the steps include an unrepeatable action.

Arguments

arrayOfIndices

arrayOfIndices is an array of position indexes in the History panel.

Returns

A string that contains the JavaScript that corresponds to the specified history steps.

Example

The following code saves the fourth, sixth, and eighth steps in the History panel as a command:

```
dreamweaver.historyPalette.saveAsCommand([3,5,7]);
```

dreamweaver.historyPalette.setSelectedSteps()

Availability

Dreamweaver 3

Description

Selects the specified steps in the History panel.

Arguments

arrayOfIndices

arrayOfIndices is an array of position indices in the History panel. If no argument is supplied, all the steps are deselected.

Returns

None.

Example

The following code selects the first, second, and third steps in the History panel:

```
dreamweaver.historyPalette.setSelectedSteps([0,1,2]);
```

dreamweaver.historyPalette.setUndoState()

Availability

Dreamweaver 3

Description

Performs the correct number of undo or redo operations to arrive at the specified undo state.

Arguments

undoState

undoState is the object returned by `dreamweaver.historyPalette.getUndoState()`.

Returns

Nothing.

HTML style functions

HTML style functions handle applying, creating, and deleting HTML styles. Methods of the `dreamweaver.htmlStylePalette` object either control or act on the selection in the HTML Styles panel, not in the current document.

dom.applyHTMLStyle()

Availability

Dreamweaver 3

Description

Applies the specified HTML style to the current selection. This function is valid only for the active document.

Arguments

`htmlStyleName`

Returns

Nothing.

dreamweaver.htmlStylePalette.deleteSelectedStyle()

Availability

Dreamweaver 3

Description

Removes the selected style from the HTML Styles panel.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.htmlStylePalette.canEditSelection()" on page 426

dreamweaver.htmlStylePalette.duplicateSelectedStyle()

Availability

Dreamweaver 3

Description

Duplicates the selected style and opens the Define HTML Style dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.htmlStylePalette.canEditSelection()" on page 426

dreamweaver.htmlStylePalette.editSelectedStyle()

Availability

Dreamweaver 3

Description

Opens the Define HTML Style dialog box for the selected style.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.htmlStylePalette.canEditSelection()" on page 426

dreamweaver.htmlStylePalette.getSelectedStyle()

Availability

Dreamweaver 3

Description

Gets the name of the selected style in the HTML Styles panel.

Arguments

None.

Returns

A string that contains the name of the selected style.

dreamweaver.htmlStylePalette.getStyles()

Availability

Dreamweaver 3

Description

Gets a list of all the names of the defined HTML styles.

Arguments

None.

Returns

An array of strings where each string represents the name of an HTML style. If no HTML styles are defined, an empty array returns.

dreamweaver.htmlStylePalette.newStyle()

Availability

Dreamweaver 3

Description

Opens the Define HTML Style dialog box for a new, untitled style.

Arguments

None.

Returns

Nothing.

dreamweaver.htmlStylePalette.setSelectedStyle()

Availability

Dreamweaver 3

Description

Selects the specified style in the HTML Style panel.

Arguments

htmlStyleName

Returns

Nothing.

JavaScript debugger functions

These commands customize the behavior of the Dreamweaver JavaScript Debugger. For more information about the Dreamweaver JavaScript Debugger, see "JavaScript Debugger Modules" on page 243.

dom.getBreakpoint()

Availability

Dreamweaver MX

Description

Queries to see if a breakpoint (the place in the JavaScript code at which the JavaScript Debugger stops executing the program) is set on a particular line in the document.

Arguments

lineNumber

lineNumber is an integer that represents the line number in the document to examine.

Returns

A Boolean value that indicates whether a breakpoint is set (`true`) or not (`false`).

dom.getLineFromOffset()

Availability

Dreamweaver MX

Description

Finds the line number of a specific character offset in the text (the HTML or JavaScript code) of the file.

Arguments

offset

offset is an integer that represents the character location from the beginning of the file.

Returns

An integer that represents the line number in the document.

dom.instrumentDocument ()

Availability

Dreamweaver 4

Description

Creates the debug version of the document and any external .js files that it references. This function parses the JavaScript in the document and calls the *debuggerModule* for code snippets to insert at various points in the JavaScript file. The *debuggerModule* is also notified of syntax errors and warnings. This function fails under any of the following conditions: syntax errors, a file error, or the document cannot be debugged for some reason. Temporary files are never deleted immediately, even if the function fails.

Arguments

debuggerModule, *outputFileName*

- *debuggerModule* is the name of a special Dreamweaver module file that implements the instrumentation API. The module is located in the Configuration/Debugger folder of the Dreamweaver Program Files directory.
- *outputFileName* is optional; it is the name to use for the debug version of the .htm file. If it is omitted, a temporary file is created. The temporary file is deleted when Dreamweaver exits. If the specified *outputFileName* exists, the existing file is replaced. The file is always written in the same directory as the source document, so it cannot have the same name as the source document. If a path is specified, it is ignored. The debug version of externally referenced .js files is named by adding *outputFileName* to the beginning of the original filename of the .js file.

Returns

An array of file URL pairs. Each pair consists of the URL of the original source file, followed by the URL of the debug version that this function created. The first pair is always the .htm file and any subsequent entries are .js files that are referenced by the .htm file. If the function fails, `null` is returned. A pair of URLs is actually two entries in the array. So, if `returnValue = dom.instrumentDocument(test.htm)`, then `returnValue[0]` is the URL of test.htm and `returnValue[1]` is the URL of the debug version of test.htm.

dom.setBreakpoint()

Availability

Dreamweaver MX

Description

Sets or removes a breakpoint (the place in the JavaScript code at which the JavaScript Debugger stops executing the program) on a line in the document.

Arguments

lineNumber, *bTurnOn*

lineNumber is an integer that represents the line number in the document on which to set or remove the breakpoint.

bTurnOn is a Boolean value that indicates whether the breakpoint should be set on (`true`) or off (`false`).

Returns

Nothing.

dreamweaver.debugDocument()

Availability

Dreamweaver 4

Description

Creates the debug version of the current document and opens it in the browser. This function can be used with only one of the browsers for which Dreamweaver supports debugging (see "dreamweaver.getDebugBrowserList()" on page 500). This function does not prompt the user if it cannot determine the browser type. If syntax errors or warnings occur, a Results window opens and displays the messages. If no errors occur, the debug version of the HTML document appears in the specified browser. If warnings occur, but no errors, the Results window appears and debugging begins.

The creation of the debug version is implemented with a call to `dom.instrumentDocument()` using one of the default instrumentation modules. The debug version of the document is temporary and is deleted the next time the JavaScript Debugger starts or when Dreamweaver exits.

Arguments

`fileName`, *`{browserName}`*

- *`fileName`* is the name of the file to be opened. It is expressed as an absolute URL.
- *`browserName`* is optional; it specifies the name of the target browser as defined in the Preview settings in Browser Preferences. It can also be primary or secondary. If omitted, the primary browser is used by default.

Returns

Nothing.

dreamweaver.getDebugBrowserList()

Availability

Dreamweaver 4

Description

Returns the defined browsers for which Dreamweaver supports JavaScript debugging. For Windows, Dreamweaver supports debugging only in Internet Explorer 5.0 and later and Netscape Navigator 4.5 and later. For the Macintosh, Dreamweaver supports debugging only in Netscape Navigator 4.5 and later.

Arguments

None.

Returns

An array of browser names in the same format as the ones `getBrowserList()` returns.

dreamweaver.getIsAnyBreakpoints()

Availability

Dreamweaver 4

Description

Finds any breakpoints that are set in any files.

Arguments

None.

Returns

A Boolean value that returns `true` if any breakpoints are set in any file.

dreamweaver.removeAllBreakpoints()

Availability

Dreamweaver 4

Description

Removes all breakpoints in all files.

Arguments

None.

Returns

Nothing.

dreamweaver.startDebugger()

Availability

Dreamweaver 4

Description

Opens the JavaScript Debugger window with the original source .htm file and the .js files that are listed in `sourceFileList()`. Then it launches the browser with the debug version file specified by `debugFileName()`. This function does not prompt the user if it cannot determine the browser type.

A call to the `dom.instrumentDocument()` function creates the "debugged" version.

Arguments

`sourceFileList`, *`isTempFiles`*, {*`browserName`*}

- *`sourceFileList`* is an array of URL pairs comprising the source file and the instrumented file. Within each pair, the first item is the .htm file and subsequent items are the external .js files. Each .htm file is a URL expressed as an absolute file URL.
- *`isTempFiles`* is a Boolean value that indicates whether the instrumented files should be tracked and deleted the next time the JavaScript Debugger launches or when Dreamweaver exits.
- *`browserName`* is optional; it specifies the name of the target browser as defined in the Preview settings in Browser Preferences. It can also be primary or secondary. If omitted, the primary browser is used by default.

Returns

Nothing.

Keyboard functions

Keyboard functions mimic document navigation tasks that are accomplished by pressing the arrow, Backspace, Delete, Page Up, and Page Down keys. In addition to such general arrow and key methods as `arrowLeft()` and `backspaceKey()`, Dreamweaver also provides methods for moving to the next or previous word or paragraph as well as moving to the start of the line or document, or the end of the line or document.

dom.arrowDown()

Availability

Dreamweaver 3

Description

Moves the insertion point down the specified number of times.

Arguments

`{nTimes}`, *`{bShiftIsDown}`*

- *`nTimes`* is the number of times that the insertion point is to move down. If this argument is omitted, the default is 1.
- *`bShiftIsDown`* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.arrowLeft()

Availability

Dreamweaver 3

Description

Moves the insertion point left the specified number of times.

Arguments

`{nTimes}`, *`{bShiftIsDown}`*

- *`nTimes`* is the number of times that the insertion point is to move left. If this argument is omitted, the default is 1.
- *`bShiftIsDown`* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.arrowRight()

Availability

Dreamweaver 3

Description

Moves the insertion point right the specified number of times.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of times that the insertion point is to move right. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.arrowUp()

Availability

Dreamweaver 3

Description

Moves the insertion point up the specified number of times.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of times that the insertion point is to move up. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.backspaceKey()

Availability

Dreamweaver 3

Description

Equivalent to pressing the Backspace key a specified number of times. The exact behavior depends on whether there is a current selection or only an insertion point.

Arguments

{nTimes}

nTimes is the number of times that a Backspace operation is to occur. If the argument is omitted, the default is 1.

Returns

Nothing.

dom.deleteKey()

Availability

Dreamweaver 3

Description

Equivalent to pressing the Delete key the specified number of times. The exact behavior depends on whether there is a current selection or only an insertion point.

Arguments

{nTimes}

nTimes is the number of times that a Delete operation is to occur. If the argument is omitted, the default is 1.

Returns

Nothing.

dom.endOfDocument()

Availability

Dreamweaver 3

Description

Moves the insertion point to the end of the document (that is, after the last visible content in the Document window, or after the closing `HTML` tag in the Code inspector, depending on which window has focus).

Arguments

{bShiftIsDown}

bShiftIsDown is a Boolean value that indicates whether to extend the selection. If the argument is omitted, the default is `false`.

Returns

Nothing.

dom.endOfLine()

Availability

Dreamweaver 3

Description

Moves the insertion point to the end of the line.

Arguments

{bShiftIsDown}

bShiftIsDown is a Boolean value that indicates whether to extend the selection. If the argument is omitted, the default is `false`.

Returns

Nothing.

dom.nextParagraph()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the next paragraph or skips multiple paragraphs if *nTimes* is greater than 1.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of paragraphs that the insertion point is to move ahead. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.nextWord()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the next word or skips multiple words if *nTimes* is greater than 1.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of words that the insertion point is to move ahead. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.pageDown()

Availability

Dreamweaver 3

Description

Moves the insertion point down one page (equivalent to pressing Page Down).

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move down. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.pageUp()

Availability

Dreamweaver 3

Description

Moves the insertion point up one page (equivalent to pressing Page Up).

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move up. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.previousParagraph()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the previous paragraph or skips multiple paragraphs if *nTimes* is greater than 1.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of paragraphs that the insertion point is to move back. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.previousWord()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the previous word or skips multiple words if *nTimes* is greater than 1.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of words that the insertion point is to move back. If this argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether to extend the selection. If this argument is omitted, the default is `false`.

Returns

Nothing.

dom.startOfDocument()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the document (that is, before the first visible content in the Document window, or before the opening `HTML` tag in the Code inspector, depending on which window has focus).

Arguments

{bShiftIsDown}

bShiftIsDown is a Boolean value that indicates whether to extend the selection. If the argument is omitted, the default is `false`.

Returns

Nothing.

dom.startOfLine()

Availability

Dreamweaver 3

Description

Moves the insertion point to the beginning of the line.

Arguments

{bShiftIsDown}

bShiftIsDown is a Boolean value that indicates whether to extend the selection. If the argument is omitted, the default is `false`.

Returns

Nothing.

dreamweaver.mapKeyCodeToChar()

Availability

Dreamweaver 4

Description

Takes a key code as retrieved from the event object's `keyCode` field and translates it to a character. You should check whether the key code is a special key, such as HOME, PGUP, and so on. If the key code is not a special key, this method can be used to translate it to a character code that is suitable for display to the user.

Arguments

`keyCode`

`keyCode` is the key code to translate to a character.

Returns

Nothing.

Layer and image map functions

Layer and image map functions handle aligning, resizing, and moving layers and image map hotspots. The function description indicates if it applies to layers or to hotspots.

dom.align()

Availability

Dreamweaver 3

Description

Aligns the selected layers or hotspots left, right, top, or bottom.

Arguments

`alignDirection`

`alignDirection` is the edge to align with the layers or hotspots—`"left"`, `"right"`, `"top"`, or `"bottom"`.

Returns

Nothing.

Enabler

"dom.canAlign()" on page 409

dom.arrange()

Availability

Dreamweaver 3

Description

Moves the selected hotspots in the specified direction.

Arguments

toBackOrFront

toBackOrFront is the direction in which the hotspots are to move—that is front or back.

Returns

Nothing.

Enabler

"dom.canArrange()" on page 410

dom.makeSizesEqual()

Availability

Dreamweaver 3

Description

Makes the selected layers or hotspots equal in height, width, or both. The last layer or hotspot selected is the guide.

Arguments

bHoriz, *bVert*

- *bHoriz* is a Boolean value that indicates whether to resize the layers or hotspots horizontally.
- *bVert* is a Boolean value that indicates whether to resize the layers or hotspots vertically.

Returns

Nothing.

dom.moveSelectionBy()

Availability

Dreamweaver 3

Description

Moves the selected layers or hotspots by the specified number of pixels horizontally and vertically.

Arguments

x, *y*

- *x* is the number of pixels that the selection is to move horizontally.
- *y* is the number of pixels that the selection is to move vertically.

Returns

Nothing.

dom.resizeSelectionBy()

Availability

Dreamweaver 3

Description

Resizes the currently selected layer or hotspot.

Arguments

left, *top*, *bottom*, *right*

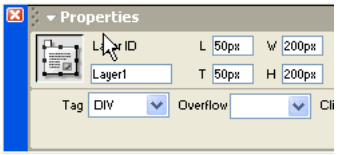
- *left* is the new position of the left boundary of the layer or hotspot.
- *top* is the new position of the top boundary of the layer or hotspot.
- *bottom* is the new position of the bottom boundary of the layer or hotspot.
- *right* is the new position of the right boundary of the layer or hotspot.

Returns

Nothing.

Example

If the selected layer has the Left, Top, Width, and Height properties shown, calling `dw.getDocumentDOM().resizeSelectionBy(-10,-30,30,10)` is equivalent to resetting Left to 40, Top to 20, Width to 240, and Height to 240.



dom.setLayerTag()

Availability

Dreamweaver 3

Description

Specifies the HTML tag that defines the selected layer or layers.

Arguments

tagName

tagName must be `"layer"`, `"ilayer"`, `"div"`, or `"span"`.

Returns

Nothing.

Layout environment functions

Layout environment functions handle operations that are related to the settings for working on a document. They affect the source, position, and opacity of the tracing image; get and set the ruler origin and units; turn the grid on and off and change its settings; and start or stop playing plug-ins.

dom.getRulerOrigin()

Availability

Dreamweaver 3

Description

Gets the origin of the ruler.

Arguments

None.

Returns

An array of two integers. The first array item is the *x* coordinate of the origin, and the second array item is the *y* coordinate of the origin. Both values are in pixels.

dom.getRulerUnits()

Availability

Dreamweaver 3

Description

Gets the current ruler units.

Arguments

None.

Returns

A string that contains one of the following values:

- `"in"`
- `"cm"`
- `"px"`

dom.getTracingImageOpacity()

Availability

Dreamweaver 3

Description

Gets the opacity setting for the document's tracing image.

Arguments

None.

Returns

A value between 0 and 100, or nothing if no opacity is set.

Enabler

"dom.hasTracingImage()" on page 418

dom.loadTracingImage()

Availability

Dreamweaver 3

Description

Opens the Select Image Source dialog box. If the user selects an image and clicks OK, the Page Properties dialog box opens with the Tracing Image field filled in.

Arguments

None.

Returns

Nothing.

dom.playAllPlugins()

Availability

Dreamweaver 3

Description

Plays all plug-in content in the document.

Arguments

None.

Returns

Nothing.

dom.playPlugin()

Availability

Dreamweaver 3

Description

Plays the selected plug-in item.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canPlayPlugin()" on page 415

dom.setRulerOrigin()

Availability

Dreamweaver 3

Description

Sets the origin of the ruler.

Arguments

xCoordinate, *yCoordinate*

- *xCoordinate* is a value, expressed in pixels, on the horizontal axis.
- *yCoordinate* is a value, expressed in pixels, on the vertical axis.

Returns

Nothing.

dom.setRulerUnits()

Availability

Dreamweaver 3

Description

Sets the current ruler units.

Arguments

units

units must be `"px"`, `"in"`, or `"cm"`.

Returns

Nothing.

dom.setTracingImagePosition()

Availability

Dreamweaver 3

Description

Moves the top left corner of the tracing image to the specified coordinates. If the arguments are omitted, the Adjust Tracing Image Position dialog box appears.

Arguments

x, *y*

Returns

Nothing.

Enabler

"dom.hasTracingImage()" on page 418

dom.setTracingImageOpacity()

Availability

Dreamweaver 3

Description

Sets the opacity of the tracing image.

Arguments

opacityPercentage

opacityPercentage must be a number between 0 and 100.

Returns

Nothing.

Enabler

"dom.hasTracingImage()" on page 418

Example

The following code sets the opacity of the tracing image to 30%:

```
dw.getDocumentDOM().setTracingOpacity('30');
```

dom.snapTracingImageToSelection()

Availability

Dreamweaver 3

Description

Aligns the top left corner of the tracing image with the top left corner of the current selection.

Arguments

None.

Returns

Nothing.

Enabler

"dom.hasTracingImage()" on page 418

dom.stopAllPlugins()

Availability

Dreamweaver 3

Description

Stops all plug-in content that is currently playing in the document.

Arguments

None.

Returns

Nothing.

dom.stopPlugin()

Availability

Dreamweaver 3

Description

Stops the selected plug-in item.

Arguments

None.

Returns

A Boolean value that indicates whether the selection is currently being played with a plug-in.

Enabler

"dom.canStopPlugin()" on page 417

dreamweaver.arrangeFloatingPalettes()

Availability

Dreamweaver 3

Description

Moves the visible floating panels to their default positions.

Arguments

None.

Returns

Nothing.

dreamweaver.showGridSettingsDialog()

Availability

Dreamweaver 3

Description

Opens the Grid Settings dialog box.

Arguments

None.

Returns

Nothing.

Layout view functions

Layout view functions handle operations that change the layout elements within a document. They affect table, column, and cell settings, including position, properties, and appearance.

dom.addSpacerToColumn()

Availability

Dreamweaver 4

Description

Creates a 1-pixel-high transparent spacer image at the bottom of a specified column in the currently selected table. This function fails if the current selection is not a table or if the operation is not successful.

Arguments

`colNum`

`colNum` is the column at the bottom of which the spacer image is created.

Returns

Nothing.

dom.createLayoutCell()

Availability

Dreamweaver 4

Description

Creates a layout cell in the current document at the specified position and dimensions, either within an existing layout table or in the area below the existing content on the page. If the cell is created in an existing layout table, it must not overlap or contain any other layout cells or nested layout tables. If the rectangle is not inside an existing layout table, Dreamweaver tries to create a layout table to house the new cell. This function does not force the document into Layout view. This function fails if the cell cannot be created.

Arguments

`left`, `top`, `width`, `height`

- `left` is the *x* position of the left border of the cell.
- `top` is the *y* position of the top border of the cell.
- `width` is the width of the cell in pixels.
- `height` is the height of the cell in pixels.

Returns

Nothing.

dom.createLayoutTable()

Availability

Dreamweaver 4

Description

Creates a layout table in the current document at the specified position and dimensions, either within an existing table or in the area below the existing content on the page. If the table is created in an existing layout table, it cannot overlap other layout cells or nested layout tables, but it can contain other layout cells or nested layout tables. This function does not force the document into Layout view. This function fails if the table cannot be created.

Arguments

left, *top*, *width*, *height*

- *left* is the *x* position of the left border of the table.
- *top* is the *y* position of the top border of the table.
- *width* is the width of the table in pixels.
- *height* is the height of the table in pixels.

Returns

Nothing.

dom.doesColumnHaveSpacer()

Availability

Dreamweaver 4

Description

Determines whether a column contains a spacer image that Dreamweaver generated. This function fails if the current selection is not a table.

Arguments

colNum

colNum is the column to check for a spacer image.

Returns

Returns `true` if the specified column in the currently selected table contains a spacer image that Dreamweaver generated; `false` otherwise.

dom.doesGroupHaveSpacers()

Availability

Dreamweaver 4

Description

Determines whether the currently selected table contains a row of spacer images that Dreamweaver generated. This function fails if the current selection is not a table.

Arguments

None.

Returns

Returns `true` if the table contains a row of spacer images; `false` otherwise.

dom.getClickedHeaderColumn()

Availability

Dreamweaver 4

Description

If the user clicked a menu button in the header of a table in Layout view, causing the table header menu to appear, this function returns the index of the column that the user clicked. The result is undefined if the table header menu is not visible.

Arguments

None.

Returns

An integer that represents the index of the column.

dom.getShowLayoutTableTabs()

Availability

Dreamweaver 4

Description

Determines whether the current document shows tabs for layout tables while in Layout view.

Arguments

None.

Returns

Returns `true` if the current document displays tabs for layout tables while in Layout view; `false` otherwise.

dom.getShowLayoutView()

Availability

Dreamweaver 4

Description

Determines the view for the current document, either Layout view or Standard view.

Arguments

None.

Returns

Returns `true` if the current document is in Layout view; `false` if the document is in Standard view.

dom.isColumnAutostretch()

Availability

Dreamweaver 4

Description

Determines whether a column is set to expand and contract automatically, depending on the document size. This function fails if the current selection is not a table.

Arguments

colNum

colNum is the column to be automatically sized or fixed width.

Returns

Returns `true` if the column at the given index in the currently selected table is set to autostretch; `false` otherwise

dom.makeCellWidthsConsistent()

Availability

Dreamweaver 4

Description

In the currently selected table, sets the width of each column in the HTML to match the currently rendered width of the column. This function fails if the current selection is not a table or if the operation is not successful.

Arguments

None.

Returns

Nothing.

dom.removeAllSpacers()

Availability

Dreamweaver 4

Description

Removes all spacer images generated by Dreamweaver from the currently selected table. This function fails if the current selection is not a table or if the operation is not successful.

Arguments

None.

Returns

Nothing.

dom.removeSpacerFromColumn()

Availability

Dreamweaver 4

Description

Removes the spacer image from a specified column and deletes the spacer row if there are no more spacer images that Dreamweaver generated. This function fails if the current selection is not a table or if the operation is not successful.

Arguments

colNum

colNum is the column from which to remove the spacer image.

Returns

Nothing.

dom.setColumnAutostretch()

Availability

Dreamweaver 4

Description

Switches a column between automatically sized or fixed width. If *bAutostretch* is `true`, the column at the given index in the currently selected table is set to autostretch; otherwise it's set to a fixed width at its current rendered width. This function fails if the current selection isn't a table or if the operation is not successful.

Arguments

colNum, *bAutostretch*

- *colNum* is the column to be automatically sized or set to a fixed width.
- *bAutostretch* specifies whether to set the column to autostretch (`true`) or to a fixed width (`false`).

Returns

Nothing.

dom.setShowLayoutTableTabs()

Availability

Dreamweaver 4

Description

Sets the current document to display tabs for layout tables whenever it's in Layout view. This function does not force the document into Layout view.

Arguments

bShow

bShow indicates whether to display tabs for layout tables when the current document is in Layout view. If *bShow* is `true`, shows tabs; `false` otherwise.

Returns

Nothing.

dom.setShowLayoutView()

Availability

Dreamweaver 4

Description

Places the current document in Layout view if *bShow* is `true`.

Arguments

bShow

bShow is a Boolean value that toggles the current document between Layout view and Standard view. If *bShow* is `true`, the current document switches to Layout view. If *bShow* is `false`, the current document switches to Standard view.

Returns

Nothing.

Library and template functions

Library and template functions handle operations that are related to library items and templates, such as creating, updating, and breaking links between a document and a template or library item. Methods of the `dreamweaver.libraryPalette` object either control or act on the selection in the Library panel, not in the current document. Likewise, methods of the `dreamweaver.templatePalette` object control or act on the selection in the Templates panel.

dom.applyTemplate()

Availability

Dreamweaver 3

Description

Applies a template to the current document. If no argument is supplied, the Select Template dialog box appears. This function is valid only for the document that has focus.

Arguments

{templateURL}, *bMaintainLink*

- *templateURL* is the path to a template in the current site, which is expressed as a file:// URL.
- *bMaintainLink* is a Boolean value that indicates whether to maintain the link to the original template (`true`) or not (`false`).

Returns

Nothing.

Enabler

"dom.canApplyTemplate()" on page 409

dom.detachFromLibrary()

Availability

Dreamweaver 3

Description

Detaches the selected library item instance from its associated LBI file by removing the locking tags from around the selection. This function is equivalent to clicking Detach from Original in the Property inspector.

Arguments

None.

Returns

Nothing.

dom.detachFromTemplate()

Availability

Dreamweaver 3

Description

Detaches the current document from its associated template.

Arguments

None.

Returns

Nothing.

dom.getAttachedTemplate()

Availability

Dreamweaver 3

Description

Gets the path of the template that is associated with the document.

Arguments

None.

Returns

A string that contains the path of the template, which is expressed as a file:// URL.

dom.getEditableRegionList()

Availability

Dreamweaver 3

Description

Gets a list of all the editable regions in the body of the document.

Arguments

None.

Returns

An array of element nodes.

Example

See “dom.getSelectedEditableRegion()” on page 523

dom.getIsLibraryDocument()

Availability

Dreamweaver 3

Description

Determines whether the document is a library item

Arguments

None.

Returns

A Boolean value that indicates whether the document is an LBI file.

dom.getIsTemplateDocument()

Availability

Dreamweaver 3

Description

Determines whether the document is a template.

Arguments

None.

Returns

A Boolean value that indicates whether the document is a DWT file.

dom.getSelectedEditableRegion()

Availability

Dreamweaver 3

Description

If the selection or insertion point is inside an editable region, gets the position of the editable region among all others in the body of the document.

Arguments

None.

Returns

An index into the array that "dom.getEditableRegionList()" on page 522 returns.

Example

The following code shows a dialog box with the contents of the selected editable region:

```
var theDOM = dw.getDocumentDOM();
var edRegs = theDOM.getEditableRegionList();
var selReg = theDOM.getSelectedEditableRegion();
alert(edRegs[selReg].innerHTML);
```

dom.insertLibraryItem()

Availability

Dreamweaver 3

Description

Inserts an instance of a library item into the document.

Arguments

libraryItemURL

libraryItemURL is the path to an LBI file, which is expressed as a file:// URL.

Returns

Nothing.

dom.markSelectionAsEditable()

Availability

Dreamweaver 3

Description

Displays the New Editable Region dialog box. When the user clicks New Region, Dreamweaver marks the selection as editable and doesn't change any text.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canMarkSelectionAsEditable()" on page 414

dom.newEditableRegion()

Availability

Dreamweaver 3

Description

Displays the New Editable Region dialog box. When the user clicks New Region, Dreamweaver inserts the name of the region, surrounded by braces ({ }), into the document at the insertion point location.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canMakeNewEditableRegion()" on page 414

dom.removeEditableRegion()

Availability

Dreamweaver 3

Description

Removes an editable region from the document. If the editable region contains any content, the content is preserved; only the editable region markers are removed.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canRemoveEditableRegion()" on page 415

dom.updateCurrentPage()

Availability

Dreamweaver 3

Description

Updates the document's library items, templates, or both. This function is valid only for the active document.

Arguments

{typeOfUpdate}

typeOfUpdate, if supplied, must be "library", "template", or "both". If omitted, the default is "both".

Returns

Nothing.

dreamweaver.exportTemplateDataAsXML()

Availability

Dreamweaver MX

Description

Exports the current document to the specified file as XML. This function operates on the front document, which must be a template. If you do not specify a filename argument, Dreamweaver opens a dialog box to request the export file string.

Arguments

{filePath}

filePath Optional. A string that specifies the name of the file to which Dreamweaver exports the template. Express *filepath* as a URL file string such as, "file:///c|/temp/mydata.txt".

Returns

Nothing.

Enabler

"dreamweaver.canExportTemplateDataAsXML()" on page 420

Example

```
if(dreamweaver.canExportTemplateDataAsXML())
{
  dreamweaver.exportTemplateDataAsXML("file:///c|/dw_temps/mytemplate.txt")
}
```

dreamweaver.updatePages()

Availability

Dreamweaver 3

Description

Opens the Update Pages dialog box and selects the specified options.

Arguments

typeOfUpdate

typeOfUpdate must be `"library"`, `"template"`, or `"both"`. If the argument is omitted, it defaults to `"both"`.

Returns

Nothing.

Live data functions

You can use the following live-data functions to mimic menu functionality:

- `showLiveDataDialog()` is used for the View > Live Data Settings menu item.
- `setLiveDataMode()` is used for the View > Live Data and View > Refresh Live Data menu items.
- `getLiveDataMode()` is also used for View > Live Data menu item.

You can use the remaining live-data functions when you implement the translator API `liveDataTranslateMarkup()` function.

dreamweaver.getLiveDataInitTags()

Availability

Dreamweaver UltraDev 1

Description

Returns the initialization tags for the currently active document. The initialization tags are the HTML tags that the user supplies in the Live Data Settings dialog box. This function is typically called from a translator's `liveDataTranslateMarkup()` function, so that the translator can pass the tags to the `liveDataTranslate()` function.

Arguments

None.

Returns

A string that contains the initialization tags.

dreamweaver.getLiveDataMode()

Availability

Dreamweaver UltraDev 1

Description

Determines whether the Live Data window is currently visible.

Arguments

None.

Returns

A Boolean value that indicates whether the Live Data window is visible.

dreamweaver.getLiveDataParameters ()

Availability

Dreamweaver MX

Description

Obtains the URL parameters that are specified as Live Data settings.

Live Data mode lets you view web page in the design stage (as if it has been translated by the application server and returned). Generating dynamic content to display in Design view lets you view your page layout with live data and adjust it, if necessary.

Before you view live data, you must enter Live Data settings for any URL parameters that you reference in your document. This prevents the web server from returning errors for parameters that are otherwise undefined in the simulation.

You enter the URL parameters in name-value pairs. For example, if you reference the URL variables `ID` and `Name` in server scripts in your document, you must set these URL parameters before you view live data.

You can enter Live Data settings through Dreamweaver MX in the following two ways:

- Through the Live Data Settings dialog box, which you can activate from the View menu
- In the URL text field that appears at the top of the document when you click the Live Data View button on the toolbar

For the `ID` and `Name` parameters, you can enter the following pairs:

```
ID     22
Name   Samuel
```

In the URL, these parameters would appear as shown in the following example:

```
http://someURL?ID=22&Name=Samuel
```

This function lets you obtain these live-data settings through JavaScript.

Arguments

None.

Returns

An array that contains the URL parameters for the current document. The array contains an even number of parameter strings. Each two elements form a URL parameter name-value pair. The even element is the parameter name and the odd element is the value. For example, `getLiveDataParameters()` returns the following array for the `ID` and `Name` parameters in the preceding example: `['ID,'22','Name','Samuel']`.

Example

```
var paramsArray = dreamweaver.getLiveDataParameters();
```

dreamweaver.liveDataTranslate()

Availability

Dreamweaver UltraDev 1

Description

Sends an entire HTML document to an application server, asks the server to execute the scripts in the document, and returns the resulting HTML document. This function can be called only from a translator's `liveDataTranslateMarkup()` function; if you try to call it at another time, an error occurs. The `dreamweaver.liveDataTranslate()` function performs the following operations:

- Makes the animated image (that appears near the right edge of the Live Data window) play.
- Listens for user input. If the Stop icon is clicked, the function returns immediately.
- Accepts a single string argument from the caller. (This string is typically the entire source code of the user's document. It is the same string that is used in the next operation.)
- Saves the HTML string from the user's document as a temporary file on the live-data server.
- Sends an HTTP request to the live-data server, using the parameters specified in the Live Data Settings dialog box.
- Receives the HTML response from the live-data server.
- Removes the temporary file from the live-data server.
- Makes the animated image stop playing.
- Returns the HTML response to the caller.

Arguments

A single string, which typically is the entire source code of the user's current document.

Returns

An `httpReply` object. This object is the same as the value that the `MMHttp.getText()` function returns. If the user clicks the Stop icon, the return value's `httpReply.statusCode` is equal to 200 (Status OK) and its `httpReply.data` is equal to the empty string. See "The HTTP API" on page 281 for more information on the `httpReply` object.

dreamweaver.setLiveDataError()

Availability

Dreamweaver UltraDev 1

Description

Specifies the error message to display if an error occurs while the `liveDataTranslateMarkup()` function executes in a translator. If the document that Dreamweaver passed to `liveDataTranslate()` contains errors, the server passes back an error message that is formatted using HTML. If the translator (the code that called `liveDataTranslate()`) determines that the server returned an error message, it calls `setLiveDataError()` to display the error message in Dreamweaver. This message is shown after the `liveDataTranslateMarkup()` function finishes executing; Dreamweaver displays the description in an error dialog box. The `setLiveDataError()` function should be called only from the `liveDataTranslateMarkup()` function.

Arguments

`source`

`source` is a string that contains source code, which is parsed and rendered in the error dialog box.

Returns

Nothing.

dreamweaver.setLiveDataMode()

Availability

Dreamweaver UltraDev 1

Description

Toggles the visibility of the Live Data window.

Arguments

`bIsVisible`

`bIsVisible` is a Boolean value that indicates whether the Live Data window should be visible. If you pass `true` to this function and Dreamweaver currently displays the Live Data window, the effect is the same as if you clicked Refresh.

Returns

Nothing.

dreamweaver.setLiveDataParameters ()

Availability

Dreamweaver MX

Description

Sets the URL parameters that you reference in your document for use in Live Data mode.

Live Data mode lets you view web page in the design stage (as if it has been translated by the application server and returned). Generating dynamic content to display in Design view lets you view your page layout with live data and adjust it, if necessary.

Before you view Live Data, though, you must enter Live Data settings for any URL parameters that you reference in your document. This prevents the web server from returning errors for parameters that are otherwise undefined in the simulation.

You enter the URL parameters in name-value pairs. For example, if you reference the URL variables `ID` and `Name` in server scripts in your document, you must set these URL parameters before you view live data.

This function lets you set Live Data values through JavaScript.

Arguments

A string that contains the URL parameters that you want to set, in name-value pairs.

Returns

Nothing.

Example

```
dreamweaver.setLiveDataParameters("ID=22&Name=Samuel")
```

dreamweaver.showLiveDataDialog()

Availability

Dreamweaver UltraDev 1

Description

Displays the Live Data Settings dialog box.

Arguments

None.

Returns

Nothing.

Menu functions

Menu functions handle optimizing and reloading the menus in Macromedia Dreamweaver MX. The "`dreamweaver.getMenuNeedsUpdating()` on page 530 and "`dreamweaver.notifyMenuUpdated()` on page 531 functions are designed specifically to prevent unnecessary update routines from running on the dynamic menus that are built into Dreamweaver.

dreamweaver.getMenuNeedsUpdating()

Availability

Dreamweaver 3

Description

Checks whether the specified menu needs to be updated.

Arguments

`menuId`

`menuId` is a string that contains the value of the `id` attribute for the menu item, as specified in the menus.xml file.

Returns

A Boolean value that indicates whether the menu needs to be updated. This function returns `false` only if "`dreamweaver.notifyMenuUpdated()` on page 531 has been called with this *`menuId`*, and the return value of *`menuListFunction`* has not changed since then. For more information, see "`dreamweaver.notifyMenuUpdated()` on page 531.

dreamweaver.notifyMenuUpdated()

Availability

Dreamweaver 3

Description

Notifies Dreamweaver when the specified menu needs to be updated.

Arguments

`menuId`, *`menuListFunction`*

- *`menuId`* is a string that contains the value of the `id` attribute for the menu item, as specified in the menus.xml file.
- *`menuListFunction`* must be one of the following strings: `"dw.cssStylePalette.getStyles()"`, `"dw.getDocumentDOM().getFrameNames()"`, `"dw.getDocumentDOM().getEditableRegionList"`, `"dw.getBrowserList()"`, `"dw.getRecentFileList()"`, `"dw.getTranslatorList()"`, `"dw.getFontList()"`, `"dw.getDocumentList()"`, `"dw.htmlStylePalette.getStyles()"`, or `"site.getSites()"`.

Returns

Nothing.

dreamweaver.reloadMenus()

Availability

Dreamweaver 3

Description

Reloads the entire menu structure from the menus.xml file in the Configuration folder.

Arguments

None.

Returns

Nothing.

Path functions

Path functions get and manipulate the paths to various files and folders on a user's hard disk. These functions determine the path to the root of the site in which the current document resides, convert relative paths to absolute URLs, and more.

dreamweaver.getConfigurationPath()

Availability

Dreamweaver 2

Description

Gets the path to the Dreamweaver Configuration folder, which is expressed as a file:// URL.

See "File Access and Multiuser Configuration API" on page 260 for information on how Dreamweaver accesses Configuration folders on a multiuser platform.

Arguments

None.

Returns

Returns the path to the application configurations.

Example

This function is useful when referencing other extension files, which are stored in the Configuration folder inside the Dreamweaver application folder.

```
var sortCmd = dreamweaver.getConfigurationPath() + ¬
"/Commands/Sort Table.htm"
var sortDOM = dreamweaver.getDocumentDOM(sortCmd);
```

dreamweaver.getDocumentPath()

Availability

Dreamweaver1.2

Description

Gets the path of the specified document, which is expressed as a file:// URL. This function is equivalent to calling `dreamweaver.getDocumentDOM()` and reading the `URL` property of the return value.

Arguments

`sourceDoc`

`sourceDoc` must be `"document"`, `"parent"`, `"parent.frames[`*`number`*`]"`, or `"parent.frames['`*`frameName`*`']"`. `document` specifies the document that has the focus and contains the current selection. `"parent"` specifies the parent frameset (if the currently selected document is in a frame), and `"parent.frames[`*`number`*`]"` and `"parent.frames['`*`frameName`*`']"` specify a document that is in a particular frame within the frameset that contains the current document.

Returns

Either a string that contains the URL of the specified document if the file was saved or an empty string if the file was not saved

dreamweaver.getSiteRoot()

Availability

Dreamweaver 1.2

Description

Gets the local root folder (as specified in the Site Definition dialog box) for the site that is associated with the currently selected document, which is expressed as a file:// URL.

Arguments

None.

Returns

Either a string that contains the URL of the local root folder of the site within which the file was saved or an empty string if the file is not associated with a site.

dreamweaver.relativeToAbsoluteURL()

Availability

Dreamweaver 2

Description

Given a relative URL and a point of reference (either the path to the current document or the site root), this function converts the relative URL to an absolute (file:///) URL.

Arguments

docPath, *siteRoot*, *relURL*

- *docPath* is the path to a document on the user's disk (for example, the current document), which is expressed as a file:// URL or an empty string if *relURL* is a root-relative URL.
- *siteRoot* is the path to the site root, which is expressed as a file:// URL or an empty string if *relURL* is a document-relative URL.
- *relURL* is the URL to be converted.

Returns

An absolute URL string. The return value is generated as described in the following list:

- If *relURL* is an absolute URL, no conversion takes place, and the return value is the same as *relURL*.
- If *relURL* is a document-relative URL, the return value is the combination of *docPath* + *relURL*.
- If *relURL* is a root-relative URL, the return value is the combination of *siteRoot* + *relURL*.

Print function

The print function lets the user print code from Code view.

dreamweaver.PrintCode()

Availability

Dreamweaver MX

Description

In Windows, prints all or selected portions of code from the Code view. On the Macintosh, prints all code or a page range of code.

Arguments

showPrintDialog, document

- *showPrintDialog* is `true` or `false`. If this argument is set to `true`, in Windows `dreamweaver.PrintCode()` displays the print dialog box to ask if the user wants to print all text or selected text. On the Macintosh, the `dreamweaver.PrintCode()` function displays the print dialog box to ask if the user wants to print all text or a page range. If the argument is set to `false`, `dreamweaver.PrintCode()` uses the user's previous selection. The default value is `true`.
- *document* is the document DOM of the document to print. For information on how to obtain the DOM for a document, see "dreamweaver.getDocumentDOM()" on page 453.

Returns

`true` if able to print the code.

`false` if unable to print the code.

Example

```
var theDOM = dreamweaver.getDocumentDOM("document");
if(!dreamweaver.PrintCode(true, theDOM))
{
  alert("Unable to execute your print request!");
}
```

Quick Tag Editor Functions

Quick tag editor functions navigate through the tags within and surrounding the current selection. They remove any tag in the hierarchy, wrap the selection inside a new tag, and show the Quick tag editor to let the user edit specific attributes for the tag.

dom.selectChild()

Availability

Dreamweaver 3

Description

Selects a child of the current selection. Calling this function is equivalent to selecting the next tag to the right in the tag selector at the bottom of the Document window.

Arguments

None.

Returns

Nothing.

dom.selectParent()

Availability

Dreamweaver 3

Description

Selects the parent of the current selection. Calling this function is equivalent to selecting the next tag to the left in the tag selector at the bottom of the Document window.

Arguments

None.

Returns

Nothing.

dom.stripTag()

Availability

Dreamweaver 3

Description

Removes the tag from around the current selection, leaving the contents, if any. If the selection contains no tags or more than one tag, Dreamweaver reports an error.

Arguments

None.

Returns

Nothing.

dom.wrapTag()

Availability

Dreamweaver 3

Description

Wraps the specified tag around the current selection. If the selection is unbalanced, Dreamweaver reports an error.

Arguments

startTag

startTag is the source that is associated with the opening tag.

Returns

Nothing.

Example

The following code wraps a link around the current selection:

```
var theDOM = dw.getDocumentDOM();
var theSel = theDOM.getSelectedNode();
if (theSel.nodeType == Node.TEXT_NODE){
  theDOM.wrapTag('<a href="foo.html">');
}
```

dreamweaver.showQuickTagEditor()

Availability

Dreamweaver 3

Description

Displays the Quick tag editor for the current selection.

Arguments

{nearWhat}, *{mode}*

- *nearWhat*, if specified, must be either `"selection"` or `"tag selector"`. The default value, if this argument is omitted, is `"selection"`.
- *mode*, if specified, must be `"default"`, `"wrap"`, `"insert"`, or `"edit"`. If *mode* is `"default"` or omitted, Dreamweaver uses heuristics to determine the mode to use for the current selection. *mode* is ignored if *nearWhat* is `"tag selector"`.

Returns

Nothing.

Report Functions

Report functions provide access to the Dreamweaver reporting features so you can initiate, monitor and customize the reporting process. For more information, see "Reports" on page 103.

dreamweaver.isReporting()

Availability

Dreamweaver 4

Description

Checks to see if a reporting process is currently running.

Arguments

None.

Returns

A Boolean value that indicates whether a process is running (`true`) or not (`false`).

dreamweaver.showReportsDialog()

Availability

Dreamweaver 4

Description

Opens the Reports dialog box.

Arguments

None.

Returns

Nothing.

Results window functions

Results window functions let you create a stand-alone window that displays columns of formatted data, or you can interact with the built-in windows of the Results panel group.

Creating a Stand-alone Results window

These functions create custom windows that are similar to the output from the JavaScript Debugger window.

dreamweaver.createResultsWindow()

Availability

Dreamweaver 4

Description

Creates a new Results window and returns a JavaScript object reference to the window.

Arguments

strName, *arrColumns*

- *strName* is the string to use for the window's title.
- *arrColumns* is an array of column names to use in the list control.

Returns

An object reference to the created window.

resWin.addItem()

Availability

Dreamweaver 4

Description

Adds a new item to the Results window.

Arguments

strIcon, *strDesc*, *iStartSel*, *iEndSel*, *colNdata*

- *strIcon* is the path to the icon to use. To display a built-in icon, use a value 1 through 10 instead of the fully qualified path name of the icon (use 0 for no icon). The following table details which icon will appear, given the corresponding value:

- *strDesc* is a detailed description of a code font item. Specify code font if there is no description.
- *iStartSel* is the start of selection offset in the file; specify `null` if not used.
- *iEndSel* is the end of selection offset in the file; specify code font if not used.
- *colNdata* is a string to use for each column.

Returns

A Boolean value; `true` if the item was added successfully, `false` otherwise.

resWin.addResultItem()

Availability

Dreamweaver 4

Description

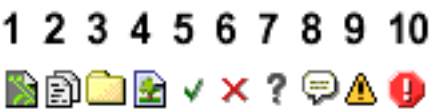
Adds a new results entry to the current Results window based on the information in the file processed by the `processfile()` function. The current Results window corresponds to the Results window that is active during the reporting process. If a report is not being generated, then this function has no effect.

This function adds new results until all files pertaining to the user's selection have been processed, or the user clicks the Stop button at the bottom of the window. Dreamweaver displays the name of each file being processed and the number of remaining files to be processed. Dreamweaver automatically releases each file's DOM when finished.

Arguments

strFilePath, strIcon, strDisplay, strDesc, iLineNo, iStartSel, iEndSel

- *strFilePath* is a fully qualified URL file path name of the file to process.
- *strIcon* is the path to the icon to use. To display a built-in icon, use a value 1 through 10 instead of the fully qualified path name of the icon (use 0 for no icon). The following table details which icon will appear, given the corresponding value:



- *strDisplay* is the string to display to the user in first column of the results window (usually, the filename).

- *strDesc* is the description to go along with the entry.
- *iLineNo* is the number of lines in file (optional).
- *iStartSel* is the start of offset into file (optional, but if used, the *iEndSel* argument must also be used.).
- *iEndSel* is the end of offset into file (required if *iStartSel* is used).

Returns

Nothing

resWin.setCallbackCommands()

Availability

Dreamweaver 4

Description

Tells the Results window on which commands to call the `processFile()` method. If this function is not called, the command that created the Results window is called.

Arguments

arrCmdNames

arrCmdNames is an array of command names on which to call the `processFile()` method.

Returns

Nothing.

resWin.setColumnWidths()

Availability

Dreamweaver 4

Description

Sets the width of each column.

Arguments

arrWidth

arrWidth is an array of integers that represents the widths to use for each column in the control.

Returns

Nothing.

resWin.setFileList()

Availability

Dreamweaver 4

Description

Gives the Results window a list of files, directories, or both to call a set of commands to process.

Arguments

arrFilePaths, *bRecursive*

- *arrFilePaths* is an array of file or folder paths to iterate through.
- *bRecursive* is a Boolean value that indicates whether the iteration should be recursive (`true`) or not (`false`).

Returns

Nothing.

resWin.setTitle()

Availability

Dreamweaver 4

Description

Sets the title of the window.

Arguments

strTitle

strTitle is the new name of the floating panel.

Returns

Nothing.

resWin.startProcessing()

Availability

Dreamweaver 4

Description

Starts processing the file.

Arguments

None.

Returns

Nothing.

resWin.stopProcessing()

Availability

Dreamweaver 4

Description

Stops processing the file.

Arguments

None.

Returns

Nothing.

Working with the built-in Results panel group

These functions produce output in the Results panel group. The Results panel group displays tabbed reports on searches, source validation, sitewide reports, browser targets, console reports, FTP logging, and link-checking.

Working with specific child panels

The following child panels are build-in Results windows that always exist in the Dreamweaver interface, and can be accessed directly. Since these windows are Results windows, you can use the same methods defined for stand-alone Results windows. For more information about using the `resWin` methods, see "Creating a Stand-alone Results window" on page 537.

- `dreamweaver.resultsPalette.siteReports`
- `dreamweaver.resultsPalette.validator`
- `dreamweaver.resultsPalette.btc` (Target Browser Check panel)

Working with the active child panel

The following general API functions apply to whichever child panel is active. Some child panels may ignore some of these methods. If the active child panel does not support the method, then calling the method has no effect.

dreamweaver.resultsPalette.clearItems()

Availability

Dreamweaver MX

Description

Clears the contents of the panel in focus.

Arguments

None.

Returns

Nothing.

Enabler

"`dreamweaver.resultsPalette.canClearItems()`" on page 426

dreamweaver.resultsPalette.clipCopy()

Availability

Dreamweaver MX

Description

Sends a copied message to the current window in focus (often used for the FTP logging window).

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canClipCopy()" on page 427

dreamweaver.resultsPalette.clipCut()

Availability

Dreamweaver MX

Description

Sends a cut message to the window in focus (often used for the FTP logging window).

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canClipCut()" on page 427

dreamweaver.resultsPalette.clipPaste()

Availability

Dreamweaver MX

Description

Sends a pasted message to the window in focus (often used for the FTP logging window).

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canClipPaste()" on page 427

dreamweaver.resultsPalette.openInBrowser

Availability

Dreamweaver MX

Description

Sends a report (Site Reports, Browser Target Check, Validation, and Link Checker) to the default browser.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canOpenInBrowser()" on page 427

dreamweaver.resultsPalette.openInEditor()

Availability

Dreamweaver MX

Description

Jumps to the selected line for specific reports (Site Reports, Browser Target Check, Validation, and Link Checker) and opens the document in the editor.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canOpenInEditor()" on page 428

dreamweaver.resultsPalette.save()

Availability

Dreamweaver MX

Description

Launches the Save dialog box for a window that supports the Save function (Site Reports, Browser Target Check, Validation, and Link Checker).

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canSave()" on page 428

dreamweaver.resultsPalette.selectAll()

Availability

Dreamweaver MX

Description

Sends a Select All command to the window in focus.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.resultsPalette.canSelectAll()" on page 428

Server debugging

Dreamweaver can request files from ColdFusion and display the response in its embedded browser. When the response returns from the server, Dreamweaver searches the response for a packet of XML that has a known signature. If Dreamweaver finds XML with that signature, it processes the XML and displays the contained information in a tree control. This tree displays information about the following items:

- All templates, custom tags, and include files that are used to generate the rendered .cfm page
- Exceptions
- SQL queries
- Object queries
- Variables
- Trace trail

Additionally, the Server Debug panel can display debug data from other server models. To set up Dreamweaver to debug other server models, use the `dreamweaver.resultsPalette.debugWindow.addDebugContextData()` method.

dreamweaver.resultsPalette.debugWindow.addDebugContextData()

Availability

Dreamweaver MX

Description

Interprets a customized .xml file that returns from the server that is specified in the Site Definition dialog box. The contents of the .xml file display as tree data in the Server Debug panel, so you can use the Server Debug panel to evaluate server-generated content from various server models.

Arguments

treedata

treedata is the XML string that the server returns. The XML string should use the following formatting:

server debug node	Root node for the debug xml data
debugnode	Corresponds to every node
context	Name of item that appears in the context list
icon	Icon to use for tree node
name	Name to display
value	Value to display
timestamp	Only applicable to context node

The following are optional:	
jumptoline	Link to a specific line number
template	Name of the template file part of the URL
path	Path of the file from server point of view
line number	Line number within the file
start position	Starting character offset within the line
end position	Ending character offset within the line

For example:

```
<serverdebuginfo>
  <context>
    <template><![CDATA[/ooo/master.cfm]]></template>
    <path><![CDATA[C:\server\wwwroot\ooo\master.cfm]]></path>
    <timestamp><![CDATA[0:0:0.0]]></timestamp>
  </context>
  <debugnode>
    <name><![CDATA[CGI]]></name>
    <icon><![CDATA[ServerDebugOutput/ColdFusion/CGIVariables.gif]]></icon>
    <debugnode>
      <name><![CDATA[Pubs.name.sourceURL]]></name>
      <icon><![CDATA[ServerDebugOutput/ColdFusion/Variable.gif]]></icon>
      <value><![CDATA[jdbc:macromedia:sqlserver://
  name.macromedia.com:1111;databaseName=Pubs]]></value>
    </debugnode>
  </debugnode>
  <debugnode>
    <name><![CDATA[Element Snippet is undefined in class
  coldfusion.compiler.TagInfoNotFoundException]]></name>
    <icon><![CDATA[ServerDebugOutput/ColdFusion/Exception.gif]]></icon>
    <jumptoline linenumber="3" startposition="2" endposition="20">
      <template><![CDATA[/ooo/master.cfm]]></template>
      <path><![CDATA[C:\Neo\wwwroot\ooo\master.cfm]]></path>
    </jumptoline>
  </debugnode>
</serverdebuginfo>
```

Returns

Nothing.

Selection functions

Selection functions get and set the selection in open documents. For information on getting or setting the selection in the Site panel, see "Site functions" on page 558.

dom.getSelectedNode()

Availability

Dreamweaver 3

Description

Gets the selected node. Using this function is equivalent to calling `dom.getSelection()` and passing the return value to `dom.offsetsToNode()`.

Arguments

None.

Returns

The tag, text, or comment object that completely contains the specified range of characters.

dom.getSelection()

Availability

Dreamweaver 3

Description

Gets the selection, which is expressed as character offsets into the document's source code.

Arguments

bAllowMultiple

- *bAllowMultiple* is a Boolean value that indicates whether the function should return multiple offsets if more than one table cell, image map hotspot, or layer is selected.

If this argument is omitted, it defaults to `false`.

Returns

For simple selections, this function returns an array that contains two integers. The first integer is the character offset of the beginning of the selection. The second integer is the character offset of the end of the selection. If the two numbers are the same, the current selection is an insertion point.

For complex selections (multiple table cells, multiple layers, or multiple image map hotspots), an array that contains 2*n* integers, where *n* is the number of selected items. The first integer in each pair is the character offset of the beginning of the selection (including the opening `TD`, `DIV`, `SPAN`, `LAYER`, `ILAYER`, or `MAP` tag); the second integer in each pair is the character offset of the end of the selection (including the closing `TD`, `DIV`, `SPAN`, `LAYER`, `ILAYER`, or `MAP` tag). If multiple table rows are selected, the offsets of each cell in each row are returned. The selection never includes the `TR` tags.

dom.nodeToOffsets()

Availability

Dreamweaver 3

Description

Gets the position of a specific node in the DOM tree, which is expressed as character offsets into the document's source code. It is valid for any document on a local drive.

Arguments

node

node must be a tag, comment, or range of text that is a node in the tree that `dreamweaver.getDocumentDOM()` returns.

Returns

An array that contains two integers. The first integer is the character offset of the beginning of the tag, text, or comment. The second integer is the character offset of the end of the node, from the beginning of the HTML document.

Enabler

None.

Example

The following code selects the first image object in the current document:

```
var theDOM = dw.getDocumentDOM();
var theImg = theDOM.images[0];
var offsets = theDom.nodeToOffsets(theImg);
theDom.setSelection(offsets[0], offsets[1]);
```

dom.offsetsToNode()

Availability

Dreamweaver 3

Description

Gets the object in the DOM tree that completely contains the range of characters between the specified beginning and end points. It is valid for any document on a local drive.

Arguments

offsetBegin, *offsetEnd*

The arguments are the beginning and end points, respectively, of a range of characters, expressed as character offsets from the beginning of the document's source code.

Returns

The tag, text, or comment object that completely contains the specified range of characters.

Example

The following code displays an alert if the selection is an image:

```
var offsets = dom.getSelection();
var theSelection = dreamweaver.offsetsToNode(offsets[0], ¬
offsets[1]);
if (theSelection.nodeType == Node.ELEMENT_NODE && ¬
theSelection.tagName == 'IMG'){
  alert('The current selection is an image.');
}
```

dom.selectAll()

Availability

Dreamweaver 3

Description

Performs a Select All operation.

Note: In most cases this function selects all the content in the active document. In some cases (for example, when the insertion point is inside a table), it selects only part of the active document. To set the selection to the entire document, use `dom.setSelection()`.

Arguments

None.

Returns

Nothing.

dom.selectTable()

Availability

Dreamweaver 3

Description

Selects an entire table.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canSelectTable()" on page 416

dom.setSelectedNode()

Availability

Dreamweaver 3

Description

Sets the selected node. This function is equivalent to calling `dom.nodeToOffsets()` and passing the return value to `dom.setSelection()`.

Arguments

node, *{bSelectInside}*, *{bJumpToNode}*

- *node* is a text, comment, or element node in the document.
- *bSelectInside* is a Boolean value that indicates whether to select the `innterHTML` of the node. This argument is relevant only if *node* is an element node, and it defaults to `false` if it is omitted.
- *bJumpToNode* is a Boolean value that indicates whether to scroll the Document window, if necessary, to make the selection visible. If it is omitted, this argument defaults to `false`.

Returns

Nothing.

dom.setSelection()

Availability

Dreamweaver 3

Description

Sets the selection in the document.

Arguments

offsetBegin, *offsetEnd*

The arguments are the beginning and end points, respectively, for the new selection, which is expressed as character offsets into the document's source code. If the two numbers are the same, the new selection is an insertion point. If the new selection is not a valid HTML selection, it is expanded to include the characters in the first valid HTML selection. For example, if *offsetBegin* and *offsetEnd* define the range `SRC="myImage.gif"` within `<IMG SRC="myImage.gif">`, the selection expands to include the entire `IMG` tag.

Returns

Nothing.

dreamweaver.nodeExists()

Available

Dreamweaver 3

Description

Determines whether the reference to the specified node is still good. Often when writing extensions, you reference a node and then perform an operation that deletes it (such as setting the `innerHTML` or `outerHTML` properties of its parent). This function lets you confirm that the node hasn't been deleted before you attempt to reference any of its properties or methods. The referenced node does not need to be in the current document.

Arguments

node is the node that you want to check.

Returns

A Boolean value that indicates whether the node exists.

Example

```
function applyFormatToSelectedTable(){

// get current selection
var selObj = dw.getDocumentDOM().getSelectedNode();

alternateRows(dwscripts.findDOMObject("presetNames").selectedIndex,
findTable());

// restore original selection, if it still exists; if not, just select the
// table.

var selArr;

if (dw.nodeExists(selObj))
  selArr = dom.nodeToOffsets(selObj);
else
  selArr = dom.nodeToOffsets(findTable());

dom.setSelection(selArr[0],selArr[1]);
}
}
```

dreamweaver.selectAll()

Availability

Dreamweaver 3

Description

Performs a Select All operation in the active Document window or the Site panel; or, on the Macintosh, the edit field that has focus in a dialog box or floating panel.

Note: If the operation takes place in the active document, it usually selects all the content in the active document. In some cases (for example, when the insertion point is inside a table), however, it selects only part of the active document. To set the selection to the entire document, use `dom.setSelection()`.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.canSelectAll()" on page 424

Server behavior functions

Server behavior functions let you manipulate the Server Behaviors panel, which you can display by selecting Window > Server Behaviors. Using these functions, you can find all the server behaviors on a page and programmatically apply a new behavior to the document or modify an existing behavior.

Note: You can abbreviate `dw.serverBehaviorInspector` to `dw.sbi`.

dreamweaver.serverBehaviorInspector.getServerBehaviors()

Availability

Dreamweaver UltraDev 1

Description

Gets a list of all the behaviors on the page. When Dreamweaver determines that the internal list of server behaviors might be out of date, it calls `findServerBehaviors()` for each currently installed behavior. Each function returns an array. Dreamweaver merges all the arrays into a single array and sorts it, based on the order that each behavior's `selectedNode` object appears in the document. Dreamweaver stores the merged array internally. The `getServerBehaviors()` function returns a pointer to that merged array.

Arguments

None.

Returns

An array of JavaScript objects. The objects in the array are returned by the `findServerBehaviors()` call. The objects are sorted in the order that they appear in the Server Behaviors panel.

dreamweaver.popupServerBehavior()

Availability

Dreamweaver UltraDev 1

Description

Applies a new server behavior to the document or modifies an existing behavior. If the user must specify parameters for the behavior, a dialog box appears.

Arguments

{behaviorName or *behaviorObject}*

- *behaviorName* is a string that represents the behavior's name, the title tag of a file, or a filename.
- *behaviorObject* is a behavior object.

If you omit the argument, Dreamweaver runs the currently selected server behavior. If the argument is the name of a server behavior, Dreamweaver adds the behavior to the page. If the argument is one of the objects in the array that is returned by `getServerBehaviors()`, a dialog box appears, so the user can modify the parameters for the behavior.

Returns

Nothing.

Server model functions

In Macromedia Dreamweaver MX, each document has an associated document type. For dynamic document types, Dreamweaver also associates a server model (such as ASP-JS, ColdFusion, or PHP-MySQL).

Server models are used to group functionality that is specific to a given server technology. Different server behaviors, data sources, and so forth, appear based on the server model that is associated with the document.

Using the server model functions, you can determine the set of server models that are currently defined; the name, language, and version of the current server model; and whether the current server model supports a named character set (such as UTF-8).

Note: Dreamweaver MX reads all the information in the server model HTML file and stores this information when it first loads the server model. So, when an extension calls functions such as `dom.serverModel.getServerName()`, `dom.serverModel.getServerLanguage()`, and `dom.serverModel.getServerVersion()`, these functions return the stored values.

dreamweaver.getServerModels()

Availability

Dreamweaver MX

Description

Gets the names for all the currently defined server models. The set of names is the same as the ones that appear in the Server Model field in the Site Definition dialog box in the user interface.

Arguments

None.

Returns

An array of strings. Each string element holds the name of a currently defined server model.

dom.serverModel.getAppURLPrefix()

Availability

Dreamweaver MX

Description

Returns the URL for the site's root folder on the testing server. This URL is the same as that specified for the Testing Server under the Advanced tab in the Site Definition dialog box.

When Dreamweaver communicates with your testing server, it uses HTTP (the same way as a browser). When doing so, it uses this URL to access your site's root folder.

Arguments

None.

Returns

A string, which holds the URL to the application server that is used for live data and debugging purposes.

Example

If the user creates a site and specifies that the testing server is on the local computer and that the root folder is named `"employeeapp"`, a call to `dom.serverModel.getAppURLPrefix()` returns this string:

```
http://localhost/employeeapp/
```

dom.serverModel.getDelimiters()

Availability

Dreamweaver MX

Description

Lets JavaScript code get the script delimiters for each server model, so managing the server model code can be separated from managing the user-scripted code.

Arguments

None.

Returns

An array of objects where each object contains the following three properties:

- *`startPattern`* is a regular expression that matches the opening script delimiter.
- *`endPattern`* is a regular expression that matches the closing script delimiter.
- *`participateInMerge`* is a Boolean value that specifies whether the content that is enclosed in the listed delimiters should (`true`) or should not (`false`) participate in block merging.

dom.serverModel.getDisplayName()

Availability

Dreamweaver MX

Description

Gets the name of the server model that appears in the user interface.

Arguments

None.

Returns

A string, the value of which is the name of the server model.

dom.serverModel.getFolderName()

Availability

Dreamweaver MX

Description

Gets the name of the folder that is used for this server model within the Configuration folder (such as in the ServerModels subfolder).

Arguments

None.

Returns

A string, the value of which is the name of the folder.

dom.serverModel.getServerExtension()

Availability

Dreamweaver UltraDev 4, deprecated in Dreamweaver MX

Description

Returns the default file extension of files that use the current server model. (The default file extension is the first in the list.) If no user document is currently selected, the `serverModel` object is set to the server model of the currently selected site.

Arguments

None.

Returns

A string that represents the supported file extensions.

dom.serverModel.getServerIncludeUrlPatterns()

Availability

Dreamweaver MX

Description

Returns the following list of properties, which let you access translator URL patterns:

- Translator URL patterns
- File references
- Type

Arguments

None.

Returns

A list of objects, one for each `searchPattern`. Each object has the following three properties:

Property	Description
`pattern`	A JavaScript regular expression that is specified in the `searchPattern` field of a .edml file that matches criteria. (A regular expression is delimited by a pair of forward slashes (`//`).)
`fileRef`	The 1-based index of the regular expression submatch that corresponds to the included file reference.
`type`	The portion of the `paramName` value that remains after removing the _includeUrl suffix. This type is assigned to the type attribute of the `<MM:BeginLock>` tag. For an example, see Server Model SSI.htm in the Configuration/Translators folder.

Example

The following snippet from a participant file illustrates a translator `searchPatterns` tag:

```
<searchPatterns whereToSearch="comment">
   <searchPattern paramNames=",ssi_comment_includeUrl">
      <![CDATA[/<!--\s*#include\s+(file|virtual)\s*=\s*"([^"]*)"\s*-->/i]]>
   </searchPattern>
</searchPatterns>
```

The search pattern contains a JavaScript regular expression that specifies two submatches (both of which are contained within parentheses). The first submatch is for the text string `file` or `virtual`. The second submatch is a file reference.

To access the translator URL pattern, your code should look like the following example:

```
var serverModel = dw.getDocumentDOM().serverModel;
var includeArray = new Array();
includeArray = serverModel.getServerIncludeUrlPatterns();
```

The call to `serverModel.getServerIncludeUrlPatterns()` returns the following three properties:

Property	Return value
`pattern`	`/<!--\s*#include\s+(file|virtual)\s*=\s*"([^"]*)"\s*-->/i`
`fileRef`	`2`
`type`	`ssi_comment`

dom.serverModel.getServerInfo()

Availability

Dreamweaver MX

Description

Returns information that is specific to the current server model. This information is defined in the HTML definition file for the server model, which is located in the Configuration/ServerModels folder.

You can modify the information in the HTML definition file or place additional variable values or functions in the file. For example, you can modify the `serverName`, `serverLanguage`, and `serverVersion` properties. The `dom.serverModel.getServerInfo()` function returns the information that the server model author adds to the definition file.

Note: The other values that are defined in the default server model files are for internal use only.

The `serverName`, `serverLanguage`, and `serverVersion` properties are special because the developer can access them directly by using the following corresponding functions:

- `dom.serverModel.getServerName()`
- `dom.serverModel.getServerLanguage()`
- `dom.serverModel.getServerVersion()`

Arguments

None.

Returns

A JavaScript object that contains a variety of information that is specific to the current server model.

dom.serverModel.getServerLanguage()

Availability

Dreamweaver 1, deprecated in Dreamweaver MX

Description

Determines the server model that is associated with the document and returns that value. The server language for a site is the value that comes from the Default Scripting Language setting in the App Server Info tab of the Site Definition dialog box. To get the return value, this function calls the `getServerLanguage()` function in the Server Models API.

Note: The Default Scripting Language list exists only in Dreamweaver 4 and earlier versions. For Dreamweaver MX, the Site Definition dialog box does not list supported scripting languages. Also, for Dreamweaver MX, the `dom.serverModel.getServerLanguage()` function reads the `serverLanguage` property of the object that is returned by a call to the `getServerInfo()` function in the Server Models API.

Arguments

None.

Returns

A string that contains the supported scripting languages.

dom.serverModel.getServerName()

Availability

Dreamweaver 1, enhanced in Dreamweaver MX

Description

Determines the server model that is associated with the document and returns that value. Possible values include ASP.NET C#, ASP.NET VB, ASP VBScript, ASP JavaScript, ColdFusion, JSP, PHP MySQL, and any additional files that are contained in the Configuration/ServerModels folder.

Note: For Dreamweaver MX, `dom.serverModel.getServerName()` reads the `serverName` property of the object that is returned by a call to the `getServerInfo()` function in the Server Models API.

Arguments

None.

Returns

A string that contains the server name.

dom.serverModel.getServerSupportsCharset()

Availability

Dreamweaver MX

Description

Determines whether the server model that is associated with the document supports the named character set.

Note: In addition to letting you call this function from the JavaScript layer, Dreamweaver MX calls this function when the user changes the encoding in the page Properties dialog box. If the server model does not support the new character encoding, this function returns `false` and Dreamweaver pops up a warning dialog box that asks if the user wants to do the conversion. An example of this situation is when a user attempts to convert a ColdFusion 4.5 document to UTF-8 because ColdFusion does not support UTF-8 encoding.

Arguments

`metaCharSetString`

`metaCharSetString` is a string value that names a particular character set. This value is the same as that of the `"charset="` attribute of a `<meta>` tag that is associated with a document. Supported values for a given server model are defined in the HTML definition file for the server model, which is located in the Configuration/ServerModels folder.

Returns

A Boolean value. The `getServerSupportsCharset()` function returns `true` if the server model supports the named character set; `false` otherwise.

dom.serverModel.getServerVersion()

Availability

Dreamweaver 1, enhanced in Dreamweaver MX

Description

Determines the server model that is associated with the document and returns that value. Each server model has a `getVersionArray()` function, as defined in the Server Models API, which returns a table of name-version pairs.

Note: For Dreamweaver MX, `dom.serverModel.getServerVersion()` first reads the `serverVersion` property of the object that is returned by a call to `getServerInfo()` in the Server Models API. If that property does not exist, `dom.serverModel.getServerVersion()` reads it from the `getVersionArray()` function.

Arguments

`name`

`name` is a string that represents the name of a server model.

Returns

A string that contains the version of the named server model.

dom.serverModel.testAppServer()

Availability

Dreamweaver MX

Description

Tests whether a connection to the application server can be made.

Arguments

None.

Returns

A Boolean value that indicates whether the request to connect to the application server was successful.

Site functions

Site functions handle operations that are related to files in the site files or site map. These functions let you perform the following tasks:

- Create links between files
- Get, put, check in, and check out files
- Select and deselect files
- Create and remove files
- Get information about the sites that the user has defined
- Import and export site information

dreamweaver.loadSitesFromPrefs()

Availability

Dreamweaver 4

Description

Loads the site information for all the sites from the system registry (Windows) or the Dreamweaver Preferences file (Macintosh) into Dreamweaver. If a site is connected to a remote server when this function is called, the site is automatically disconnected.

Arguments

None.

Returns

Nothing.

dreamweaver.saveSitesToPrefs()

Availability

Dreamweaver 4

Description

Saves all information for each site that the user has defined to the system registry (Windows) or the Dreamweaver Preferences file (Macintosh).

Arguments

None.

Returns

Nothing.

site.addLinkToExistingFile()

Availability

Dreamweaver 3

Description

Opens the Select HTML File dialog box to let the user select a file and creates a link from the selected document to that file

Arguments

None.

Returns

Nothing.

Enabler

"site.canAddLink()" on page 433

site.addLinkToNewFile()

Availability

Dreamweaver 3

Description

Opens the Link to New File dialog box to let the user specify details for the new file and creates a link from the selected document to that file

Arguments

None.

Returns

Nothing.

Enabler

"site.canAddLink()" on page 433

site.canEditColumns()

Description

Determines whether a site exists.

Arguments

None.

Returns

`true` if a site exists; otherwise `false`.

site.changeLinkSitewide()

Availability

Dreamweaver 3

Description

Opens the Change Link Sitewide dialog box.

Arguments

None.

Returns

Nothing.

site.changeLink()

Availability

Dreamweaver 3

Description

Opens the Select HTML File dialog box to let the user select a new file for the link.

Arguments

None.

Returns

Nothing.

Enabler

"site.canChangeLink()" on page 433

site.checkIn()

Availability

Dreamweaver 3

Description

Checks in the selected files and handles dependent files in one of the following ways:

- If the user selects Prompt on Put/Check In in the Site FTP preferences, the Dependent Files dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked Yes, dependent files are uploaded and no dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked No, dependent files are not uploaded and no dialog box appears.

Arguments

siteOrURL

siteOrURL must be the keyword `"site"`, which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canCheckIn()" on page 433

site.checkLinks()

Availability

Dreamweaver 3

Description

Opens the Link Checker dialog box and checks links in the specified files.

Arguments

scopeOfCheck

scopeOfCheck specifies where links will be checked. It must be `"document"`, `"selection"`, or `"site"`.

Returns

Nothing.

site.checkOut()

Availability

Dreamweaver 3

Description

Checks out the selected files and handles dependent files in one of the following ways:

- If the user selects Prompt on Get/Check Out in the Site FTP preferences, the Dependent Files dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked Yes, dependent files are downloaded and no dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked No, dependent files are not downloaded and no dialog box appears.

Arguments

siteOrURL

siteOrURL must be the keyword `"site"`, which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canCheckOut()" on page 434

site.checkTargetBrowsers()

Availability

Dreamweaver 3

Description

Runs a target browser check on the selected files.

Arguments

None.

Returns

Nothing.

site.cloak()

Availability

Dreamweaver MX

Description

Cloaks the current selection in the Site panel or the specified folder.

Arguments

siteOrURL

siteOrURL must contain one of the following two values:

- The keyword `"site"`, which indicates that `cloak()` should act on the selection in the Site panel
- The URL of a particular folder, which indicates that `cloak()` should act on the specified folder and all its contents

Returns

Nothing.

Enabler

"site.canCloak()" on page 434

site.defineSites()

Availability

Dreamweaver 3

Description

In Dreamweaver MX, opens the Edit Sites dialog box; in Dreamweaver 4 and earlier versions, opens the Edit Sites dialog box.

Arguments

None.

Returns

Nothing.

site.deleteSelection()

Availability

Dreamweaver 3

Description

Deletes the selected files.

Arguments

None.

Returns

Nothing.

site.editColumns()

Description

In Dreamweaver MX, displays the Edit Sites dialog box; in Dreamweaver 4 and earlier versions, displays the Edit Sites dialog box. Both dialog boxes show the File View Columns section.

Arguments

None.

Returns

Nothing.

site.exportSite()

Availability

Dreamweaver MX

Description

Exports a Dreamweaver site to an XML file, which can be imported into another Dreamweaver instance to duplicate the former site.

All the information that is contained in the Site Definition dialog box is saved in an XML file that includes the list of cloaked folders and information about the default document type. The exception is that the user can omit the user login and password when FTP access is set. The following example shows a sample XML file that Dreamweaver creates when you export a site.

```
<?xml version="1.0" ?>
<site>
  <localinfo
    sitename="DW00"
    localroot="C:\Documents and Settings\jlondon\Desktop\DWServer\"
    imagefolder="C:\Documents and Settings\jlondon\Desktop\DWServer\Images\"
    spacerfilepath=""
    refreshlocal="TRUE"
    cache="FALSE"
    httpaddress="http://" curserver="webserver" />
  <remoteinfo
    accesstype="ftp"
    host="dreamweaver"
    remoteroot="kojak/"
    user="dream"
    checkoutname="Jay"
    emailaddress="jay@macromedia.com"
    usefirewall="FALSE"
    usepasv="TRUE"
    enablecheckin="TRUE"
    checkoutwhenopen="TRUE" />
  <designnotes
    usedesignnotes="TRUE"
    sharedesignnotes="TRUE" />
  <sitemap
```

```
    homepage="C:\Documents and Settings\jlondon\Desktop\DWServer\Untitled-2.htm"
    pagesperrow="200" columnwidth="125" showdependentfiles="TRUE"
    showpagetitles="FALSE" showhiddenfiles="TRUE" />
  <fileviewcolumns sharecolumns="TRUE">
    <column name="Local Folder"
      align="left" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="180" remotewidth="180" />
    <column name="Notes"
      align="center" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="36" remotewidth="36" />
    <column name="Size"
      align="right" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="-2" remotewidth="-2" />
    <column name="Type"
      align="left" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="60" remotewidth="60" />
    <column name="Modified"
      align="left" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="102" remotewidth="102" />
    <column name="Checked Out By"
      align="left" show="TRUE" share="FALSE" builtin="TRUE"
      localwidth="50" remotewidth="50" />
    <column name="Status" note="status"
      align="left" show="TRUE" share="FALSE" builtin="FALSE"
      localwidth="50" remotewidth="50" />
  </fileviewcolumns>
  <appserverinfo
    servermodel="ColdFusion"
    urlprefix="http://dreamweaver/kojak/"
    serverscripting="CFML"
    serverpageext=""
    connectionsmigrated="TRUE"
    useUD4andUD5pages="TRUE"
    defaultdoctype=""
    accesstype="ftp"
    host="dreamweaver"
    remoteroot="kojak/"
    user="dream"
    usefirewall="FALSE"
    usepasv="TRUE" />
  <cloaking enabled="TRUE" patterns="TRUE">
    <cloakedfolder folder="databases/" />
    <cloakedpattern pattern=".png" />
    <cloakedpattern pattern=".jpg" />
    <cloakedpattern pattern=".jpeg" />
  </cloaking>
</site>
```

Arguments

siteName

siteName identifies the site to export. If *siteName* is an empty string, Dreamweaver exports the current site.

Returns

A Boolean value. `exportSite()` returns `true` if the named site exists and if the XML file is successfully exported; `false` otherwise.

site.findLinkSource()

Availability

Dreamweaver 3

Description

Opens the file that contains the selected link or dependent file, and highlights the text of the link or the reference to the dependent in that file. This function operates only on files in the Site Map view.

Arguments

None.

Returns

Nothing.

Enabler

"site.canFindLinkSource()" on page 435

site.get()

Availability

Dreamweaver 3

Description

Gets the specified files and handles dependent files in one of the following ways:

- If the user selects Prompt on Get/Check Out in the Site FTP preferences, the Dependent Files dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked Yes, dependent files are downloaded and no dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked No, dependent files are not downloaded and no dialog box appears.

Arguments

siteOrURL

siteOrURL must be the keyword `"site"`, which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canGet()" on page 435

site.getAppServerAccessType()

Availability

Dreamweaver MX

Description

Returns the access method that is used for all files on the current site's application server. The current site is the site that is associated with the document that currently has focus. If no document has focus, the site that you opened in Dreamweaver MX is used.

Note: ColdFusion Component Explorer makes use of this function, see "site.getAppServerPathToFiles()" on page 566, and "site.getLocalPathToFiles()" on page 569.

Arguments

None.

Returns

One of the following strings:

- `none`
- `local/network`
- `ftp`
- `source_control`

site.getAppServerPathToFiles()

Availability

Dreamweaver MX

Description

Determines the disk path to the remote files on the application server that is defined for the current site. The current site is the site that is associated with the document that currently has focus. If no document has focus, the site opened in Dreamweaver MX is used.

Note: ColdFusion Component Explorer makes use of this function, see "site.getAppServerAccessType()" on page 565, and "site.getLocalPathToFiles()" on page 569.

Arguments

None.

Returns

If the access type to the application server file is `local/network`, this function returns a path; otherwise, this function returns an empty string.

site.getCheckOutUser()

Availability

Dreamweaver 3

Description

Gets the login and check-out name that is associated with the current site.

Arguments

None.

Returns

A string that contains a login and check-out name, if defined, or an empty string if check-in/check-out is disabled

Example

A call to `site.getCheckOutUser()` might return `"denise (deniseLaptop)"`. If no check-out name is specified, only the login is returned (for example, `"denise"`).

site.getCheckOutUserForFile()

Availability

Dreamweaver 3

Description

Gets the login and check-out name of the user who has the specified file checked out.

Arguments

`fileName`

`fileName` is the path to the file being queried, which is expressed as a file:// URL.

Returns

A string that contains the login and check-out name of the user who has the file checked out or an empty string if the file is not checked out.

Example

A call to `site.getCheckOutUserForFile("file:/C:/sites/avocado8/index.html")` might return `"denise (deniseLaptop)"`. If no check-out name is specified, only the login returns (for example, `"denise"`).

site.getCloakingEnabled()

Availability

Dreamweaver MX

Description

Determines whether cloaking is enabled for the current site.

Arguments

None.

Returns

A Boolean value. The `getCloakingEnabled()` function returns `true` if cloaking is enabled for the current site; `false` otherwise.

site.getConnectionState()

Availability

Dreamweaver 3

Description

Gets the current connection state.

Arguments

None.

Returns

A Boolean value that indicates whether the remote site is connected.

Enabler

"site.canConnect()" on page 435

site.getCurrentSite()

Availability

Dreamweaver 3

Description

Gets the current site.

Arguments

None.

Returns

A string that contains the name of the current site.

Example

If you defined several sites, a call to `site.getCurrentSite()` returns the one that is currently showing in the Current Sites List in the Site panel.

site.getFocus()

Availability

Dreamweaver 3

Description

Determines which pane of the Site panel has the focus.

Arguments

None.

Returns

One of the following strings: `"local"`, `"remote"`, or `"site map"`.

site.getLinkVisibility()

Availability

Dreamweaver 3

Description

Checks whether all the selected links in the site map are visible (that is, not marked hidden).

Arguments

None.

Returns

A Boolean value that indicates whether all the selected links are visible.

site.getLocalPathToFiles()

Availability

Dreamweaver MX

Description

Determines the disk path to the local files that are defined for the current site. The current site is the site that is associated with the document that currently has focus. If no document has focus, the site that you opened in Dreamweaver MX is used.

Note: ColdFusion Component Explorer makes use of this function, "site.getAppServerAccessType()" on page 565, and "site.getAppServerPathToFiles()" on page 566.

Arguments

None.

Returns

Path to the files residing on the local machine for the current site.

site.getSelection()

Availability

Dreamweaver 3

Description

Determines which files are currently selected in the Site panel.

Arguments

None.

Returns

An array of strings that represents the paths of the selected files and folders, which is expressed as a file:// URLs; or an empty array if no files or folders are selected.

site.getSiteForURL()

Availability

Dreamweaver MX

Description

Gets the name of the site, if any, that is associated with a specific file.

Arguments

`fileURL`

`fileURL` is the fully qualified URL (including the string *`file://`*) for a named file.

Returns

A string that contains the name of the site, if any, in which the specified file exists. The string is empty when the specified file does not exist in any defined site.

site.getSites()

Availability

Dreamweaver 3

Description

Gets a list of the defined sites.

Arguments

None.

Returns

An array of strings that represents the names of the defined sites, or an empty array if no sites are defined.

site.importSite()

Availability

Dreamweaver MX

Description

Creates a Dreamweaver site from an XML file. During import, if the folder that is specified by the `localroot` attribute of the `<localinfo>` element does not exist on the local computer, Dreamweaver prompts for a different local root folder. Dreamweaver behaves the same way when it tries to locate the default images folder that is specified by the `imagefolder` attribute of the `<localinfo>` element.

Arguments

`fileURL`

`fileURL` is a string that contains the URL for the XML file. Dreamweaver uses this XML file to create a new site. If `fileURL` is an empty string, Dreamweaver prompts the user to select an XML file to import.

Returns

A Boolean value. The `importSite()` function returns `true` if the named XML file exists and if the site is successfully created; `false` otherwise.

site.invertSelection()

Availability

Dreamweaver 3

Description

Inverts the selection in the site map.

Arguments

None.

Returns

Nothing.

site.isCloaked()

Availability

Dreamweaver MX

Description

Determines whether the current selection in the Site panel or the specified folder is cloaked.

Arguments

siteOrURL

- The keyword `"site"`, which indicates that `isCloaked()` should test the selection in the Site panel
- The URL of a particular folder, which indicates that `isCloaked()` should test the specified folder

Returns

A Boolean value. The `isCloaked()` function returns `true` if the specified object is cloaked; `false` otherwise.

site.locateInSite()

Availability

Dreamweaver 3

Description

Locates the specified file (or files) in the specified pane of the Site panel and selects the found files.

Arguments

localOrRemote, *siteOrURL*

- *localOrRemote* must be either `"local"` or `"remote"`.
- *siteOrURL* must be the keyword `"site"`, which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canLocateInSite()" on page 436

site.makeEditable()

Availability

Dreamweaver 3

Description

Turns off the read-only flag on the selected files.

Arguments

None.

Returns

Nothing.

Enabler

"site.canMakeEditable()" on page 436

site.makeNewDreamweaverFile()

Availability

Dreamweaver 3

Description

Creates a new Dreamweaver file in the Site panel in the same directory as the first selected file or folder.

Arguments

None.

Returns

Nothing.

Enabler

"site.canMakeNewFileOrFolder()" on page 436

site.makeNewFolder()

Availability

Dreamweaver 3

Description

Creates a new folder in the Site panel in the same directory as the first selected file or folder.

Arguments

None.

Returns

Nothing.

Enabler

"site.canMakeNewFileOrFolder()" on page 436

site.newHomePage()

Availability

Dreamweaver 3

Description

Opens the New Home Page dialog box to let the user create a new home page.

Note: This function operates only on files in the Site Map view.

Arguments

None.

Returns

Nothing.

site.newSite()

Availability

Dreamweaver 3

Description

Opens the Site Definition dialog box for a new, unnamed site.

Arguments

None.

Returns

Nothing.

site.open()

Availability

Dreamweaver 3

Description

Opens the files that are currently selected in the Site panel. If any folders are selected, they are expanded in the Site Files view.

Arguments

None.

Returns

Nothing.

Enabler

"site.canOpen()" on page 437

site.put()

Availability

Dreamweaver 3

Description

Puts the selected files and handles dependent files in one of the following ways:

- If the user selects Prompt on Put/Check In in the Site FTP preferences, the Dependent Files dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked Yes, dependent files are uploaded and no dialog box appears.
- If the user previously selected the Don't Show Me Again option in the Dependent Files dialog box and clicked No, dependent files are not uploaded and no dialog box appears.

Arguments

siteOrURL

siteOrURL must be the keyword `"site"`, which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canPut()" on page 437

site.recreateCache()

Availability

Dreamweaver 3

Description

Recreates the cache for the current site.

Arguments

None.

Returns

Nothing.

Enabler

"site.canRecreateCache()" on page 437

site.refresh()

Availability

Dreamweaver 3

Description

Refreshes the file listing on the specified side of the Site panel.

Arguments

whichSide

whichSide must be "local", or "remote". If the site map has focus and *whichSide* is "local", the site map refreshes.

Returns

Nothing.

Enabler

"site.canRefresh()" on page 438

site.remoteIsValid()

Availability

Dreamweaver 3

Description

Determines whether the remote site is valid.

Arguments

None.

Returns

A Boolean value that indicates whether a remote site has been defined and, if the server type is Local/Network, whether the drive is mounted.

site.removeLink()

Availability

Dreamweaver 3

Description

Removes the selected link from the document above it in the site map.

Arguments

None.

Returns

Nothing.

Enabler

"site.canRemoveLink()" on page 438

site.renameSelection()

Availability

Dreamweaver 3

Description

Turns the name of the selected file into an text field, so the user can rename the file. If more than one file is selected, this function acts on the last selected file.

Arguments

None.

Returns

Nothing.

site.runValidation()

Availability

Dreamweaver MX

Description

Runs the Validator on the entire site or only highlighted items.

Arguments

selection

selection is the parameter that specifies that the Validator should check only the highlighted items; otherwise, the Validator checks the entire current site.

Returns

Nothing.

Enabler

"canAcceptCommand()" on page 62

site.saveAsImage()

Availability

Dreamweaver 3

Description

Opens the Save As dialog box to let the user save the site map as an image.

Arguments

fileType

fileType is the type of image that should be saved. Valid values for Windows are "bmp" and "png"; valid values for Macintosh are "pict" and "jpeg". If the argument is omitted, or if the value is not valid on the current platform, the default is "bmp" in Windows and "pict" on the Macintosh.

Returns

Nothing.

site.selectAll()

Availability

Dreamweaver 3

Description

Selects all files in the active view (either the site map or the site files).

Arguments

None.

Returns

Nothing.

site.selectHomePage()

Availability

Dreamweaver 3

Description

Opens the Open File dialog box to let the user choose a new home page.

Note: This function operates only on files in the Site Map view.

Arguments

None.

Returns

Nothing.

site.selectNewer()

Availability

Dreamweaver 3

Description

Selects all files that are newer on the specified side of the Site panel.

Arguments

whichSide

whichSide must be either "local" or "remote".

Returns

Nothing.

Enabler

"site.canSelectNewer()" on page 439

site.setAsHomePage()

Availability

Dreamweaver 3

Description

Designates the file that is selected in the Site Files view as the home page for the site.

Arguments

None.

Returns

Nothing.

site.setCloakingEnabled()

Availability

Dreamweaver MX

Description

Determines whether cloaking should be enabled for the current site.

Arguments

enable

enable is a Boolean value that indicates whether cloaking should be enabled. A value of true enables cloaking for the current site; a value of false disables cloaking for the current site.

Returns

None.

site.setConnectionState()

Availability

Dreamweaver 3

Description

Sets the connection state of the current site.

Arguments

bConnected

bConnected is a Boolean value that indicates if there is a connection (`true`) or not (`false`) to the current site.

Returns

Nothing.

site.setCurrentSite()

Availability

Dreamweaver 3

Description

Opens the specified site in the local pane of the Site panel.

Arguments

whichSite

whichSite is the name of a defined site (as it appears in the Current Sites list in the Site panel or the Edit Sites dialog box).

Returns

Nothing.

Example

If three sites are defined (for example, avocado8, dreamcentral, and testsite), a call to `site.setCurrentSite("dreamcentral");` makes dreamcentral the current site.

site.setFocus()

Availability

Dreamweaver 3

Description

Gives focus to a specified pane in the Site panel. If the specified pane is not showing, this function displays the pane and gives it focus.

Arguments

whichPane

whichPane must be one of the following strings: `"local"`, `"remote"`, or `"site map"`.

Returns

Nothing.

site.setLayout()

Availability

Dreamweaver 3

Description

Opens the Site Map Layout pane of the Site Definition dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"site.canSetLayout()" on page 438

site.setLinkVisibility()

Availability

Dreamweaver 3

Description

Shows or hides the current link.

Arguments

bShow

bShow is a Boolean value that indicates whether to remove the Hidden designation from the current link.

Returns

Nothing.

site.setSelection()

Availability

Dreamweaver 3

Description

Selects files or folders in the active pane in the Site panel.

Arguments

arrayOfURLs

arrayOfURLs is an array of strings where each string is a path to a file or folder in the current site, which is expressed as a file:// URL.

Note: Omit the trailing slash (/) when specifying folder paths.

Returns

Nothing.

site.synchronize()

Availability

Dreamweaver 3

Description

Opens the Synchronize Files dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"site.canSynchronize()" on page 439

site.uncloak()

Availability

Dreamweaver MX

Description

Uncloaks the current selection in the Site panel or the specified folder.

Arguments

siteOrURL

siteOrURL must contain one of the following two values:

- The keyword "site", which indicates that unCloak() should act on the selection in the Site panel
- The URL of a particular folder, which indicates that unCloak() should act on the specified folder and all its contents

Returns

Nothing.

Enabler

"site.canUncloak()" on page 440

site.uncloakAll()

Availability

Dreamweaver MX

Description

Uncloaks all folders in the current Site and unchecks the Cloak Files Ending With: checkbox in the Cloaking Settings.

Arguments

Nothing.

Returns

Nothing.

Enabler

"site.canUncloak()" on page 440

site.undoCheckOut()

Availability

Dreamweaver 3

Description

Removes the lock files that are associated with the specified files from the local and remote sites, and replaces the local copy of the specified files with the remote copy.

Arguments

siteOrURL

siteOrURL must be the keyword "site", which indicates that the function should act on the selection in the Site panel or on the URL for a single file.

Returns

Nothing.

Enabler

"site.canUndoCheckOut()" on page 440

site.viewAsRoot()

Availability

Dreamweaver 3

Description

Temporarily moves the selected file to the top position in the site map.

Arguments

None.

Returns

Nothing.

Enabler

"site.canViewAsRoot()" on page 440

Snippets panel functions

Using Macromedia Dreamweaver MX, web developers can edit and save reusable blocks of code in the Snippets panel and retrieve them as needed.

The Snippets panel stores each code snippet in a .csn file within the Configuration/Snippets folder. Snippets that ship with Dreamweaver MX are stored in the following folders:

- Accessible
- Comments
- Content_tables
- Filelist.txt
- Footers
- Form_elements
- Headers
- Javascript
- Meta
- Navigation
- Text

Snippet files are XML documents, so you can specify the encoding in the XML directive as in the following example:

```
<?XML version="1.0" encoding="utf-8">
```

The following sample shows a snippet file:

```
<snippet name="Detect Flash" description="VBscript to check for Flash
  ActiveX control" preview="code" factory="true" type="wrap" >
     <insertText location="beforeSelection">
          <![CDATA[ ------- code --------- ]]>
     </insertText>
     <insertText location="afterSelection">
          <![CDATA[ ------- code --------- ]]>
     </insertText>
</snippet>
```

Snippet tags in .csn files have the following attributes:

Attribute	Description
`name`	name of snippet
`description`	snippet description
`preview`	Type of preview: "`code`" to display the snippet in preview area or "`design`" to display the snippet rendered in HTML in the Preview area.
`type`	If the snippet is used to wrap a user selection, "`wrap`"; if the snippet should be inserted before the selection, "`block`".

You can use the following methods to add Snippets panel functions to your extensions.

dreamweaver.snippetPalette.newFolder()

Availability

Dreamweaver MX

Description

Creates a new folder with the default name *untitled* and puts an text box around the default name.

Arguments

None.

Returns

Nothing.

dreamweaver.snippetPalette.newSnippet()

Availability

Dreamweaver MX

Description

Opens the Add Snippet dialog box and brings it to the front.

Arguments

None.

Returns

Nothing.

dreamweaver.snippetPalette.editSnippet()

Availability

Dreamweaver MX

Description

Opens the Edit Snippet dialog box and brings it to the front, enabling editing for the selected element.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.snippetpalette.canEditSnippet()" on page 428

dreamweaver.snippetPalette.insert()

Availability

Dreamweaver MX

Description

Applies selected snippet from Panel to the current selection.

Arguments

None.

Returns

Nothing.

Enabler

"dw.snippetpalette.canInsert()" on page 429

dreamweaver.snippetPalette.insertSnippet()

Availability

Dreamweaver MX

Description

Inserts indicated snippet to current selection.

Arguments

Snippet path relative to snippet folder.

Returns

A Boolean value.

Enabler

"dw.snippetpalette.canInsert()" on page 429

dreamweaver.snippetPalette.rename()

Availability

Dreamweaver MX

Description

Activates text box around selected folder name or file nickname and lets you edit the selected element.

Arguments

None.

dreamweaver.snippetPalette.remove()

Availability

Dreamweaver MX

Description

Deletes selected element or folder from Snippets Panel and deletes file from disk.

Return Value

None.

String manipulation functions

String manipulation functions help you get information about a string as well as convert a string from Latin 1 encoding to platform-native encoding and back.

dreamweaver.doURLEncoding()

Availability

Dreamweaver 1

Description

This function takes a string and returns a URL-encoded string by replacing all spaces and special characters with specified entities.

Arguments

stringToConvert

Returns

A URL-encoded string.

Example

The following example shows the `URL.value` for `"My URL-encoded string"`:

```
var URL = dw.doURLEncoding(theURL.value);
returns "My%20URL-encoded%20string"
```

dreamweaver.getTokens()

Availability

Dreamweaver 1

Description

Accepts a string and splits it into tokens.

Arguments

searchString, *separatorCharacters*

- *searchString* is the string to be separated into tokens.
- *separatorCharacters* is the character or characters that signifies the end of a token. Separator characters in quoted strings are ignored. If *separatorCharacters* contains a space, all white-space characters (such as tabs) are treated as separator characters, as if they are explicitly specified. Two or more consecutive white space characters are treated as a single separator.

Returns

An array of token strings.

Example

`dreamweaver.getTokens('foo("my arg1", 34)', '(),')` returns the tokens:

- `foo`
- `"my arg 1"`
- `34`

dreamweaver.latin1ToNative()

Availability

Dreamweaver 2

Description

Converts a string in Latin 1 encoding to the native encoding on the user's computer. This function is intended for displaying the user interface of an extension file in another language.

Note: This function has no effect in Windows because Windows encodings are already based on Latin 1.

Arguments

stringToConvert

stringToConvert is the string to convert from Latin 1 encoding to native encoding.

Returns

The converted string.

dreamweaver.nativeToLatin1()

Availability

Dreamweaver 2

Description

Converts a string in native encoding to Latin 1 encoding.

Note: This function has no effect in Windows because Windows encodings are already based on Latin 1.

Arguments

stringToConvert

stringToConvert is the string to convert from native encoding to Latin 1 encoding.

Returns

The converted string.

Enabler

None.

dreamweaver.scanSourceString()

Availability

Dreamweaver UltraDev 1

Description

Scans a string of HTML and finds the tags, attributes, directives, and text. For each tag, attribute, directive, and text span that it finds, `scanSourceString()` invokes a callback function that you must supply. Dreamweaver supports the following callback functions: `openTagBegin()`, `openTagEnd()`, `closeTagBegin()`, `closeTagEnd()`, `directive()`, `attribute()`, and `text()`.

Dreamweaver calls the seven callback functions on the following occasions:

1. Dreamweaver calls `openTagBegin()` for each opening tag (for example, `<font>`, as opposed to `</font>`) and each empty tag (for example, `<img>` or `<hr>`). The `openTagBegin()` function accepts two arguments: the name of the tag (for example, `"font"` or `"img"`) and the document offset, which is the number of bytes in the document before the beginning of the tag. The function returns `true` if scanning should continue or `false` if it should stop.
2. After `openTagBegin()` executes, Dreamweaver calls `attribute()` for each HTML attribute. The `attribute()` function accepts two arguments, a string that contains the attribute name (for example, `"color"` or `"src"`) and a string that contains the attribute value (for example, `"#000000"` or `"foo.gif"`). The `attribute()` function returns a Boolean value that indicates whether scanning should continue.
3. After all the attributes in the tag have been scanned, Dreamweaver calls `openTagEnd()`. The `openTagEnd()` function accepts one argument, the document offset, which is the number of bytes in the document before the end of the opening tag. It returns a Boolean value that indicates whether scanning should continue.
4. Dreamweaver calls `closeTagBegin()` for each closing tag (for example, `</font>`). The function accepts two arguments, the name of the tag to close (for example, `"font"`) and the document offset, which is the number of bytes in the document before the beginning of the close tag. The function returns a Boolean value that indicates whether scanning should continue.
5. After `closeTagBegin()` returns, Dreamweaver calls the `closeTagEnd()` function. The `closeTagEnd()` function accepts one argument, the document offset, which is the number of bytes in the document before the end of the closing tag. It returns a Boolean value that indicates whether scanning should continue.
6. Dreamweaver calls the `directive()` function for each HTML comment, ASP script, JSP script, or PHP script. The `directive()` function accepts two arguments, a string that contains the directive and the document offset, which is the number of bytes in the document before the end of the closing tag. The function returns a Boolean value that indicates whether scanning should continue.
7. Dreamweaver calls the `text()` function for each span of text in the document; that is, everything that is not a tag or a directive. Text spans include text that is not visible to the user, such as the text inside a `<title>` or `<option>` tag. The `text()` function accepts two arguments, a string that contains the text and the document offset, which is the number of bytes in the document before the end of the closing tag. The `text()` function returns a Boolean value that indicates whether scanning should continue.

Arguments

`HTMLstr`, *`parserCallbackObj`*

- *`HTMLstr`* is a string that contains code.
- *`parserCallbackObj`* is a JavaScript object that has one or more of the following methods: `openTagBegin()`, `openTagEnd()`, `closeTagBegin()`, `closeTagEnd()`, `directive()`, `attribute()`, and `text()`. For best performance, *`parserCallbackObj`* should be a shared library that is defined using the C-Level Extensibility interface. Performance is also improved if *`parserCallbackObj`* defines only the callback functions that it needs.

Returns

A Boolean value that indicates whether the operation completed successfully.

Example

The following sequence of steps provide an example of how to use the `dreamweaver.scanSourceString()` function.

Steps

1. You create an implementation for one or more of the seven callback functions.
2. You write a script that calls the `dreamweaver.scanSourceString()` function.
3. The `dreamweaver.scanSourceString()` function passes a string that contains HTML and pointers to the callback functions that you wrote. For example, suppose the string of HTML is `"<font size=2>hello</font>"`.
4. Dreamweaver analyzes the string and determines that the string contains a font tag. Dreamweaver calls the callback functions in the following sequence:

- The `openTagBegin()` function
- The `attribute()` function (for the size attribute)
- The `openTagEnd()` function
- The `text()` function (for the `"hello"` string)
- The `closeTagBegin()` and `closeTagEnd()` functions

Source view functions

Source view functions include operations that are related to editing document source code (and that have subsequent impact on the Design view). The functions in this section let you add navigational controls to Code views within a split document view, the Code inspector, and the JavaScript Debugger window.

dom.formatRange()

Availability

Dreamweaver MX

Description

Applies Dreamweaver automatic syntax formatting to a specified range of characters in the source view according to the settings in the Preferences > Code Format dialog box.

Arguments

`startOffset`, *`endOffset`*

- *`startOffset`* is an integer that represents the beginning of the specified range as the offset from the beginning of the document.
- *`endOffset`* is an integer representing the end of the specified range as the offset from the beginning of the document.

Returns

Nothing.

dom.formatSelection()

Availability

Dreamweaver MX

Description

Applies Dreamweaver automatic syntax formatting to the selected content (the same as choosing the Commands > Apply Source Formatting to Selection option) according to the settings in the Preferences > Code Format dialog box.

Arguments

None.

Returns

Nothing.

dom.getShowNoscript()

Availability

Dreamweaver MX

Description

Gets the current state of the `noscript` content option (from the view > Noscript Content menu option). On by default, the `noscript` tag identifies page script content that can be rendered, or not (by choice), in the browser.

Arguments

None.

Returns

A Boolean value that indicates whether the noscript tag content is currently rendered. If `bIsVisible` is `true`, the content appears; `false` otherwise.

dom.isDesignviewUpdated()

Availability

Dreamweaver 4

Description

Determines whether the Design view and Text view content is synchronized for those Dreamweaver operations that require a valid document state.

Arguments

None.

Returns

`true` if the Design view (WYSIWYG) is synchronized with the text in the Text view; `false` otherwise.

dom.isSelectionValid()

Availability

Dreamweaver 4

Description

Determines whether a selection is valid, meaning it is currently synchronized with the Design view, or if it needs to be moved before an operation occurs.

Arguments

None.

Returns

`true` if the current selection is in a valid piece of code. If the document has not been synchronized, returns `false` (because the selection is not updated).

dom.setShowNoscript

Availability

Dreamweaver MX

Description

Sets the `noscript` content option on or off (the same as choosing the View > Noscript Content option). On by default, the `noscript` tag identifies page script content that can be rendered, or not (by choice), in the browser.

Arguments

`{bShowNoscript}`

- *`bShowNoscript`* is a Boolean value that indicates whether the noscript tag content should be rendered. If `bShowNoScript` is `true`, the content appears.

Returns

Nothing.

dom.source.arrowDown()

Availability

Dreamweaver 4

Description

Moves the insertion point down the source view document, line by line. If content is already selected, this function extends the selection line by line.

Arguments

`{nTimes}`, *`{bShiftIsDown}`*

- *`nTimes`* is the number of lines that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *`bShiftIsDown`* is a Boolean value that indicates whether content is being selected. If *`bShiftIsDown`* is `true`, the content is selected.

Returns

Nothing.

dom.source.arrowLeft()

Availability

Dreamweaver 4

Description

Moves the insertion point to the left in the current line of the Source view. If content is already selected, this function extends the selection to the left.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of characters that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.arrowRight()

Availability

Dreamweaver 4

Description

Moves the insertion point to the right in the current line of the Source view. If content is already selected, this function extends the selection to the right.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of characters that the insertion point is to move. If the `nTimes` argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.arrowUp()

Availability

Dreamweaver 4

Description

Moves the insertion point up the source view document, line by line. If content is already selected, this function extends the selection line by line.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of lines that the insertion point is to move. If the `nTimes` argument is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.balanceBracesTextview()

Availability

Dreamweaver 4

Description

Source view extension that enables parentheses balancing. You can call `dom.source.balanceBracesTextview()` to extend a currently highlighted selection or insertion point from the start of the surrounding parenthetical statement to the end of the statement to balance the following characters: `[]`, `{}` and `()`. Subsequent calls expand the selection through further levels of punctuation nesting.

Arguments

None.

Returns

Nothing.

dom.source.endOfDocument()

Availability

Dreamweaver 4

Description

Places the insertion point at the end of the current source view document. If content is already selected, this function extends the selection to the end of the document.

Arguments

bShiftIsDown

A Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.endOfLine()

Availability

Dreamweaver 4

Description

Places the insertion point at the end of the current line. If content is already selected, this function extends the selection to the end of the current line.

Arguments

bShiftIsDown

A Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.endPage()

Availability

Dreamweaver 4

Description

Moves the insertion point to the end of the current page or to the end of the next page if the insertion point is already at the end of a page. If content is already selected, this function extends the selection page by page.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.getCurrentLines()

Availability

Dreamweaver 4

Description

Returns the line numbers for the specified offset locations from the beginning of the document.

Arguments

None.

Returns

The line numbers for the current selection.

dom.source.getSelection()

Description

Gets the selection in the current document, which is expressed as character offsets into the document's Code view.

Arguments

None.

Returns

A pair of integers that represent offsets from the beginning of the source document. The first integer is the beginning of the selection; the second is the end of the selection. If the two numbers are equal, the selection is an insertion point. If there is no selection in the source, both numbers are -1.

dom.source.getLineFromOffset()

Availability

Dreamweaver MX

Description

Takes an offset into the source document.

Arguments

None.

Returns

The associated line number, or -1 if the offset is negative or past the end of the file.

dom.source.getText()

Availability

Dreamweaver 4

Description

Returns the text string in the source between the designated offsets.

Arguments

`startOffset`, *`endOffset`*

- *`start`* is an integer that represents the offset from the beginning of the document.
- *`end`* is an integer that represents the end of the document.

Returns

A string that represents the text in the source code between the offsets *`start`* and *`end`*.

dom.source.indentTextview()

Availability

Dreamweaver 4

Description

Moves selected source view text one tab stop to the right.

Arguments

None.

Returns

Nothing.

dom.source.insert()

Description

Inserts the specified string into the source code at the specified offset from the beginning of the source file. If the offset is not greater than, or equal to, zero, the insertion fails and the function returns `false`.

Arguments

`offset`, *`string`*

- *`offset`* is the offset from the beginning of the file where the string is to be inserted.
- *`string`* is the string to insert.

Returns

`true` if successful; `false` otherwise.

dom.source.nextWord()

Availability

Dreamweaver 4

Description

Moves the insertion point to the beginning of the next word (or words, if specified) in the Source view. If content is already selected, this function extends the selection to the right.

Arguments

`{nTimes}`, *`{bShiftIsDown}`*

- *`nTimes`* is the number of words that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *`bShiftIsDown`* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.outdentTextview()

Availability

Dreamweaver 4

Description

Moves selected source view text one tab stop to the left.

Arguments

None.

Returns

Nothing.

dom.source.pageDown()

Availability

Dreamweaver 4

Description

Moves the insertion point down the source view document, page by page. If content is already selected, this function extends the selection page by page.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.pageUp()

Availability

Dreamweaver 4

Description

Moves the insertion point up the source view document, page by page. If content is already selected, this function extends the selection page by page.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.previousWord()

Availability

Dreamweaver 4

Description

Moves the insertion point to the beginning of the previous word (or words, if specified) in the source view. If content is already selected, this function extends the selection to the left.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of words that the insertion point is to move. If nTimes is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If bShiftIsDown is true, the content is selected.

Returns

Nothing.

dom.source.replaceRange()

Availability

Dreamweaver 4

Description

Replaces the range of source text between *startOffset* and *endOffset* with *string*. If *startOffset* is greater than *endOffset* or if either offset is not a positive integer, it does nothing and returns false. If *endOffset* is greater than the number of characters in the file, it replaces the range between *startOffset* and the end of the file. If both *startOffset* and *endOffset* are greater than the number of characters in the file, it inserts the text at the end of the file.

Arguments

startOffset, *endOffset*, *string*

- *startOffset* is the offset that indicates the beginning of the block to replace.
- *endOffset* is the offset that indicates the end of the block to replace.
- *string* is the string to insert.

Returns

true if successful; false otherwise.

dom.source.scrollEndFile()

Availability

Dreamweaver 4

Description

Scrolls the Source view to the bottom of the document file without moving the insertion point.

Arguments

None.

Returns

Nothing.

dom.source.scrollLineDown()

Availability

Dreamweaver 4

Description

Scrolls the Source view down line by line without moving the insertion point.

Arguments

nTimes

nTimes is the number of lines to scroll. If `nTimes` is omitted, the default is 1.

Returns

Nothing.

dom.source.scrollLineUp()

Availability

Dreamweaver 4

Description

Scrolls the Source view up line by line without moving the insertion point.

Arguments

nTimes

nTimes is the number of lines to scroll. If `nTimes` is omitted, the default is 1.

Returns

Nothing.

dom.source.scrollPageDown()

Availability

Dreamweaver 4

Description

Scrolls the Source view down page by page without moving the insertion point.

Arguments

nTimes

nTimes is the number of pages to scroll. If `nTimes` is omitted, the default is 1.

Returns

Nothing.

dom.source.scrollPageUp()

Availability

Dreamweaver 4

Description

Scrolls the Source view up page by page without moving the insertion point.

Arguments

nTimes

nTimes is the number of pages to scroll. If `nTimes` is omitted, the default is 1.

Returns

Nothing.

dom.source.scrollTopFile()

Availability

Dreamweaver 4

Description

Scrolls the Source view to the top of the document file without moving the insertion point.

Arguments

None.

Returns

Nothing.

dom.source.selectParentTag()

Availability

Dreamweaver 4

Description

Source view extension that enables tag balancing. You can call `dom.source.selectParentTag()` to extend a currently highlighted selection or insertion point from the surrounding open tag to the closing tag. Subsequent calls extend the selection to additional surrounding tags until there are no more enclosing tags.

Arguments

None.

Returns

Nothing.

dom.source.setCurrentLine()

Availability

Dreamweaver 4

Description

Puts the insertion point at the beginning of the specified line. If the *`lineNumber`* argument is not a positive integer, the function does nothing and returns `false`. Puts the insertion point at the beginning of the last line if *`lineNumber`* is larger than the number of lines in the source.

Arguments

`lineNumber`

`lineNumber` is the line at the beginning of which the insertion point is placed.

Returns

`true` if successful, `false` if not

dom.source.startOfDocument()

Availability

Dreamweaver 4

Description

Places the insertion point at the beginning of the source view document. If content is already selected, this function extends the selection to the beginning of the document.

Arguments

bShiftIsDown

A Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.startOfLine()

Availability

Dreamweaver 4

Description

Places the insertion point at the beginning of the current line. If content is already selected, this function extends the selection to the beginning of the current line.

Arguments

bShiftIsDown

A Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.topPage()

Availability

Dreamweaver 4

Description

Moves the insertion point to the top of the current page or to the top of the previous page if the insertion point is already at the top of a page. If content is already selected, this function extends the selection page by page.

Arguments

{nTimes}, *{bShiftIsDown}*

- *nTimes* is the number of pages that the insertion point is to move. If `nTimes` is omitted, the default is 1.
- *bShiftIsDown* is a Boolean value that indicates whether content is being selected. If `bShiftIsDown` is `true`, the content is selected.

Returns

Nothing.

dom.source.wrapSelection()

Availability

Dreamweaver 4

Description

Inserts the text of *startTag* before the current selection and the text of *endTag* after the current selection. The function then selects the entire range between, and including, the inserted tags. If the current selection was an insertion point, then the function places the insertion point between the *startTag* and *endTag*. (*startTag* and *endTag* don't have to be tags; they can be any arbitrary text.)

Arguments

startTag, *endTag*

- *startTag* is the text to insert at the beginning of the selection.
- *endTag* is the text to insert at the end of the selection.

Returns

Nothing.

dom.synchronizeDocument()

Availability

Dreamweaver 4

Description

Synchronizes the Design and Source views.

Arguments

None.

Returns

Nothing.

Table editing functions

Table functions add and remove table rows and columns, change column widths and row heights, convert measurements from pixels to percents and back, and perform other standard table-editing tasks.

dom.convertWidthsToPercent()

Availability

Dreamweaver 3

Description

Converts all `WIDTH` attributes in the current table from pixels to percentages.

Arguments

None.

Returns

Nothing.

dom.convertWidthsToPixels()

Availability

Dreamweaver 4

Description

Converts all `WIDTH` attributes in the current table from percentages to pixels.

Arguments

None.

Returns

Nothing.

dom.decreaseColspan()

Availability

Dreamweaver 3

Description

Decreases the column span by one.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canDecreaseColspan()" on page 411

dom.decreaseRowspan()

Availability

Dreamweaver 3

Description

Decreases the row span by one

Arguments

None.

Returns

Nothing.

Enabler

"dom.canDecreaseRowspan()" on page 412

dom.deleteTableColumn()

Availability

Dreamweaver 3

Description

Removes the selected table column or columns.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canDeleteTableColumn()" on page 412

dom.deleteTableRow()

Availability

Dreamweaver 3

Description

Removes the selected table row or rows.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canDeleteTableRow()" on page 412

dom.doDeferredTableUpdate()

Availability

Dreamweaver 3

Description

If the Faster Table Editing option is selected in the General preferences, it forces the table layout to reflect recent changes without moving the selection outside the table. This function has no effect if the Faster Table Editing option is not selected.

Arguments

None.

Returns

Nothing.

dom.getTableExtent()

Availability

Dreamweaver 3

Description

Gets the number of columns and rows in the selected table.

Arguments

None.

Returns

An array that contains two whole numbers. The first array item is the number of columns, and the second array item is the number of rows. If no table is selected, nothing returns.

dom.increaseColspan()

Availability

Dreamweaver 3

Description

Increases the column span by one.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canIncreaseColspan()" on page 413

dom.increaseRowspan()

Availability

Dreamweaver 3

Description

Increases the row span by one.

Arguments

None.

Returns

Nothing.

Enabler

"dom.canDecreaseRowspan()" on page 412

dom.insertTableColumns()

Availability

Dreamweaver 3

Description

Inserts the specified number of table columns into the current table.

Arguments

numberOfCols, *bBeforeSelection*

- *numberOfCols* is the number of columns to insert.
- *bBeforeSelection* is a Boolean value that indicates whether the columns should be inserted before the column that contains the selection.

Returns

Nothing.

Enabler

"dom.canInsertTableColumns()" on page 413

dom.insertTableRows()

Availability

Dreamweaver 3

Description

Inserts the specified number of table rows into the current table.

Arguments

numberOfRows, *bBeforeSelection*

- *numberOfRows* is the number of rows to insert.
- *bBeforeSelection* is a Boolean value that indicates whether the rows should be inserted above the row that contains the selection.

Returns

Nothing.

Enabler

"dom.canInsertTableRows()" on page 414

dom.mergeTableCells()

Availability
Dreamweaver 3

Description
Merges the selected table cells.

Arguments
None.

Returns
Nothing.

Enabler
"dom.canMergeTableCells()" on page 414

dom.removeAllTableHeights()

Availability
Dreamweaver 3

Description
Removes all `HEIGHT` attributes from the selected table.

Arguments
None.

Returns
Nothing.

dom.removeAllTableWidths()

Availability
Dreamweaver 3

Description
Removes all `WIDTH` attributes from the selected table.

Arguments
None.

Returns
Nothing.

dom.setTableCellTag()

Availability
Dreamweaver 3

Description
Specifies the tag for the selected cell.

Arguments

tdOrTh

tdOrTh must be either `"td"` or `"th"`.

Returns

Nothing.

dom.setTableColumns()

Availability

Dreamweaver 3

Description

Sets the number of columns in the selected table.

Arguments

numberOfCols

Returns

Nothing.

dom.setTableRows()

Availability

Dreamweaver 3

Description

Sets the number of rows in the selected table.

Arguments

numberOfRows

Returns

Nothing.

dom.showInsertTableRowsOrColumnsDialog()

Availability

Dreamweaver 3

Description

Opens the Insert Rows or Columns dialog box.

Arguments

None.

Returns

Nothing.

Enabler

“dom.canInsertTableColumns()” on page 413 or “dom.canInsertTableRows()” on page 414

dom.splitTableCell()

Availability

Dreamweaver 3

Description

Splits the current table cell into the specified number of rows or columns. If one or both of the arguments is omitted, the Split Cells dialog box appears.

Arguments

{colsOrRows}, *{numberToSplitInto}*

- *colsOrRows*, if supplied, must be either `"columns"` or `"rows"`.
- *numberToSplitInto*, if supplied, is the number of rows or columns into which the cell will be split.

Returns

Nothing.

Enabler

"dom.canSplitTableCell()" on page 417

Tag editor and tag library functions

You can use tag editors to insert new tags, edit existing tags, access reference information about tags. The Tag Chooser lets users organize their tags so that they can easily select tags that they use frequently. The Tag Libraries that come with Dreamweaver store information about tags that are used in standards-based markup languages and most widely used, tag-based scripting languages. You can use the JavaScript tag editor, Tag Chooser, and Tag Library functions when you need to access and work with tag editors and Tag Libraries in your extensions.

dom.getTagSelectorTag()

Availability

Dreamweaver MX

Description

Gets the DOM node for the tag that is currently selected in the tag selector bar at the bottom of the document window.

Arguments

None.

Returns

The DOM node for the currently selected tag, `null` if no tag is selected.

dreamweaver.popupInsertTagDialog()

Availability

Dreamweaver MX

Description

Checks the .vtm files to see if a tag editor has been defined for the tag. If so, the editor for that tag pops up and accepts the start tag. If not, the start tag is inserted unmodified into the user's document.

Arguments

A start tag string that includes one of the following types of initial values:

- A tag, an in `<input>`
- A tag with attributes, as in `<input type='text'>`
- A directive, as in `<%= %>`

Returns

A Boolean value: `true` if anything is inserted into the document; `false` otherwise.

dreamweaver.popupEditTagDialog()

Availability

Dreamweaver MX

Description

If a tag is selected, the tag editor for that tag opens, so you can edit the tag.

Arguments

None.

Returns

Nothing.

Enabler

`dw.canPopupEditTagDialog()`

dreamweaver.showTagChooser()

Availability

Dreamweaver MX

Description

Displays the Tag Chooser dialog box, brings it to the front, and sets focus.

Arguments

None.

Returns

Nothing.

dreamweaver.showTagLibraryEditor()

Availability

Dreamweaver MX

Description

Opens the Tag Library editor.

Arguments

None.

Returns

None.

dreamweaver.tagLibrary.getTagLibraryDOM()

Availability

Dreamweaver MX

Description

Given the URL of a *filename*.vtm file, this function returns the DOM for that file, so that its contents can be edited. This function should only be called when the Tag Library editor is active.

Arguments

The URL of a *filename*.vtm file, relative to the Configuration/Tag Libraries folder, as in the following example:

```
"HTML/img.vtm"
```

Returns

DOM pointer to a new or previously existing file within the Tag Library folder.

dreamweaver.tagLibrary.getSelectedLibrary()

Availability

Dreamweaver MX

Description

If a library node is selected in the Tag Library editor, this function gets the library name.

Arguments

None.

Returns

A string, the name of the library that is currently selected in the Tag Library editor; returns an empty string if no library is selected.

dreamweaver.tagLibrary.getSelectedTag()

Availability

Dreamweaver MX

Description

If an attribute node is currently selected, gets the name of the tag that contains the attribute.

Arguments

None.

Returns

A string, name of the tag that is currently selected in the Tag Library editor; returns an empty string if no tag is selected.

dreamweaver.tagLibrary.importDTDOrSchema()

Availability

Dreamweaver MX

Description

Imports a DTD or schema file from a remote server into the Tag Library.

Arguments

File URL: Path to DTD or schema file, in local URL format.

Prefix: The prefix string that should be added to all tags in this tag library.

Returns

Name of the imported tag library.

dreamweaver.tagLibrary.getImportedTagList()

Availability

Dreamweaver MX

Description

Generates a list of `TagInfo` objects from an imported tag library.

Arguments

Name of imported tag library.

Returns

Array of `tagInfo` objects.

A `taginfo` object contains information about a single tag that is included in the tag library. The following properties are defined in a `tagInfo` object:

- `tagName`: a string
- `attributes`: an array of strings. Each string is the name of an attribute that is defined for this tag.

Example:

```
// "fileURL" and "prefix" have been entered by the user.
// tell the Tag Library to Import the DTD/Schema
var libName = dw.tagLibrary.importDTDOrSchema(fileURL, prefix);

// get the array of tags for this library
// this is the TagInfo object
var tagArray = dw.tagLibrary.getImportedTagList(libName);

// now I have an array of tagInfo objects.
// I can get info out of them. This gets info out of the first one.
// note: this assumes there is at least one TagInfo in the array.
var firstTagName = tagArray[0].name;
var firstTagAttributes = tagArray[0].attributes;
// note that firstTagAttributes is an array of attributes.
```

Tag inspector functions

These JavaScript functions manipulate the generic Tag inspector panel, specifically the context menus in the Tag inspector panel. They are useful when creating extensions that incorporate new context menus for the Tag inspector.

dreamweaver.tagInspector.tagBefore()

Availability

Dreamweaver MX

Description

Inserts a new tag before the currently selected tag. The new tag is either be empty, non-empty or it accepts a tag name that is passed in as an argument.

Arguments

string: *MM:non-empty*, *MM:empty* or *{tagname}*

Returns

Nothing.

Enabler

"dreamweaver.tagInspector.tagBeforeEnabled()" on page 429

dreamweaver.tagInspector.tagInside()

Availability

Dreamweaver MX

Description

Inserts a new tag inside the currently selected tag as its first child. The tag will either be empty, non-empty, or the tag name that is passed in as an argument.

Arguments

String: *MM:non-empty*, *MM:empty* or *{tagname}*

Returns

Nothing.

Enabler

"dreamweaver.tagInspector.tagInsideEnabled()" on page 429

dreamweaver.tagInspector.tagAfter()

Availability

Dreamweaver MX

Description

Inserts a new tag after the selected tag. The tag will either be empty, non-empty, or the tag name that is passed in as an argument.

Arguments

string: *MM:non-empty*, *MM:empty* or *{tagname}*

Returns

Nothing.

Enabler

"dreamweaver.tagInspector.tagAfterEnabled()" on page 430

dreamweaver.tagInspector.deleteTags()

Availability

Dreamweaver MX

Description

Deletes the currently selected tags.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.tagInspector.deleteTagsEnabled()" on page 430

dreamweaver.tagInspector.editTagName()

Availability

Dreamweaver MX

Description

Puts an text box around the selected tag so that the user can enter a new tag name. The function performs no validation—Dreamweaver simply replaces the selected tag with the new name.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.tagInspector.editTagNameEnabled()" on page 430

Timeline functions

Timeline functions act on timelines. They add, remove, and change objects in a timeline; add behaviors, frames, and keyframes to a timeline; specify whether the timeline should play and loop automatically; and more. All the functions in this section are methods of the `dreamweaver.timelineInspector` object because they affect the contents of the Timelines panel.

dreamweaver.timelineInspector.addBehavior()

Availability

Dreamweaver 3

Description

Opens the Behaviors panel and automatically supplies the correct *`onFrameN`* event (where *`N`* is the frame that is marked by the playback head) when the user chooses an action and clicks OK.

Arguments

None.

Returns

Nothing.

dreamweaver.timelineInspector.addFrame()

Availability

Dreamweaver 3

Description

Adds a frame to the current timeline at the frame that contains the playback head.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.timelineInspector.canAddFrame()" on page 430

dreamweaver.timelineInspector.addKeyframe()

Availability

Dreamweaver 3

Description

Adds a keyframe to the selected animation bar at the frame that contains the playback head.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.timelineInspector.canAddKeyFrame()" on page 431

dreamweaver.timelineInspector.addObject()

Availability

Dreamweaver 3

Description

Adds the currently selected object to the timeline.

Arguments

None.

Returns

Nothing.

dreamweaver.timelineInspector.addTimeline()

Availability

Dreamweaver 3

Description

Adds a new timeline to the current document.

Arguments

None.

Returns

Nothing.

dreamweaver.timelineInspector.changeObject()

Availability

Dreamweaver 3

Description

Opens the Change Object dialog box.

Arguments

None.

Returns

Nothing.

Enabler

"dreamweaver.timelineInspector.canChangeObject()" on page 431

dreamweaver.timelineInspector.getAutoplay()

Availability

Dreamweaver 3

Description

Gets the state of the Autoplay option for the current timeline.

Arguments

None.

Returns

A Boolean value that indicates whether the Autoplay option is selected.

dreamweaver.timelineInspector.getCurrentFrame()

Availability

Dreamweaver 3

Description

Gets the current frame of the current timeline.

Arguments

None.

Returns

A frame number.

dreamweaver.timelineInspector.getLoop()

Availability

Dreamweaver 3

Description

Gets the state of the Loop option for the current timeline.

Arguments

None.

Returns

A Boolean value that indicates whether the Loop option is selected.

dreamweaver.timelineInspector.recordPathOfLayer()

Availability

Dreamweaver 3

Description

Records the path of a layer as the user drags it.

Arguments

None.

Returns

Nothing.

dreamweaver.timelineInspector.removeBehavior()

Availability
Dreamweaver 3

Description
Removes the selected behavior from the timeline.

Arguments
None.

Returns
Nothing.

Enabler
"dreamweaver.timelineInspector.canRemoveBehavior()" on page 431

dreamweaver.timelineInspector.removeFrame()

Availability
Dreamweaver 3

Description
Removes the selected frame from the timeline.

Arguments
None.

Returns
Nothing.

Enabler
"dreamweaver.timelineInspector.canRemoveFrame()" on page 432

dreamweaver.timelineInspector.removeKeyframe()

Availability
Dreamweaver 3

Description
Removes the selected keyframe from an animation bar.

Arguments
None.

Returns
Nothing.

Enabler
"dreamweaver.timelineInspector.canRemoveKeyFrame()" on page 432

dreamweaver.timelineInspector.removeObject()

Availability
Dreamweaver 3

Description
Removes the currently selected object from the timeline.

Arguments
None.

Returns
Nothing.

Enabler
"dreamweaver.timelineInspector.canRemoveObject()" on page 432

dreamweaver.timelineInspector.removeTimeline()

Availability
Dreamweaver 3

Description
Removes the current timeline from the document.

Arguments
None.

Returns
Nothing.

dreamweaver.timelineInspector.renameTimeline()

Availability
Dreamweaver 3

Description
Opens the Rename Timeline dialog box for the current timeline.

Arguments
None.

Returns
Nothing.

dreamweaver.timelineInspector.setAutoplay()

Availability
Dreamweaver 3

Description
Sets the Autoplay option for the current timeline.

Arguments

bAutoplay

bAutoplay is a Boolean value that indicates whether to turn on the Autoplay option.

Returns

Nothing.

dreamweaver.timelineInspector.setCurrentFrame()

Availability

Dreamweaver 3

Description

Moves the playback head to the specified frame.

Arguments

frameNumber

Returns

Nothing.

dreamweaver.timelineInspector.setLoop()

Availability

Dreamweaver 3

Description

Sets the Loop option for the current timeline.

Arguments

bLoop

bLoop is a Boolean value that indicates whether to turn on the Loop option.

Returns

Nothing.

Toggle functions

Toggle functions get and set various options either on or off.

dom.getEditNoFramesContent()

Availability

Dreamweaver 3

Description

Gets the current state of the Modify > Frameset > Edit NoFrames Content option.

Arguments

None.

Returns

A Boolean value that indicates whether the `NOFRAMES` content is the active view (`true`) or not (`false`).

dom.getHideAllVisualAids()

Availability

Dreamweaver 4

Description

Determines whether visual aids are set as hidden.

Arguments

None.

Returns

A Boolean value that is `true` if Hide All Visual Aids is set; `false` otherwise.

dom.getPreventLayerOverlaps()

Availability

Dreamweaver 3

Description

Gets the current state of the Prevent Layer Overlaps option.

Arguments

None.

Returns

A Boolean value that indicates whether the option is on (`true`) or off (`false`).

dom.getShowAutoIndent()

Availability

Dreamweaver 4

Description

Determines whether auto-indenting is on in the Code view of the Document window.

Arguments

None.

Returns

Returns `true` if auto-indenting is on.

dom.getShowFrameBorders()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Frame Borders option.

Arguments

None.

Returns

A Boolean value that indicates whether frame borders are visible (`true`) or not (`false`).

dom.getShowGrid()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Grid > Show option.

Arguments

None.

Returns

A Boolean value that indicates whether the grid is visible (`true`) or not (`false`).

dom.getShowHeadView()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Head Content option.

Arguments

None.

Returns

A Boolean value that indicates whether head content is visible (`true`) or not (`false`).

dom.getShowInvalidHTML()

Availability

Dreamweaver 4

Description

Determines whether invalid HTML code is currently highlighted in the Code view of the Document window.

Arguments

None.

Returns

Returns `true` if invalid HTML code is being highlighted.

dom.getShowImageMaps()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Image Maps option.

Arguments

None.

Returns

A Boolean value that indicates whether image maps are visible (`true`) or not (`false`).

dom.getShowInvisibleElements()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Invisible Elements option.

Arguments

None.

Returns

A Boolean value that indicates whether invisible element markers are visible (`true`) or not (`false`).

dom.getShowLayerBorders()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Layer Borders option.

Arguments

None.

Returns

A Boolean value that indicates whether layer borders are visible (`true`) or not (`false`).

dom.getShowLineNumbers()

Availability

Dreamweaver 4

Description

Determines whether line numbers are shown in the Code view.

Arguments

None.

Returns

Returns a Boolean value that indicates whether line numbers are shown (`true`) or not (`false`).

dom.getShowRulers()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Rulers > Show option.

Arguments

None.

Returns

A Boolean value that indicates whether the rulers are visible (`true`) or not (`false`).

dom.getShowSyntaxColoring()

Availability

Dreamweaver 4

Description

Determines whether syntax coloring is on in the Code view of the Document window.

Arguments

None.

Returns

Returns `true` if syntax coloring is on.

dom.getShowTableBorders()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Table Borders option.

Arguments

None.

Returns

A Boolean value that indicates whether table borders are visible (`true`) or not (`false`).

dom.getShowToolbar()

Availability

Dreamweaver 4

Description

Determines whether the toolbar is displayed.

Arguments

None.

Returns

Returns `true` if the toolbar is displayed; `false` otherwise.

dom.getShowTracingImage()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Tracing Image > Show option.

Arguments

None.

Returns

A Boolean value that indicates whether the option is on (`true`) or off (`false`).

dom.getShowWordWrap()

Availability

Dreamweaver 4

Description

Determines whether word wrap is on in the Code view of the Document window.

Arguments

None.

Returns

Returns `true` if word wrap is on.

dom.getSnapToGrid()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Grid > Snap To option.

Arguments

None.

Returns

A Boolean value that indicates whether grid snapping is on (`true`) or off (`false`).

dom.setEditNoFramesContent()

Availability

Dreamweaver 3

Description

Turns the Modify > Frameset > Edit NoFrames Content option on (`true`) or off (`false`).

Arguments

`bEditNoFrames`

Returns

Nothing.

Enabler

"dom.canEditNoFramesContent()" on page 412

dom.setHideAllVisualAids()

Availability

Dreamweaver 4

Description

Turns off the display of all borders, image maps, and invisible elements, regardless of their individual settings in the View menu.

Arguments

`bSet`

`bSet` is a Boolean value; when set to `false`, the previous settings are restored.

Returns

Nothing.

dom.setPreventLayerOverlaps()

Availability

Dreamweaver 3

Description

Turns the Prevent Layer Overlaps option on (`true`) or off (`false`).

Arguments

`bPreventLayerOverlaps`

Returns

Nothing.

dom.setShowFrameBorders()

Availability

Dreamweaver 3

Description

Turns the View > Frame Borders option on (`true`) or off (`false`).

Arguments

`bShowFrameBorders`

Returns

Nothing.

dom.setShowGrid()

Availability

Dreamweaver 3

Description

Turns the View > Grid > Show option on (`true`) or off (`false`).

Arguments

`bShowGrid`

Returns

Nothing.

dom.setShowHeadView()

Availability

Dreamweaver 3

Description

Turns the View > Head Content option on (`true`) or off (`false`).

Arguments

`bShowHead`

Returns

Nothing.

dom.setShowInvalidHTML()

Availability

Dreamweaver 4

Description

Turns highlighting of invalid HTML code on or off in the Code view of the Document window.

Arguments

bShow

bShow is a Boolean value that indicates whether the highlighting of invalid HTML code should be visible (`true`) or not (`false`).

Returns

Nothing.

dom.setShowImageMaps()

Availability

Dreamweaver 3

Description

Turns the View > Image Maps option on (`true`) or off (`false`).

Arguments

bShowImageMaps

Returns

Nothing.

dom.setShowInvisibleElements()

Availability

Dreamweaver 3

Description

Turns the View > Invisible Elements option on (`true`) or off (`false`).

Arguments

bViewInvisibleElements

Returns

Nothing.

dom.setShowLayerBorders()

Availability

Dreamweaver 3

Description

Turns the View > Layer Borders option on (`true`) or off (`false`).

Arguments

bShowLayerBorders

Returns

Nothing.

dom.setShowLineNumbers()

Availability

Dreamweaver 4

Description

Shows or hides the line numbers in the Code view of the Document window.

Arguments

bShow

bShow is a Boolean value that indicates whether the line numbers should be visible (`true`) or not (`false`).

Returns

Nothing.

dom.setShowRulers()

Availability

Dreamweaver 3

Description

Turns the View >Rulers > Show option on (`true`) or off (`false`).

Arguments

bShowRulers

Returns

Nothing.

dom.setShowSyntaxColoring()

Availability

Dreamweaver 4

Description

Turns syntax coloring on or off in the Code view of the Document window.

Arguments

bShow

bShow is a Boolean value that indicates whether the syntax coloring should be visible (`true`) or not (`false`).

Returns

Nothing.

dom.setShowTableBorders()

Availability

Dreamweaver 3

Description

Turns the View > Table Borders option on (`true`) or off (`false`).

Arguments

`bShowTableBorders`

Returns

Nothing.

dom.setShowToolbar()

Availability

Dreamweaver 4

Description

Shows or hides the Toolbar.

Arguments

`bShow`

`bShow` is a Boolean value that indicates whether the toolbar should be visible (`true`) or not (`false`).

Returns

Nothing.

dom.setShowTracingImage()

Availability

Dreamweaver 3

Description

Turns the View > Tracing Image > Show option on (`true`) or off (`false`).

Arguments

`bShowTracingImage`

Returns

Nothing.

dom.setShowWordWrap()

Availability

Dreamweaver 4

Description

Turns word wrap off or on in the Code view of the Document window.

Arguments

bShow

bShow is a Boolean value that indicates whether the line numbers should be visible (`true`) or not (`false`).

Returns

Nothing.

dom.setSnapToGrid()

Availability

Dreamweaver 3

Description

Turns the View > Grid > Snap To option on (`true`) or off (`false`).

Arguments

bSnapToGrid

Returns

Nothing.

dreamweaver.getHideAllFloaters()

Availability

Dreamweaver 3

Description

Gets the current state of the Hide Floating panels option.

Arguments

None.

Returns

A Boolean value that indicates whether the Hide Floating panels option (`true`) or the Show Floating panels option (`false`) is available.

dreamweaver.getShowStatusBar()

Availability

Dreamweaver 3

Description

Gets the current state of the View > Status Bar option.

Arguments

None.

Returns

A Boolean value that indicates whether the status bar is visible (`true`) or not (`false`).

dreamweaver.htmlInspector.getShowAutoIndent()

Availability

Dreamweaver 4

Description

Determines whether auto-indenting is on in the Code view of the Code inspector.

Arguments

None.

Returns

Returns `true` if auto-indenting is on.

dreamweaver.htmlInspector.getShowInvalidHTML()

Availability

Dreamweaver 4

Description

Determines whether invalid HTML code is currently highlighted in the Code view of the Code inspector.

Arguments

None.

Returns

Returns `true` if invalid HTML code is currently highlighted.

dreamweaver.htmlInspector.getShowLineNumbers()

Availability

Dreamweaver 4

Description

Determines whether line numbers are being shown in the Code view of the Code inspector.

Arguments

None.

Returns

Returns `true` if line numbers are shown.

dreamweaver.htmlInspector.getShowSyntaxColoring()

Availability

Dreamweaver 4

Description

Determines whether syntax coloring is on in the Code view of the Code inspector.

Arguments

None.

Returns

Returns `true` if syntax coloring is on.

dreamweaver.htmlInspector.getShowWordWrap()

Availability

Dreamweaver 4

Description

Determines whether word wrap is on in the Code view of the Code inspector.

Arguments

None.

Returns

Returns `true` if word wrap is on.

dreamweaver.htmlInspector.setShowAutoIndent()

Availability

Dreamweaver 4

Description

Turns auto-indenting on or off in the Code view of the Code inspector.

Arguments

`bShow`

`bShow` is a Boolean value that indicates whether the auto-indenting should be on (`true`) or off (`false`).

Returns

Nothing.

dreamweaver.htmlInspector.setShowInvalidHTML()

Availability

Dreamweaver 4

Description

Turns highlighting of invalid HTML code on or off in the Code view of the Code inspector.

Arguments

`bShow`

`bShow` is a Boolean value that indicates whether the highlighting of invalid HTML code should be visible (`true`) or not (`false`).

Returns

Nothing.

dreamweaver.htmlInspector.setShowLineNumbers()

Availability

Dreamweaver 4

Description

Shows or hides the line numbers in the Code view of the Code inspector.

Arguments

bShow

bShow is a Boolean value that indicates whether the line numbers should be visible (`true`) or not (`false`).

Returns

Nothing.

dreamweaver.htmlInspector.setShowSyntaxColoring()

Availability

Dreamweaver 4

Description

Turns syntax coloring on or off in the Code view of the Code inspector.

Arguments

bShow

bShow is a Boolean value that indicates whether the syntax coloring should be visible (`true`) or not (`false`).

Returns

Nothing.

dreamweaver.htmlInspector.setShowWordWrap()

Availability

Dreamweaver 4

Description

Turns word wrap off or on in the Code view of the Code inspector.

Arguments

bShow

bShow is a Boolean value that indicates whether the word wrapping should be on (`true`) or off (`false`).

Returns

Nothing.

dreamweaver.setHideAllFloaters()

Availability

Dreamweaver 3

Description

Turns on either the Hide Floating panels option (`true`) or the Show Floating panels option (`false`).

Arguments

`bShowFloatingPalettes`

Returns

Nothing.

dreamweaver.setShowStatusBar()

Availability

Dreamweaver 3

Description

Turns the View > Status Bar option on (`true`) or off (`false`).

Arguments

`bShowStatusBar`

Returns

Nothing.

site.getShowDependents()

Availability

Dreamweaver 3

Description

Gets the current state of the Show Dependent Files option.

Arguments

None.

Returns

A Boolean value that indicates whether dependent files are visible in the site map (`true`) or not (`false`).

site.getShowHiddenFiles()

Availability

Dreamweaver 3

Description

Gets the current state of the Show Files Marked as Hidden option.

Arguments

None.

Returns

A Boolean value that indicates whether hidden files are visible in the site map (`true`) or not (`false`).

site.getShowPageTitles()

Availability

Dreamweaver 3

Description

Gets the current state of the Show Page Titles option.

Arguments

None.

Returns

A Boolean value that indicates whether page titles are visible in the site map (`true`) or not (`false`).

site.getShowToolTips()

Availability

Dreamweaver 3

Description

Gets the current state of the Tool Tips option.

Arguments

None.

Returns

A Boolean value that indicates whether tool tips are visible in the Site panel (`true`) or not (`false`).

site.setShowDependents()

Availability

Dreamweaver 3

Description

Turns the Show Dependent Files option in the site map on (`true`) or off (`false`).

Arguments

`bShowDependentFiles`

Returns

Nothing.

site.setShowHiddenFiles()

Availability

Dreamweaver 3

Description

Turns the Show Files Marked as Hidden option in the site map on (`true`) or off (`false`).

Arguments

`bShowHiddenFiles`

Returns

Nothing.

site.setShowPageTitles()

Availability

Dreamweaver 3

Description

Turns the Show Page Titles option in the site map on (`true`) or off (`false`).

Arguments

`bShowPageTitles`

Returns

Nothing.

Enabler

"site.canShowPageTitles()" on page 439

site.setShowToolTips()

Availability

Dreamweaver 3

Description

Turns the Tool Tips option on (`true`) or off (`false`).

Arguments

`bShowToolTips`

Returns

Nothing.

Toolbar functions

The following JavaScript functions let you get and set the visibility of toolbars and toolbar labels, obtain the labels of toolbar items in the current window, position toolbars, and obtain toolbar IDs. For more information on creating or modifying toolbars, see "Toolbars" on page 77.

dom.getToolbarVisibility()

Availability

Dreamweaver MX

Description

Returns a Boolean value that indicates whether the toolbar that is specified by *toolbar_id* is visible in the document window or the Dreamweaver MX workspace frame. If the toolbar is docked to the Dreamweaver MX workspace frame, this function affects that toolbar, regardless of the DOM on which it is called. If the toolbar is docked to individual document windows, the function affects the toolbar in the given document.

Arguments

toolbar_id is the ID string that is assigned to the toolbar.

Returns

`true` if the toolbar is visible in the front document window or the Dreamweaver MX workspace frame; `false` if the toolbar is not visible or does not exist.

Example

```
var retval = dom.getToolbarVisibility("myEditbar");
return retval;
```

dom.setToolbarVisibility()

Availability

Dreamweaver MX

Description

Shows or hides the specified toolbar. If the toolbar is docked to the Dreamweaver MX workspace frame because the `container` attribute is set to `mainframe`, this function affects that toolbar, regardless of the DOM on which it is called. If the toolbar is docked to individual document windows, the function affects the toolbar in the given document.

Arguments

toolbar_id, bShow

- *toolbar_id* is the ID of the toolbar, the value of the ID attribute on the toolbar tag in the toolbars.xml file.
- *bShow* is a Boolean value that indicates whether to show or hide the toolbar. If *bshow* is `true`, `dom.setToolbarVisibility()` makes the toolbar visible. If *bShow* is `false`, `dom.setToolbarVisibility()` makes the toolbar invisible.

Returns

Nothing.

Example

```
var dom = dw.getDocumentDOM();
if(dom != null && dom.getToolbarVisibility("myEditbar") == false)
{
  dom.setToolbarVisibility("myEditbar", true);
{
```

dom.setToolbarPosition()

Availability

Dreamweaver MX

Description

Moves the specified toolbar to the specified position.

Note: There is no way to determine the current position of a toolbar.

Arguments

toobar_id, position, relative_to

- *toolbar_id* is the ID of the toolbar, which is the value of the ID attribute on the toolbar tag in the toolbars.xml file.
- *position* specifies where Dreamweaver positions the toolbar, relative to other toolbars. The possible values for *position* are described in the following list:

  top is the default position. The toolbar appears at the top of the document window.

  below causes the toolbar to appear at the beginning of the row immediately below the toolbar that *relative_to* specifies. Dreamweaver reports an error if the toolbar does not find the toolbar that *relative_to* specifies.

  floating causes the toolbar to float above the document. Dreamweaver automatically places the toolbar so it is offset from other floating toolbars. On the Macintosh, floating is treated the same way as top.

- *relative_to*="*toolbar_id*" This argument is required if *position* specifies below. Otherwise, it is ignored. Specifies the ID of the toolbar below which this toolbar should be positioned.

Returns

Nothing.

Example

```
dom.setToolbarPosition("myEditbar", "below", "myPicturebar");
```

dom.getToolbarIdArray()

Availability

Dreamweaver MX

Description

Returns an array of the IDs of all the toolbars in the application. You can use dom.getToolbarIdArray() to turn off all toolbars so you can reposition them and make only a specific set visible.

Arguments

None.

Returns

An array of all toolbar IDs.

Example

```
var tb_ids = new Array();
tb_ids = dom.getToolbarIdArray();
```

dom.getToolbarLabel()

Availability

Dreamweaver MX

Description

Obtains the label of the specified toolbar. You can use `dom.getToolbarLabel()` for menus that show or hide toolbars.

Arguments

`toolbar_id` is the ID of the toolbar, which is the value of the ID attribute on the toolbar tag in the toolbars.xml file.

Returns

`label` which is a name string that is assigned as an attribute on the `toolbar` tag.

Example

```
var label = dom.getToolbarLabel("myEditbar");
```

dom.setShowToolbarIconLabels()

Availability

Dreamweaver MX

Description

Tells Dreamweaver to show the labels of buttons that have labels. In the Dreamweaver 4 workspace, this function operates on the document that is specified by the DOM. In the Dreamweaver MX workspace, there is only one set of toolbars, so the function operates on that set.

Dreamweaver always shows labels for nonbutton controls, if the labels are defined.

Arguments

`bShow` is a Boolean value that indicates whether to show or hide labels for buttons.

Returns

Nothing.

Example

```
dom.setShowToolbarIconLabels(true);
```

dom.getShowToolbarIconLabels()

Availability

Dreamweaver MX

Description

Determines whether labels for buttons are visible in the current document window. In the Dreamweaver 4 workspace, this function operates on the toolbars of the document that is specified by the DOM. In the Dreamweaver MX workspace, there is only one set of toolbars so the function operates on that set.

Dreamweaver always shows labels for nonbutton controls, if the labels are defined.

Arguments

None.

Returns

`true` if labels for buttons are visible in the current document window; `false` if labels for buttons are not visible in the current document window.

Example

```
var dom = dw.getDocumentDom();
if (dom.getShowToolbarIconLabels())
{
  dom.setShowToolbarIconLabels(true);
}
```

Translation functions

Translation functions deal either directly with translators or with the results of translation. These functions get information about or run a translator, edit content in a locked region, and specify that the translated source should be used when getting and setting selection offsets.

dom.runTranslator()

Availability

Dreamweaver 3

Description

Runs the specified translator on the document. This function is valid only for the active document.

Arguments

`translatorName`

`translatorName` is the name of a translator as it appears in the Translation preferences.

Returns

Nothing.

dreamweaver.editLockedRegions()

Availability

Dreamweaver 2

Description

Depending on the value of the argument, this function makes locked regions editable or noneditable. By default, locked regions are noneditable; if you try to edit a locked region before specifically making it editable with this function, Dreamweaver beeps and does not allow the change.

Note: Editing locked regions can have unintended consequences for library items and templates. You should not use this function outside the context of data translators.

Arguments

bAllowEdits

bAllowEdits is a Boolean value that indicates that edits are allowed (`true`) or not allowed (`false`). Dreamweaver automatically restores locked regions to their default (noneditable) state when the script that calls this function finishes executing.

Returns

Nothing.

dreamweaver.getTranslatorList()

Availability

Dreamweaver 3

Description

Gets a list of the installed translators.

Arguments

None.

Returns

An array of strings where each string represents the name of a translator as it appears in the Translation preferences.

dreamweaver.useTranslatedSource()

Availability

Dreamweaver 2

Description

Specifies that the values that `dom.nodeToOffsets()` and `dom.getSelection()` return. These are used by `dom.offsetsToNode()` and `dom.setSelection()` and should be offsets into the translated source (the HTML that is contained in the DOM after a translator runs), not the untranslated source.

Note: This function is relevant only in Property inspector files.

Arguments

bUseTranslatedSource

The default value of the argument is `false`. Dreamweaver automatically uses the untranslated source for subsequent calls to `dw.getSelection()`, `dw.setSelection()`, `dw.nodeToOffsets()`, and `dw.offsetsToNode()` when the script that calls `dw.useTranslatedSource()` finishes executing, if `dw.useTranslatedSource()` is not explicitly called with an argument of `false` before then.

Returns

Nothing.

Window functions

Window functions handle operations that are related to the Document window and the floating panels. The window functions show and hide floating panels, determine which part of the Document window has focus, and set the active document. For operations that are related specifically to the Site panel, see"Site functions" on page 558.

Macromedia Dreamweaver MX introduces a new user interface, known as the multiple document interface (MDI). This interface, or type of workspace, is optional but it is also the default workspace. In the multiple document interface, Dreamweaver MX integrates all the documents into one parent container in which you can dock all objects and panels. If you prefer, you can choose to work in the Dreamweaver 4 workspace, in which you manage separate, floating windows. The Dreamweaver 4 workspace is also called the classic workspace. You can switch from one type of workspace to the other through Dreamweaver MX Preferences

Note: Some of the functions in this section operate only in MDI mode and only on the Windows operating system. The description of the function indicates whether this is the case.

dom.getFocus()

Availability

Dreamweaver 3

Description

Determines the part of the document that is currently in focus.

Arguments

None.

Returns

One of the following strings:

- `"head"` if the `HEAD` area is active
- `"body"` if the `BODY` or `NOFRAMES` area is active
- `"frameset"` if a frameset or any of its frames is selected
- `"none"` if the focus is not in the document (for example, if it's in the Property inspector or another floating panel)

dom.getView()

Availability

Dreamweaver 4

Description

Determines which view is visible.

Arguments

None.

Returns

`"design"`, `"code"`, or `"split"`, depending on the visible view.

dom.getWindowTitle()

Availability

Dreamweaver 3

Description

Gets the title of the window that contains the document.

Arguments

None.

Returns

A string that contains the text that appears between the `TITLE` tags in the document, or nothing, if the document is not in an open window.

dom.setView()

Availability

Dreamweaver 4

Description

Shows or hides the Design or Code view to produce a design-only, code-only, or split view.

Arguments

`viewString`

`viewString` is the view to produce; it must be one of the following values: `design"`, `"code"`, or `"split"`.

Returns

Nothing.

dreamweaver.cascade()

Availability

Dreamweaver MX (Windows only)

Description

In MDI mode `dw.cascade()` cascades the document windows, starting in the upper left corner and positioning each window below and slightly offset from the previous one. This function works only if Dreamweaver is in MDI mode.

Arguments

None.

Returns

Nothing.

Example

```
if(dw.isMDI())
{
     dw.cascade()
}
```

dreamweaver.getActiveWindow()

Availability

Dreamweaver 3

Description

Gets the document in the active window.

Arguments

None.

Returns

The document object that corresponds to the document in the active window; or, if the document is in a frame, the document object that corresponds to the frameset.

dreamweaver.getDocumentList()

Availability

Dreamweaver 3

Description

Gets a list of all the open documents.

Arguments

None.

Returns

An array of document objects, each corresponding to an open Document window. If a Document window contains a frameset, the document object refers to the frameset, not the contents of the frames.

dreamweaver.getFloaterVisibility()

Availability

Dreamweaver 3

Description

Checks whether the specified panel or inspector is visible.

Arguments

floaterName

floaterName is the name of a floating panel. If *floaterName* does not match one of the built-in panel names, Dreamweaver searches in the Configuration/Floaters folder for a file called *floaterName*.htm where *floaterName* is the name of a floating panel.

The *floaterName* values for built-in Dreamweaver panels are the strings to the right of the panel names in the following list:

Assets = "assets"
Answers = "answers"
Behaviors = "behaviors"
Code inspector = "html"
Components = "server components"
CSS Styles = "css styles"
Databases = "databases"
Bindings = "data bindings"
Frames = "frames"
FTP Log = "ftplog"
History = "history"
HTML Styles = "html styles"
Insert bar = "objects"
Layers = "layers"
Link Checker Results = "linkchecker"
Properties = "properties"
Reference = "reference"
Report Results = "reports"
Search Results = "search"
Server Behaviors = "server behaviors"
Server Debug = "debug"
Site = "site files"
Sitespring = "sitespring"
Snippets = "snippets"
Tag inspector = "tag inspector"
Target Browser Check Results ="btc"
Timelines = "timelines"
Validation Results = "validation"

Returns

`true` if the floating panel is visible and in the front; `false` otherwise or if Dreamweaver cannot find a floating panel named *floaterName*.

dreamweaver.getFocus()

Availability

Dreamweaver 4

Description

Determines what part of the application is currently in focus.

Arguments

`bAllowFloaters`

Returns

One of the following strings:

- `"document"` if the Document window is in focus
- `"site"` if the Site panel is in focus

- `"textView"` if the Text view is in focus
- `"html"` if the Code inspector is in focus
- `floaterName`, if `bAllowFloaters` is `true` and a floating panel has focus, where `floaterName` is `"objects"`, `"properties"`, `"launcher"`, `"library"`, `"css styles"`, `"html styles"`, `"behaviors"`, `"timelines"`, `"layers"`, `"frames"`, `"templates"`, or `"history"`
- (Macintosh) `"none"` if neither the Site panel nor any Document windows are open

dreamweaver.getPrimaryView()

Availability

Dreamweaver 4

Description

Determines which view is visible as the primary (on top) view.

Arguments

None.

Returns

`"design"` or `"code"`, depending on which view is visible or on the top in a split view.

dreamweaver.getSnapDistance()

Availability

Dreamweaver 4

Description

Returns the snapping distance in pixels.

Arguments

None.

Returns

An integer that represents the snapping distance in pixels. The default is 10 pixels; 0 indicates that the Snap feature is off.

dreamweaver.isMDI()

Availability

Dreamweaver MX (Windows only)

Description

Indicates whether Dreamweaver is in MDI mode. In MDI mode, Dreamweaver integrates all the document windows within a single parent container or frame. When Dreamweaver is not in MDI mode, it is in classic mode, the traditional look of the Dreamweaver interface in which the user manages separate, floating windows.

Arguments

None.

Returns

`true` if Dreamweaver is in MDI mode; `false` if Dreamweaver is in classic mode.

Example

```
if(dw.isMDI())
{
  dw.cascade()
}
```

dreamweaver.minimizeRestoreAll()

Availability

Dreamweaver 4

Description

Minimizes (reduces the window to an icon) or restores all windows in Dreamweaver.

Arguments

bMinimize

bMinimize is a Boolean value. `true` indicates that windows should be minimized; `false` indicates that minimized windows should be restored.

Returns

Nothing.

dreamweaver.setActiveWindow()

Availability

Dreamweaver 3

Description

Activates the window that contains the specified document.

Arguments

documentObject, *{bActivateFrame}*

- *documentObject* is the object at the root of a document's DOM tree (the value that `dreamweaver.getDocumentDOM()` returns).
- *bActivateFrame*, applicable only if *documentObject* is inside a frameset, is a Boolean value that indicates whether to activate the frame that contains the document as well as the window that contains the frameset.

Returns

Nothing.

dreamweaver.setFloaterVisibility()

Availability

Dreamweaver 3

Description

Specifies whether to make a particular floating panel or inspector visible.

Arguments

floaterName, *bIsVisible*

- *floaterName* is the name of a floating panel. If *floaterName* does not match one of the built-in panel names, Dreamweaver searches in the Configuration/Floaters folder for a file called *floaterName*.htm where *floaterName* is the name of a floating panel. If Dreamweaver cannot find a floating panel named *floaterName*, this function has no effect.

  The *floaterName* values for built-in Dreamweaver panels are the strings to the right of the panel names in the following list:

  Assets ="assets"
  Answers = "answers"
  Behaviors = "behaviors"
  Code inspector = "html"
  Components = "server components"
  CSS Styles = "css styles"
  Databases = "databases"
  Bindings = "data bindings"
  Frames = "frames"
  FTP Log = "ftplog"
  History = "history"
  HTML Styles = "html styles"
  Insert bar = "objects"
  Layers = "layers"
  Link Checker Results = "linkchecker"
  Properties = "properties"
  Reference = "reference"
  Report Results = "reports"
  Search Results = "search"
  Server Behaviors = "server behaviors"
  Server Debug = "debug"
  Site = "site files"
  Sitespring = "sitespring"
  Snippets = "snippets"
  Tag inspector = "tag inspector"
  Target Browser Check Results ="btc"
  Timelines = "timelines"
  Validation Results = "validation"

- *bIsVisible* is a Boolean value that indicates whether to make the floating panel visible.

Returns

Nothing.

dreamweaver.setPrimaryView()

Availability

Dreamweaver 4

Description

Displays the specified view at the top of the Document window.

Arguments

viewString

viewString is the view to bring to the top of the Document window; it can be one of the following values: "design" or "code".

Returns

Nothing.

dreamweaver.setSnapDistance()

Availability

Dreamweaver 4

Description

Sets the snapping distance in pixels (0 turns it off; default is 10 pixels).

Arguments

snapDistance

snapDistance is an integer that represents the snapping distance in pixels. The default is 10 pixels. Specify 0 to turn off the Snap feature.

Returns

Nothing.

dreamweaver.showProperties()

Availability

Dreamweaver 3

Description

Makes the Property inspector visible and gives it focus.

Arguments

None.

Returns

Nothing.

dreamweaver.tileHorizontally()

Availability

Dreamweaver MX (Windows only)

Description

In MDI mode, dw.tileHorizontally() tiles the document windows horizontally, positioning each window next to another one without overlapping the documents. This process is similar to splitting the workspace vertically. If Dreamweaver is not in MDI mode, dw.tileHorizonatally has no effect.

Arguments

None.

Returns

Nothing.

Example

```
if(dw.isMDI())
{
     dw.tileHorizontally()
}
```

dreamweaver.tileVertically()

Availability

Dreamweaver MX (Windows only)

Description

In MDI mode, `dw.tileVertically()` tiles the document window vertically, positioning one document window below the other without overlapping documents. This is similar to splitting the workspace horizontally. If Dreamweaver is not in MDI mode, `dw.tileVertically` has no effect.

Arguments

None.

Returns

Nothing.

Example

```
if(dw.isMDI())
{
     dw.tileVertically()
}
```

dreamweaver.toggleFloater()

Availability

Dreamweaver 3

Description

Shows, hides, or brings to the front the specified panel or inspector.

Note: This function is meaningful only in the menus.xml file. To show, bring forward, or hide a floating panel, use dw.setFloaterVisibility().

Arguments

`floaterName`

`floaterName` is the name of the window. If the floating panel name is `reference`, the visible/invisible state of the Reference panel is updated by the user's selection in Code view. All other panels track the selection all the time, but the Reference panel tracks the selection in Code view only when the user invokes tracking.

Returns

Nothing.

dreamweaver.updateReference()

Availability

Dreamweaver 4

Description

Updates the Reference floating panel. If the Reference floating panel is not visible, `dreamweaver.updateReference()` makes it visible and then updates it.

Arguments

None.

Returns

Nothing.

APPENDIX A

Deprecated JavaScript API functions

The functions in this appendix are deprecated JavaScript API functions. Deprecated functions work but have been superseded by new Dreamweaver functions or features. You should use the newer alternatives because support for the deprecated functions might be withdrawn in future Dreamweaver versions.

dreamweaver.cssStylePalette.getSelectedTarget()

Availability

Dreamweaver 3, deprecated in Dreamweaver MX because there is no longer an Apply To Menu in the CSS Styles panel.

Description

Gets the selected element in the Apply To pop-up menu at the top of the CSS Styles panel.

Arguments

None.

Returns

Deprecated function; always returns a null value.

dreamweaver.exportEditableRegionsAsXML()

Availability

Dreamweaver 3, deprecated in MX.

Description

Opens the Export Editable Regions as XML dialog box.

Arguments

None.

Returns

Nothing.

dreamweaver.getBehaviorEvent()

Availability

Dreamweaver 1.2; deprecated in Dreamweaver 2 because actions are now selected before events.

Description

In a Behavior action file, gets the event that triggers this action.

Arguments

None.

Returns

A string that represents the event. This is the same string that is passed as an argument (*event*) to the `canAcceptBehavior()` function.

dreamweaver.getObjectRefs()

Availability

Dreamweaver 1, deprecated in 3

Description

Scans the specified document for instances of the specified tags or, if no tags are specified, for all tags in the document and formulates browser-specific references to those tags. This function is equivalent to calling `getElementsByTagName()` and then calling `dreamweaver.getElementRef()` for each tag in the `nodelist`.

Arguments

NSorIE, *sourceDoc*, *{tag1}*, *{tag2}*,...*{tagN}*

- *NSorIE* must be either `"NS 4.0"` or `"IE 4.0"`. The DOM and rules for nested references differ in Netscape Navigator 4.0 and Internet Explorer 4.0. This argument specifies for which browser to return a valid reference.
- *sourceDoc* must be `"document"`, `"parent"`, `"parent.frames[`*number*`]"`, `"parent.frames['`*frameName*`']"`, or a URL. `document` specifies the document that has the focus and contains the current selection. `parent` specifies the parent frameset (if the currently selected document is in a frame), and `parent.frames[`*number*`]` and `parent.frames['`*frameName*`']` specify a document that is in a particular frame within the frameset that contains the current document. If the argument is a relative URL, it is relative to the extension file.
- The third and subsequent arguments, if supplied, are the names of tags (for example, `"IMG"`, `"FORM"`, `"HR"`).

Returns

An array of strings where each array is a valid JavaScript reference to a named instance of the requested tag type in the specified document (for example, `"document.myLayer.document.myImage"`) for the specified browser.

- Dreamweaver returns correct references for Internet Explorer for `A`, `AREA`, `APPLET`, `EMBED`, `DIV`, `SPAN`, `INPUT`, `SELECT`, `OPTION`, `TEXTAREA`, `OBJECT`, and `IMG` tags.
- Dreamweaver returns correct references for Netscape Navigator for `A`, `AREA`, `APPLET`, `EMBED`, `LAYER`, `ILAYER`, `SELECT`, `OPTION`, `TEXTAREA`, `OBJECT`, and `IMG` tags, and for absolutely positioned `DIV` and `SPAN` tags. For `DIV` and `SPAN` tags that are not absolutely positioned, Dreamweaver returns `"cannot reference <`*tag*`>"`.

- Dreamweaver does not return references for unnamed objects. If an object does not contain either a NAME or an ID attribute, Dreamweaver returns "unnamed <*tag*>". If the browser does not support a reference by name, Dreamweaver references the object by index (for example, document.myform.applets[3]).
- Dreamweaver does return references for named objects that are contained in unnamed forms and layers (for example, document.forms[2].myCheckbox).

When the same list of arguments passes to getObjectTags(), the two functions return arrays of the same length and with parallel content.

dreamweaver.getObjectTags()

Availability

Dreamweaver1, deprecated in 3

Description

Scans the specified document for instances of the specified tags or, if no tags are specified, for all tags in the document. This function is equivalent to calling getElementsByTagName() and then getting outerHTML for each element in the nodelist.

Arguments

sourceDoc, *{tag1}*, *{tag2}*,...*{tagN}*

- *sourceDoc* must be "document", "parent", "parent.frames[*number*]", "parent.frames['*frameName*']", or a URL. document specifies the document that has the focus and contains the current selection. parent specifies the parent frameset (if the currently selected document is in a frame), and parent.frames[*number*] and parent.frames['*frameName*'] specify a document that is in a particular frame within the frameset that contains the current document. If the argument is a relative URL, it is relative to the extension file.
- The second and subsequent arguments, if supplied, are the names of tags (for example, "IMG", "FORM", "HR").

Returns

An array of strings where each array is the source code for an instance of the requested tag type in the specified document.

- If one of the *tag* arguments is LAYER, the function returns all LAYER and ILAYER tags and all absolutely positioned DIV and SPAN tags.
- If one of the *tag* arguments is INPUT, the function returns all form elements. To get a particular type of form element, specify INPUT/*TYPE*, where *TYPE* is button, text, radio, checkbox, password, textarea, select, hidden, reset, or submit.

When the same list of arguments passes to getObjectRefs(), the two functions return arrays of the same length.

Example

`dreamweaver.getObjectTags("document", "IMG")`, depending on the contents of the active document, might return an array with the following items:

- `"<IMG SRC="/images/dot.gif" WIDTH="10" HEIGHT="10" NAME="bullet">"`
- `"<IMG SRC="header.gif" WIDTH="400" HEIGHT="32" NAME="header">"`
- `"<IMG SRC="971208_nj.jpg" WIDTH="119" HEIGHT="119" NAME="headshot">"`

dreamweaver.getSelection()

Availability

Dreamweaver 2, deprecated in 3. See "dom.getSelection()" on page 546.

Description

Gets the selection in the current document, which is expressed as byte offsets into the document's source code.

Arguments

None.

Returns

An array that contains two integers. The first integer is the byte offset for the beginning of the selection; the second integer is the byte offset for the end of the selection. If the two numbers are the same, the current selection is an insertion point.

dreamweaver.libraryPalette.deleteSelectedItem()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of using `dreamweaver.assetPalette.setSelectedCategory()`, then calling `dreamweaver.assetPalette.removeFromFavorites()`.

Description

Removes the selected library item from the Library panel and deletes its associated Dreamweaver Library Item (LBI) file from the Library folder at the root of the current site. Instances of the deleted item might still exist in pages throughout the site.

Arguments

None.

Returns

Nothing.

dreamweaver.libraryPalette.getSelectedItem()

Availability

Dreamweaver 3, deprecated in 4 in favor of `dreamweaver.assetPalette.getSelectedItems()`.

Description

Gets the path of the selected library item.

Arguments

None.

Returns

A string that contains the path of the library item, which is expressed as a file:// URL.

dreamweaver.libraryPalette.newFromDocument()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of using `dreamweaver.assetPalette.setSelectedCategory()`, then calling `dreamweaver.assetPalette.newAsset()`.

Description

Creates a new library item based on the selection in the current document.

Arguments

`bReplaceCurrent`

`bReplaceCurrent` is a Boolean value that indicates whether to replace the selection with an instance of the newly created library item.

Returns

Nothing.

dreamweaver.libraryPalette.recreateFromDocument()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of `dreamweaver.assetPalette.recreateLibraryFromDocument()`.

Description

Creates an LBI file for the selected instance of a library item in the current document. This function is equivalent to clicking Recreate in the Property inspector.

Arguments

None.

Returns

Nothing.

dreamweaver.libraryPalette.renameSelectedItem()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of using `dreamweaver.assetPalette.setSelectedCategory()` with the "library" argument, then calling `dreamweaver.assetPalette.renameNickname()`.

Description

Turns the name of the selected library item into an text field, so the user can rename the selection.

Arguments

None.

Returns

Nothing.

dreamweaver.nodeToOffsets()

Availability

Dreamweaver 2, deprecated in 3 in favor of `dom.nodeToOffsets()`.

Description

Gets the position of a specific node in the DOM tree, which is expressed as byte offsets into the document's source code.

Arguments

`node`

`node` must be a tag, comment, or range of text that is a node in the tree that `dreamweaver.getDocumentDOM()` returns.

Returns

An array that contains two integers. The first integer is the byte offset for the beginning of the tag, text, or comment; the second integer is the byte offset for the end of the node.

dreamweaver.templatePalette.getSelectedTemplate()

Availability

Dreamweaver 3, deprecated in 4 in favor of `dreamweaver.assetPalette.getSelectedItems()`.

Description

Gets the path of the selected template.

Arguments

None.

Returns

A string that contains the path of the template, which is expressed as a file:// URL.

dreamweaver.offsetsToNode()

Availability

Dreamweaver 2, deprecated in 3 in favor of `dom.offsetsToNode()`.

Description

Gets the object in the DOM tree that completely contains the range of characters between the specified beginning and end points.

Arguments

`offsetBegin`, *`offsetEnd`*

The arguments are the beginning and end points, respectively, of a range of characters, which is expressed as byte offsets into the document's source code.

Returns

The tag, text, or comment object that completely contains the specified range of characters.

dreamweaver.popupCommand()

Availability

Dreamweaver 2, deprecated in 3 in favor of `dreamweaver.runCommand()`.

Description

Executes the specified command. To the user, the effect is the same as choosing the command from a menu; if a dialog box is associated with the command, it appears. This function provides the ability to call a command from another extension file. It blocks other edits until the user dismisses the dialog box.

Note: This function can be called only within `objectTag()` or in any script in a command or Property inspector file.

Arguments

`commandFile`

`commandFile` is the name of a command file within the Configuration/Commands folder (for example, `"Format Table.htm"`).

Returns

Nothing.

dreamweaver.setSelection()

Availability

Dreamweaver 2, deprecated in 3 in favor of `dom.setSelection()`.

Description

Sets the selection in the current document. This function can move the selection only within the current document; it cannot change the focus to a different document.

Arguments

`offsetBegin`, *`offsetEnd`*

The arguments are the beginning and end points, respectively, for the new selection, which is expressed as byte offsets into the document's source code. If the two numbers are the same, the new selection is an insertion point. If the new selection is not a valid HTML selection, it is expanded to include the characters in the first valid HTML selection. For example, if *`offsetBegin`* and *`offsetEnd`* define the range `SRC="myImage.gif"` within `<IMG SRC="myImage.gif">`, the selection expands to include the entire `IMG` tag.

Returns

Nothing.

dreamweaver.templatePalette.deleteSelectedTemplate()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of using `dreamweaver.assetPalette.setSelectedCategory()` with the "`templates`" argument, then calling `dreamweaver.assetPalette.removeFromFavorites()`.

Description

Deletes the selected template from the templates folder.

Arguments

None.

Returns

Nothing.

dreamweaver.templatePalette.renameSelectedTemplate()

Availability

Dreamweaver 3, deprecated in Dreamweaver 4 in favor of using dreamweaver.assetPalette.setSelectedCategory() with the `"templates"` argument, then calling `dreamweaver.assetPalette.renameNickname()`.

Description

Turns the name of the selected template into an text field, so the user can rename the selection.

Arguments

None.

Returns

Nothing.

INDEX

A

action files 135
addBehavior() 380, 614
addDebugContextData() 544
addDynamicSource() 193
addFrame() 614
addItem() 537
addKeyframe() 614
addLinkToExistingFile() 558
addLinkToNewFile() 559
addObject() 615
addResultItem() 538
addSpacerToColumn() 516
addTimeline() 615
alert() 42
align() 508
APIs, types of 17
  behavior 136
  C language 257
  command 62
  Component panel 208
  data formatting 199
  data source 193
  data translator 225
  database 311
  database connection dialog box 338
  design note 288
  file I/O 271
  Fireworks integration 299
  Flash object 307
  floating panel 127
  HTTP 281
  JavaBeans 345
  JavaScript (core) 371
  JavaScript debugger module 245
  menu command 68
  object 57
  Property inspector 121
  report 104
  server behavior 151
  server model 217
  Source Control Integration 350
  Tag editor 117
  toolbar 93
applyBehavior() 136
applyCharacterMarkup() 466
applyConnection() 341
applyCSSStyle() 403
applyFontMarkup() 466
applyFormat() 202
applyFormatDefinition() 202
applyHTMLStyle() 495
applySB() 157
applySelectedStyle() 404
applyTag() 118
applyTemplate() 521
appName property 47
appVersion property 47
arguments
  optional 372
  passed from menuitem 67
  receiveArguments() 70
arguments attribute 93
arrange() 508
arrangeFloatingPalettes() 515
array object 42
arrowDown() 502, 590
arrowLeft() 502, 591
arrowRight() 503, 591
arrowUp() 503, 591
assetPalette.addToFavoritesFromDocument() 372
assetPalette.addToFavoritesFromSiteAssets() 373
assetPalette.addToFavoritesFromSiteWindow() 373
assetPalette.canEdit() 418
assetPalette.canInsertOrApply() 418
assetPalette.copyToSite() 373

assetPalette.edit() 374
assetPalette.getSelectedCategory() 374
assetPalette.getSelectedItems() 374
assetPalette.getSelectedView() 375
assetPalette.insertOrApply() 375
assetPalette.locateInSite() 376
assetPalette.newAsset() 376
assetPalette.newFolder() 376
assetPalette.recreateLibraryFromDocument() 377
assetPalette.refreshSiteAssets() 377
assetPalette.removeFromFavorites() 377
assetPalette.renameNickname() 378
assetPalette.setSelectedCategory() 378
assetPalette.setSelectedView() 378
assets panel functions 372
attachExternalStylesheet() 404
attribute translators 229
  creating 229
  debugging 241
  sample code 230
attributes
  arguments 93
  checked 91
  colorRect 90
  command 92
  disabledImage 89
  domRequired 90
  enabled 90
  id 88
  image 88
  Insertbar tag 54
  label 89
  menu_ID 90
  overImage 89
  showif 88
  snippets
    tag attributes 582
  toolbars item tag 88
  tooltip 89
  update 91
  value 91
  width 89
attributes property 45
attributes tag 180
attributes, file 90

B

backspaceKey() 503
balanceBracesTextView() 592
beep() 480
beginReporting() 105
behavior API
  applyBehavior() 136
  behaviorFunction() 137
  canAcceptBehavior() 138
  deleteBehavior() 139
  displayHelp() 139
  identifyBehaviorArguments() 140
  inspectBehavior() 141
  windowDimensions() 142
behavior extensions
  definition 20
behaviorFunction() 137
behaviors
  API 136
  helper functions 136
  inserting multiple functions with 136
  required functions 136
  sample code 143
  user experience 135
Binding inspector 191
block/tag translators 229
  debugging 241
  sample code 235
blur() 42
body property 44
boolean object 42
bringDWToFront() 299
bringFWToFront() 299
browseDocument() 432, 441
browseForFileURL() 449
browseForFolderURL() 449
browser targets 541
button
  object 42
  tag 83
button tag 53

C

C extensibility API
  JS_BooleanToValue() 257
  JS_DoubleToValue() 257
  JS_ExecuteScript() 259
  JS_GetArrayLength() 258
  JS_GetElement() 259
  JS_IntegerToValue() 257
  JS_NewArrayObject() 258
  JS_ObjectToValue() 257
  JS_ObjectType() 258
  JS_ReportError() 260
  JS_SetElement() 259
  JS_StringToValue() 256

JS_ValueToBoolean() 256
JS_ValueToDouble() 255
JS_ValueToInteger() 255
JS_ValueToObject() 256
JS_ValueToString() 255
MM_ConfigFileExists() 263
MM_GetConfigFileAttributes() 264
MM_GetConfigFolderList() 262
MM_OpenConfigFile() 263
C functions
calling from JavaScript 267
in the mm_jsapi.h file 253
canAcceptBehavior() 138
canAcceptCommand()
in menu commands 68
using 93
canAddFrame() 430
canAddKeyFrame() 431
canAddLinkToFile() 433
canAlign() 409
canApplyTemplate() 409
canArrange() 410
canChangeLink() 433
canChangeObject() 431
canCheckIn() 433
canCheckOut() 434
canClear() 426
canClipCopy() 418, 427
canClipCopyText() 410
canClipCut() 419, 427
canClipPaste() 410, 419, 427
canClipPasteText() 410
canCloak() 434
canConnect() 435
canConvertLayersToTable() 411
canConvertTablesToLayers() 411
canDecreaseColspan() 411
canDecreaseRowspan() 412
canDeleteTableColumn() 412
canDeleteTableRow() 412
canEditColumns() 559
canEditNoFramesContent() 412
canEditSelection() 426
canExportCSS() 420
canExportTemplateDataAsXML() 420
canFindLinkSource() 435
canFindNext() 420
canGet() 435
canIncreaseColspan() 413
canIncreaseRowspan() 413
canInsertObject() 51, 57
canInsertTableColumns() 413
canInsertTableRows() 414
canLocateInSite() 436
canMakeEditable() 436
canMakeNewEditableRegion() 414
canMakeNewFileOrFolder() 436
canMarkSelectionAsEditable() 414
canMergeTableCells() 414
canOpen() 437
canOpenInBrowser() 427
canOpenInEditor() 428
canOpenInFrame() 420
canPlayPlugin() 415
canPlayRecordedCommand() 421
canPopupEditTagDialog() 421
canPut() 437
canRecognizeDocument() 217
canRecreateCache() 437
canRedo() 415, 421
canRefresh() 438
canRemoveEditableRegion() 415
canRemoveFrame() 432
canRemoveKeyFrame() 432
canRemoveLink() 438
canRemoveObject() 432
canRevertDocument() 422
canSave() 428
canSaveAll() 422
canSaveDocument() 422
canSaveDocumentAsTemplate() 423
canSaveFrameset() 423
canSaveFramesetAs() 423
canSelectAll() 424, 428
canSelectAllCheckedOutFiles() 438
canSelectNewer() 439
canSelectTable() 416
canSetLayout() 438
canSetLinkHref() 416
canShowFindDialog() 424
canShowListPropertiesDialog() 416
canSplitFrame() 416
canSplitTableCell() 417
canStopPlugin() 417
canSynchronize() 439
canUncloak() 440
canUndo() 417, 424
canUndoCheckOut() 440
canViewAsRoot() 440
Cascading Style Sheets to HTML markup 402

category tag 53
changeLink() 560
changeLinkSitewide() 559
changeObject() 615
checkbox object 42
checkbutton tag 53, 84
checked attribute 91
checkIn() 560
checkLinks() 560
checkOut() 561
checkSpelling() 483
checkTargetBrowsers() 483, 561
childNodes property
  of comment objects 46
  of document objects 44
  of tag objects 45
  of text objects 46
cleanupXHTML() 447
clearInterval() 42
clearItems() 541
clearSteps() 490
clearTemp() 282
clearTimeout() 42
C-level extensibility, in translators 225
clipCopy() 388, 390, 541
clipCopyText() 388
clipCut() 388, 391, 542
clipPaste() 389, 391, 542
clipPasteText() 389
cloak() 562
close() 42
closeDocument() 450
CloseNotesFile() 292
closeTag tag 181
code editing, enhanced 14
Code Hint extensions, definition 20
Code Snippet extensions, definition 20
Code view 588
CodeHints
  codehints tag 393
  description tag 394
  function tag 396
  menu tag 395
  menugroup tag 394
  menuitem tag 395
codeHints.addFunction() 398
codeHints.addMenu() 397
codeHints.resetMenu() 399
codeHints.showCodeHints() 400
CodeHints.xml file 392
ColdFusion Component Explorer 566, 569
color button control 40
colorpicker tag 87
colorRect attribute 90
columns
  getting from statements 325
  getting from stored procedures 329
combobox tag 86
command API
  canAcceptCommand() 62
  commandButtons() 62
  isDomRequired() 63
  receiveArguments() 63
  windowDimensions() 64
command attribute 92
Command extensions
  definition 20
Command menu functions 400
commandButtons() 105
  in menu commands 69
commands
  adding to menus 66
  sample code 65
  toolbars 79
  user experience 61
comment object 46
Component panel
  files 205
  tree control 207
Component panel API functions
  displayHelp() 208
  displayInstructions() 207
  getCodeViewDropCode() 212
  getComponentChildren() 208
  getContextMenuId() 210
  getSetupSteps() 212
  handleDoubleClick() 213
  setupStepsCompleted() 213
  toolbarControls() 214
components 205
configurations, multiple 14
configureSettings() 106
confirm() 42
connection handling, database 15, 312
connection objects, properties 340
connection types, creating 337
connection_includefile.edml connection definition file 343
connections 315
  getting list of 314

names 324
connectivity functions overview 337
conventions, in book 12
conversion functions 402
convertLayersToTable() 402
convertTablesToLayers() 402
convertTo30() 402
convertToXHTML() 448
convertWidthsToPercent() 602
convertWidthsToPixels() 602
copy() 271
copySteps() 490
createDocument() 450
createFolder() 272
createLayoutCell() 516
createLayoutTable() 516
createResultsWindow() 537
createXHTMLDocument() 451
createXMLDocument() 452
CSS style functions 403
cssStyle.canEditSelectedStyle() 425
cssStylePalette.canApplySelectedStyle() 424
cssStylePalette.canDeleteSelectedStyle() 425
cssStylePallette.canEditStyleSheet() 425
custom JavaScript controls 33
customizing or extending Dreamweaver 11

D

data formatting 199
Data Manager 181
data property
of comment objects 46
of httpReply objects 281
of text objects 46
data source API 193
addDynamicSource() 193
deleteDynamicSource() 193
displayHelp() 194
editDynamicSource() 194
findDynamicSources() 194
generateDynamicDataRef() 195
generateDynamicSourceBindings() 196
inspectDynamicDataRef() 197
data source extensions
definition 21
data source functions 588
data sources 191
data translator API
getTranslatorInfo() 226
liveDataTranslateMarkup function() 228
translateMarkup() 228
data translator extensions
definition 21
data translators
debugging 241
for attributes 229
for tags or blocks of code 234
kinds of 229
user experience 225
database access functions 324
database API 311
access functions 324
connection functions 312
MMDB.deleteConnection() 312
MMDB.getColdFusionDsnList() 313
MMDB.getColumnAndTypeList() 325
MMDB.getColumnList() 325
MMDB.getColumns() 326
MMDB.getColumnsOfTable() 327
MMDB.getConnection() 313
MMDB.getConnectionList() 314
MMDB.getConnectionName() 315
MMDB.getConnectionString() 315
MMDB.getDriverName() 316
MMDB.getDriverUrlTemplateList() 316
MMDB.getLocalDsnList() 317
MMDB.getPassword() 317
MMDB.getPrimaryKeys() 327
MMDB.getProcedures() 328
MMDB.getRdsPassword() 318
MMDB.getRdsUserName() 318
MMDB.getRemoteDsnList() 318
MMDB.getRuntimeConnectionType() 319
MMDB.getSPColumnList() 329
MMDB.getSPColumnListNamedParams() 329
MMDB.getSPParameters() 330
MMDB.getSPParamsAsString() 331
MMDB.getTables() 332
MMDB.getUserName() 319
MMDB.getViews() 332
MMDB.hasConnectionWithName() 320
MMDB.needToPromptForRdsInfo() 320
MMDB.needToRefreshColdFusionDsnList() 320
MMDB.popupConnection() 321
MMDB.setRdsPassword() 321
MMDB.setRdsUserName() 322
MMDB.showColdFusionAdmin() 322
MMDB.showConnectionMgrDialog() 322
MMDB.showOdbcDialog() 323
MMDB.showRdsUserDialog() 323
MMDB.showRestrictDialog() 323

MMDB.showResultset() 333
MMDB.showSPResultset() 334
MMDB.showSPResultsetNamedParams() 334
MMDB.testConnection() 324
database connection dialog box API 338
applyConnection() 341
definition files 343
findConnection() 339
include files
generated 341
inspectConnection() 341
database connection functions 312
database connection type definition files 343
database controls 36
database tree controls 36
databases
access functions 324
API 311
connection dialog box API 338
connection functions 312
connection handling 15
connection type definition files 343
dataSource attribute 161
date object 42
debugDocument() 500
decreaseColspan() 602
decreaseRowspan() 603
defineSites() 562
delete tag 175
deleteBehavior() 139
deleteConnection() 312
deleteDynamicSource() 193
deleteFormat() 203
deleteKey() 504
deleteSB() 157
deleteSelectedItem() 656
deleteSelectedStyle() 405, 495
deleteSelectedTemplate() 659
deleteSelection() 466, 477, 562
deleteTableColumn() 603
deleteTableRow() 603
deleteType attribute 176
deprecated functions 653
description attribute 582
description tag 394
design notes
C API 292
file structure 287
JavaScript API 288
user experience 287
design notes API
MMNotes.close() 288
MMNotes.filePathToLocalURL() 288
MMNotes.get() 288
MMNotes.getKeyCount() 289
MMNotes.getKeys() 289
MMNotes.getSiteRootForFile() 290
MMNotes.getVersionName() 290
MMNotes.getVersionNum() 290
MMNotes.localURLToFilePath() 290
detachFromLibrary() 521
detachFromTemplate() 522
disabledImage attribute 89
display tag 181
displayHelp()
in behavior API 139
in Component panel API 208
in data source API 194
in floating panel API 127
in floating panels 127
in object API 57
in object files 57
in Property inspector API 121
in server behavior API 154
displayInstructions() 207
docking toolbars 78
DOCTYPE 32
document extensions 28
document node 44
document object
DOM Level 1 properties and methods of 44
Netscape DOM properties and methods of 42
Document Object Model 41
DOM Level 1 specification 42
DOM object 371
Dreamweaver 42
document type extensions
definition 21
document types 22
definition file 23, 24
definition file, rules 29
dynamic templates 27
extensions 28
localizing 24, 28
new 15
opening, procedure for 30
tags in definition file 25
documentEdited() 127
documentElement property 44
doDeferredTableUpdate() 604

doesColumnHaveSpacer() 517
doesGroupHaveSpacers() 517
DOM. *See* Document Object Model.
dom.addBehavior() 380
dom.addSpacerToColumn() 516
dom.align() 508
dom.applyCharacterMarkup() 466
dom.applyCSSStyle() 403
dom.applyFontMarkup() 466
dom.applyHTMLStyle() 495
dom.applyTemplate() 521
dom.arrange() 508
dom.arrowDown() 502
dom.arrowLeft() 502
dom.arrowRight() 503
dom.arrowUp() 503
dom.backspaceKey() 503
dom.canAlign() 409
dom.canApplyTemplate() 409
dom.canArrange() 410
dom.canClipCopyText() 410
dom.canClipPaste() 410
dom.canClipPasteText() 410
dom.canConvertLayersToTable() 411
dom.canConvertTablesToLayers() 411
dom.canDecreaseColspan() 411
dom.canDecreaseRowspan() 412
dom.canDeleteTableColumn() 412
dom.canDeleteTableRow() 412
dom.canEditNoFramesContent() 412
dom.canIncreaseColspan() 413
dom.canIncreaseRowspan() 413
dom.canInsertTableColumns() 413
dom.canInsertTableRows() 414
dom.canMakeNewEditableRegion() 414
dom.canMarkSelectionAsEditable() 414
dom.canMergeTableCells() 414
dom.canPlayPlugin() 415
dom.canRedo() 415
dom.canRemoveEditableRegion() 415
dom.canSelectTable() 416
dom.canSetLinkHref() 416
dom.canShowListPropertiesDialog() 416
dom.canSplitFrame() 416
dom.canSplitTableCell() 417
dom.canStopPlugin() 417
dom.canUndo() 417
dom.checkSpelling() 483
dom.checkTargetBrowsers() 483
dom.cleanupXHTML() 447
dom.clipCopy 388
dom.clipCopyText() 388
dom.clipCut() 388
dom.clipPaste 389
dom.clipPasteText() 389
dom.convertLayersToTable() 402
dom.convertTablesToLayers() 402
dom.convertTo30() 402
dom.convertToXHTML() 448
dom.convertWidthsToPercent() 602
dom.convertWidthsToPixels() 602
dom.createLayoutCell() 516
dom.createLayoutTable() 516
dom.decreaseColspan() 602
dom.decreaseRowspan() 603
dom.deleteKey() 504
dom.deleteSelection() 466
dom.deleteTableColumn() 603
dom.deleteTableRow() 603
dom.detachFromLibrary() 521
dom.detachFromTemplate() 522
dom.doDeferredTableUpdate() 604
dom.doesColumnHaveSpacer() 517
dom.doesGroupHaveSpacers() 517
dom.editAttribute() 467
dom.endOfDocument() 504
dom.endOfLine() 504
dom.exitBlock() 467
dom.formatRange() 588
dom.formatSelection() 589
dom.getAttachedTemplate() 522
dom.getBehavior() 380
dom.getBreakpoint() 498
dom.getCharSet() 467
dom.getClickedHeaderColumn() 518
dom.getEditableRegionList() 522
dom.getEditableRetionList() 523
dom.getEditNoFramesContent() 620
dom.getFocus() 642
dom.getFontMarkup() 468
dom.getFrameNames() 464
dom.getHideAllVisualAids() 620
dom.getIsLibraryDocument() 523
dom.getIsTemplateDocument() 523
dom.getIsXHTMLDocument() 449
dom.getLineFromOffset() 498
dom.getLinkHref() 468
dom.getLinkTarget() 468
dom.getListTag() 468
dom.getPreventLayerOverlaps() 620

dom.getRulerOrigin() 511
dom.getRulerUnits() 511
dom.getSelectedEditableRegion() 523
dom.getSelectedNode() 546
dom.getSelection() 546
dom.getShowAutoIndent() 620
dom.getShowFrameBorders() 621
dom.getShowGrid() 621
dom.getShowHeadView() 621
dom.getShowImageMaps() 622
dom.getShowInvalidHTML() 622
dom.getShowInvisibleElements() 622
dom.getShowLayerBorders() 622, 628
dom.getShowLayoutTableTabs() 518
dom.getShowLayoutView() 518
dom.getShowLineNumbers() 623
dom.getShowNoscript 589
dom.getShowRulers() 623
dom.getShowSyntaxColoring() 623
dom.getShowTableBorders() 623
dom.getShowToolbar() 624
dom.getShowToolbarIconLabels() 640
dom.getShowTracingImage() 624
dom.getShowWordWrap() 624
dom.getSnapToGrid() 625
dom.getTableExtent() 604
dom.getTagSelectorTag() 608
dom.getTextAlignment() 469
dom.getTextFormat() 469
dom.getToolbarIdArray() 638
dom.getToolbarLabel() 639
dom.getToolbarVisibility() 637
dom.getTracingImageOpacity() 511
dom.getView() 642
dom.getWindowTitle() 643
dom.hasCharacterMarkup() 469
dom.hasTracingImage() 418
dom.increaseColspan() 604
dom.increaseRowspan() 604
dom.indent() 470
dom.insertHTML() 470
dom.insertLibraryItem() 524
dom.insertObject() 471
dom.insertTableColumns() 605
dom.insertTableRows() 605
dom.insertText() 471
dom.instrumentDocument() 498
dom.isColumnAutostretch() 518
dom.isDesignViewUpdated() 589
dom.isDocumentInFrame() 465
dom.isSelectionValid() 590
dom.loadTracingImage() 512
dom.makeCellWidthsConsistent() 519
dom.makeSizesEqual() 509
dom.markSelectionAsEditable() 524
dom.mergeTableCells() 606
dom.moveSelectionBy() 509
dom.newBlock() 472
dom.newEditableRegion() 524
dom.nextParagraph() 505
dom.nextWord() 505
dom.nodeToOffsets() 546
dom.notifyFlashObjectChanged() 472
dom.offsetsToNode() 547
dom.outdent() 473
dom.pageDown() 505
dom.pageUp() 506
dom.playAllPlugins() 512
dom.playPlugin() 512
dom.previousParagraph() 506
dom.previousWord() 507
dom.reapplyBehaviors() 381
dom.redo() 487
dom.removeAllSpacers() 519
dom.removeAllTableHeights() 606
dom.removeAllTableWidths() 606
dom.removeBehavior() 381
dom.removeCharacterMarkup() 473
dom.removeCSSStyle() 403
dom.removeEditableRegion() 525
dom.removeFontMarkup() 473
dom.removeLink() 474
dom.removeSpacerFromColumn() 519
domRequired attribute 90
dom.resizeSelection() 474
dom.resizeSelectionBy() 509
dom.runTranslator() 640
dom.runValidation() 483
dom.saveAllFrames() 465
dom.selectAll() 548
dom.selectChild() 534
dom.selectParent() 535
dom.selectTable() 548
dom.serverModel.getDelimiters() 553
dom.serverModel.getServerExtension() 554
dom.serverModel.getServerLanguage() 556
dom.serverModel.getServerName() 556
dom.serverModel.getServerVersion() 557
dom.setAttributeWithErrorChecking() 474
dom.setBreakpoint() 499

dom.setColumnAutostretch() 520
dom.setEditNoFramesContent() 625
dom.setHideAllVisualAids() 625
dom.setLayerTag() 510
dom.setLinkHref() 474
dom.setLinkTarget() 475
dom.setListBoxKind() 475
dom.setListTag() 476
dom.setPreventLayerOverlaps() 625
dom.setRulerOrigin() 512
dom.setRulerUnits() 513
dom.setSelectedNode() 548
dom.setSelection() 549
dom.setShowFrameBorders() 626
dom.setShowGrid() 626
dom.setShowHeadView() 626
dom.setShowImageMaps() 627
dom.setShowInvalidHTML() 627
dom.setShowInvisibleElements() 627
dom.setShowLayerBorders() 627
dom.setShowLayoutTableTabs() 520
dom.setShowLayoutView() 520
dom.setShowLineNumbers() 628
dom.setShowNoscript 590
dom.setShowRulers() 628
dom.setShowSyntaxColoring() 628
dom.setShowTableBorders() 629
dom.setShowToolbar() 629
dom.setShowToolbarIconLabels() 639
dom.setShowTracingImage() 629
dom.setShowWordWrap() 629
dom.setSnapToGrid() 630
dom.setTableCellTag() 606
dom.setTableColumns() 607
dom.setTableRows() 607
dom.setTextAlignment() 476
dom.setTextFieldKind() 476
dom.setTextFormat() 477
dom.setToolbarPosition() 638
dom.setToolbarVisibility() 637
dom.setTracingImageOpacity() 513
dom.setTracingImagePosition() 513
dom.setView() 643
dom.showFontColorDialog() 477
dom.showInsertTableRowsOrColumnsDialog() 607
dom.showListPropertiesDialog() 475
dom.showPagePropertiesDialog() 484
dom.snapTracingImageToSelection() 514
dom.source.arrowDown() 590
dom.source.arrowLeft() 591
dom.source.arrowRight() 591
dom.source.arrowUp() 591
dom.source.balanceBracesTextView() 592
dom.source.endOfDocument() 592
dom.source.endOfLine() 592
dom.source.endPage() 593
dom.source.getCurrentLines() 593
dom.source.getLineFromOffset() 594
dom.source.getSelection() 593
dom.source.getText() 594
dom.source.indentTextView() 594
dom.source.insert() 595
dom.source.nextWord() 595
dom.source.outdentTextView() 595
dom.source.pageDown() 596
dom.source.pageUp() 596
dom.source.previousWord() 596
dom.source.replaceRange() 597
dom.source.scrollEndFile() 597
dom.source.scrollLineDown() 597
dom.source.scrollLineUp() 598
dom.source.scrollPageDown() 598
dom.source.scrollPageUp() 598
dom.source.scrollTopFile() 599
dom.source.selectParentTag() 599
dom.source.setCurrentLine() 599
dom.source.startOfDocument() 600
dom.source.startOfLine() 600
dom.source.topPage() 600
dom.source.wrapSelection() 601
dom.splitFrame() 465
dom.splitTableCell() 608
dom.startOfDocument() 507
dom.startOfLine() 507
dom.stopAllPlugins() 514
dom.stopPlugin() 514
dom.stripTag() 535
dom.synchronizeDocument() 601
dom.undo() 488
dom.updateCurrentPage() 525
dom.wrapTag() 535
doURLDecoding() 484
doURLEncoding() 585
Dreamweaver DOM 42
dreamweaver object 47
  methods of 371
  properties of 47
dreamweaver.arrangeFloatingPalettes() 515
dreamweaver.assetPalette.addToFavoritesFromDocument() 372

dreamweaver.assetPalette.addToFavoritesFromSiteAssets() 373
dreamweaver.assetPalette.addToFavoritesFromSiteWindow() 373
dreamweaver.assetPalette.canEdit() 418
dreamweaver.assetPalette.canInsertOrApply() 418
dreamweaver.assetPalette.copyToSite() 373
dreamweaver.assetPalette.edit() 374
dreamweaver.assetPalette.getSelectedCategory() 374
dreamweaver.assetPalette.getSelectedItems() 374
dreamweaver.assetPalette.getSelectedView() 375
dreamweaver.assetPalette.insertOrApply() 375
dreamweaver.assetPalette.locateInSite() 376
dreamweaver.assetPalette.newAsset() 376
dreamweaver.assetPalette.newFolder() 376
dreamweaver.assetPalette.recreateLibraryFromDocument() 377
dreamweaver.assetPalette.refreshSiteAssets() 377
dreamweaver.assetPalette.removeFromFavorites() 377
dreamweaver.assetPalette.renameNickname() 378
dreamweaver.assetPalette.setSelectedCategory() 378
dreamweaver.assetPalette.setSelectedView() 378
dreamweaver.beep() 480
dreamweaver.behaviorInspector object 380
dreamweaver.behaviorInspector.getBehaviorAt() 384
dreamweaver.behaviorInspector.getBehaviorCount() 384
dreamweaver.behaviorInspector.getSelectedBehavior() 384
dreamweaver.behaviorInspector.moveBehaviorDown() 385
dreamweaver.behaviorInspector.moveBehaviorUp() 386
dreamweaver.behaviorInspector.setSelectedBehavior() 387
dreamweaver.browseDocument() 441
dreamweaver.browseForFileURL() 449
dreamweaver.browseForFolderURL() 449
dreamweaver.canClipCopy() 418
dreamweaver.canClipCut() 419
dreamweaver.canClipPaste() 419
dreamweaver.canExportCSS() 420
dreamweaver.canExportTemplateDataAsXML() 420
dreamweaver.canFindNext() 420
dreamweaver.canOpenInFrame() 420
dreamweaver.canPlayRecordedCommand() 421
dreamweaver.canPopupEditTagDialog() 421
dreamweaver.canRedo() 421
dreamweaver.canRevertDocument() 422
dreamweaver.canSaveAll() 422
dreamweaver.canSaveDocument() 422
dreamweaver.canSaveDocumentAsTemplate() 423
dreamweaver.canSaveFrameset() 423
dreamweaver.canSaveFramesetAs() 423
dreamweaver.canSelectAll() 424
dreamweaver.canShowFindDialog() 424
dreamweaver.canUndo() 424
dreamweaver.clipCopy() 390
dreamweaver.clipCut() 391
dreamweaver.clipPaste() 391
dreamweaver.closeDocument() 450
dreamweaver.codeHints.addFunction() 398
dreamweaver.codeHints.addMenu() 397
dreamweaver.codeHints.resetMenu() 399
dreamweaver.codeHints.showCodeHints() 400
dreamweaver.createDocument() 450
dreamweaver.createResultsWindow() 537
dreamweaver.createXHTMLDocument() 451
dreamweaver.createXMLDocument() 452
dreamweaver.cssStyle.canEditSelectedStyle() 425
dreamweaver.cssStylePalette object 403
dreamweaver.cssStylePalette.applySelectedStyle() 404
dreamweaver.cssStylePalette.canApplySelectedStyle() 424
dreamweaver.cssStylePalette.canDeleteSelectedStyle() 425
dreamweaver.cssStylePalette.canDuplicateSelectedStyle() 425
dreamweaver.cssStylePalette.deleteSelectedStyle() 405
dreamweaver.cssStylePalette.duplicateSelectedStyle() 405
dreamweaver.cssStylePalette.editSelectedStyle() 405
dreamweaver.cssStylePalette.editStyleSheet() 406
dreamweaver.cssStylePalette.getSelectedStyle() 406
dreamweaver.cssStylePalette.getSelectedTarget() 406, 653
dreamweaver.cssStylePalette.getStyles() 407
dreamweaver.cssStylePalette.newStyle() 408
dreamweaver.cssStylePallette.canEditStyleSheet() 425
dreamweaver.dbi.getDataSources() 408
dreamweaver.debugDocument() 500
dreamweaver.deleteSelection() 477
dreamweaver.doURLDecoding() 484
dreamweaver.doURLEncoding() 585
dreamweaver.editCommandList() 400
dreamweaver.editFontList() 478
dreamweaver.editLockedRegions() 640
dreamweaver.exportCSS() 452
dreamweaver.exportEditableRegionsAsXML() 653
dreamweaver.exportTemplateDataAsXML() 452, 525

dreamweaver.findNext() 460
dreamweaver.getActiveWindow() 644
dreamweaver.getBehaviorElement() 381
dreamweaver.getBehaviorEvent() 653
dreamweaver.getBehaviorTag() 382
dreamweaver.getBrowserList() 442
dreamweaver.getClipboardText() 391
dreamweaver.getConfigurationPath() 532
dreamweaver.getDebugBrowserList() 500
dreamweaver.getDocumentDOM() 453
dreamweaver.getDocumentList() 644
dreamweaver.getDocumentPath() 532
dreamweaver.getElementRef() 484
dreamweaver.getExtDataArray() 182
dreamweaver.getExtDataValue() 182
dreamweaver.getExtensionEditorList() 442
dreamweaver.getExternalTextEditor() 442
dreamweaver.getExtGroups() 183
dreamweaver.getExtParticipants() 182
dreamweaver.getFlashPath() 443
dreamweaver.getFloaterVisibility() 644
dreamweaver.getFocus() 645
dreamweaver.getFontList() 478
dreamweaver.getFontStyles() 478
dreamweaver.getHideAllFloaters() 630
dreamweaver.getIsAnyBreakpoints() 501
dreamweaver.getKeyState() 479
dreamweaver.getLiveDataInitTags() 526
dreamweaver.getLiveDataMode() 527
dreamweaver.getLiveDataParameters () 527
dreamweaver.getMenuNeedsUpdating() 530
dreamweaver.getObjectRefs() 654
dreamweaver.getObjectTags() 655
dreamweaver.getParticipants() 155
dreamweaver.getPreferenceInt() 485
dreamweaver.getPreferenceString() 486
dreamweaver.getPrimaryBrowser() 443
dreamweaver.getPrimaryExtensionEditor() 443
dreamweaver.getPrimaryView() 646
dreamweaver.getRecentFileList() 454
dreamweaver.getRedoText() 488
dreamweaver.getSecondaryBrowser() 444
dreamweaver.getShowDialogsOnInsert() 480
dreamweaver.getShowStatusBar() 630
dreamweaver.getSiteRoot() 533
dreamweaver.getSnapDistance() 646
dreamweaver.getSystemFontList() 480
dreamweaver.getTokens() 585
dreamweaver.getTranslatorList() 641
dreamweaver.getUndoText() 488
dreamweaver.historyPalette object 487, 498
dreamweaver.historyPalette.clearSteps() 490
dreamweaver.historyPalette.copySteps() 490
dreamweaver.historyPalette.getSelectedSteps() 491
dreamweaver.historyPalette.getStepCount() 492
dreamweaver.historyPalette.getStepsAsJavaScript() 492
dreamweaver.historyPalette.getUndoState() 493
dreamweaver.historyPalette.replaySteps() 493
dreamweaver.historyPalette.saveAsCommand() 493
dreamweaver.historyPalette.setSelectedSteps() 494
dreamweaver.historyPalette.setUndoState() 494
dreamweaver.htmlInspector.getShowAutoIndent() 631
dreamweaver.htmlInspector.getShowHighlightInvalidHTML() 631
dreamweaver.htmlInspector.getShowLineNumbers() 631
dreamweaver.htmlInspector.getShowSyntaxColoring() 631
dreamweaver.htmlInspector.getShowWordWrap() 632
dreamweaver.htmlInspector.setShowAutoIndent() 632
dreamweaver.htmlInspector.setShowHighightInvalidHTML() 632
dreamweaver.htmlInspector.setShowLineNumbers() 633
dreamweaver.htmlInspector.setShowSyntaxColoring() 633
dreamweaver.htmlInspector.setShowWordWrap() 633
dreamweaver.htmlStylePalette object 495
dreamweaver.htmlStylePalette.canEditSelection() 426
dreamweaver.htmlStylePalette.deleteSelectedStyle() 495
dreamweaver.htmlStylePalette.duplicateSelectedStyle() 495
dreamweaver.htmlStylePalette.editSelectedStyle() 496
dreamweaver.htmlStylePalette.getSelectedStyle() 496
dreamweaver.htmlStylePalette.getStyles() 496
dreamweaver.htmlStylePalette.newStyle() 496
dreamweaver.htmlStylePalette.setSelectedStyle() 497
dreamweaver.importXMLIntoTemplate() 454
dreamweaver.isRecording() 426
dreamweaver.isReporting() 536
dreamweaver.latin1ToNative() 586
dreamweaver.libraryPalette object 521
dreamweaver.libraryPalette.deleteSelectedItem() 656
dreamweaver.libraryPalette.getSelectedItem() 656
dreamweaver.libraryPalette.newFromDocument() 657
dreamweaver.libraryPalette.recreateFromDocument() 657
dreamweaver.libraryPalette.renameSelectedItem() 657
dreamweaver.liveDataTranslate() 528

dreamweaver.loadSitesFromPrefs() 558
dreamweaver.minimizeRestoreAll() 647
dreamweaver.nativeToLatin1() 586
dreamweaver.newDocumentDOM() 454
dreamweaver.newFromTemplate() 455
dreamweaver.nodeExists() 549
dreamweaver.nodeToOffsets() 658
dreamweaver.notifyMenuUpdated() 531
dreamweaver.offsetsToNode() 658
dreamweaver.openDocument() 455
dreamweaver.openDocumentFromSite() 455
dreamweaver.openInFrame() 456
dreamweaver.openWithApp() 445
dreamweaver.openWithBrowseDialog() 446
dreamweaver.openWithExternalTextEditor() 446
dreamweaver.openWithImageEditor() 446
dreamweaver.outputResultsPanel.clearItems() 541
dreamweaver.outputResultsPanel.clipCopy() 541
dreamweaver.outputResultsPanel.clipCut() 542
dreamweaver.outputResultsPanel.clipPaste() 542
dreamweaver.outputResultsPanel.debugWindow.addDebugContextData() 544
dreamweaver.outputResultsPanel.openInBrowser() 542
dreamweaver.outputResultsPanel.openInEditor() 543
dreamweaver.outputResultsPanel.save() 543
dreamweaver.outputResultsPanel.selectAll() 543
dreamweaver.playRecordedCommand() 489
dreamweaver.popupAction() 383
dreamweaver.popupCommand() 659
dreamweaver.popupEditTagDialog() 609
dreamweaver.popupInsertTagDialog() 608
dreamweaver.popupServerBehavior() 551
dreamweaver.PrintCode() 534
dreamweaver.quitApplication() 481
dreamweaver.redo() 489
dreamweaver.referencePalette.getFontSize() 379
dreamweaver.referencePalette.setFontSize() 379
dreamweaver.refreshExtData() 183
dreamweaver.relativeToAbsoluteURL() 533
dreamweaver.releaseDocument() 456
dreamweaver.reloadMenus() 531
dreamweaver.removeAllBreakpoints() 501
dreamweaver.replace() 460
dreamweaver.replaceAll() 460
dreamweaver.resultsPalette.canClear() 426
dreamweaver.resultsPalette.canClipCopy() 427
dreamweaver.resultsPalette.canClipCut() 427
dreamweaver.resultsPalette.canClipPaste() 427
dreamweaver.resultsPalette.canOpenInBrowser() 427
dreamweaver.resultsPalette.canOpenInEditor() 428
dreamweaver.resultsPalette.canSave() 428
dreamweaver.resultsPalette.canSelectAll() 428
dreamweaver.revertDocument() 457
dreamweaver.runCommand() 400
dreamweaver.saveAll() 457
dreamweaver.saveDocument() 457
dreamweaver.saveDocumentAs() 458
dreamweaver.saveDocumentAsTemplate() 458
dreamweaver.saveFrameset() 459
dreamweaver.saveFramesetAs() 459
dreamweaver.saveSitesToPrefs() 558
dreamweaver.scanSourceString() 586
dreamweaver.selectAll() 550
dreamweaver.serverBehaviorInspector.getServerBehaviors() 551
dreamweaver.serverComponents.getSelectedNode() 401
dreamweaver.serverComponents.refresh() 401
dreamweaver.setActiveWindow() 647
dreamweaver.setFloaterVisibility() 647
dreamweaver.setHideAllFloaters() 634
dreamweaver.setLiveDataError() 528
dreamweaver.setLiveDataMode() 529
dreamweaver.setLiveDataParameters () 529
dreamweaver.setPreferenceInt() 486
dreamweaver.setPreferenceString() 487
dreamweaver.setPrimaryView() 648
dreamweaver.setSelection() 659
dreamweaver.setShowStatusBar() 634
dreamweaver.setSnapDistance() 649
dreamweaver.setUpComplexFind() 461
dreamweaver.setUpComplexFindReplace() 461
dreamweaver.setUpFind() 462
dreamweaver.setUpFindReplace() 463
dreamweaver.showAboutBox() 481
dreamweaver.showDynamicData() 481
dreamweaver.showFindDialog() 463
dreamweaver.showFindReplaceDialog() 464
dreamweaver.showGridSettingsDialog() 515
dreamweaver.showLiveDataDialog() 530
dreamweaver.showPreferencesDialog() 482
dreamweaver.showProperties() 649
dreamweaver.showQuickTagEditor() 536
dreamweaver.showReportsDialog() 536
dreamweaver.showTagChooser() 482, 609
dreamweaver.showTagLibraryEditor() 609
dreamweaver.snippetPalette.editSnippet() 583
dreamweaver.snippetPalette.insert() 583
dreamweaver.snippetPalette.insertSnippet() 584
dreamweaver.snippetPalette.newFolder() 583

dreamweaver.snippetPalette.remove() 584
dreamweaver.snippetPalette.rename() 584
dreamweaver.startDebugger() 501
dreamweaver.startRecording() 489
dreamweaver.stopRecording() 490
dreamweaver.stylePalette.attachExternalStylesheet() 404
dreamweaver.tagInspector.deleteTags() 613
dreamweaver.tagInspector.editTagName() 613
dreamweaver.tagInspector.tagAfter() 612
dreamweaver.tagInspector.tagBefore() 612
dreamweaver.tagInspector.tagInside() 612
dreamweaver.tagLibrary.getImportedTagList() 611
dreamweaver.tagLibrary.getSelectedLibrary() 610
dreamweaver.tagLibrary.getSelectedTag() 610
dreamweaver.tagLibrary.getTagLibraryDOM() 610
dreamweaver.tagLibrary.importDTDOrSchema() 611
dreamweaver.templatePalette object 521
dreamweaver.templatePalette.deleteSelectedTemplate() 659
dreamweaver.templatePalette.getSelectedTemplate() 658
dreamweaver.templatePalette.renameSelectedTemplate( ) 660
dreamweaver.timelineInspector object 614
dreamweaver.timelineInspector.addBehavior() 614
dreamweaver.timelineInspector.addFrame() 614
dreamweaver.timelineInspector.addKeyframe() 614
dreamweaver.timelineInspector.addObject() 615
dreamweaver.timelineInspector.addTimeline() 615
dreamweaver.timelineInspector.canAddFrame() 430
dreamweaver.timelineInspector.canAddKeyFrame() 431
dreamweaver.timelineInspector.canChangeObject() 431
dreamweaver.timelineInspector.canRemoveFrame() 432
dreamweaver.timelineInspector.canRemoveKeyFrame() 432
dreamweaver.timelineInspector.canRemoveObject() 432
dreamweaver.timelineInspector.changeObject() 615
dreamweaver.timelineInspector.getAutoplay() 615
dreamweaver.timelineInspector.getCurrentFrame() 616
dreamweaver.timelineInspector.getLoop() 616
dreamweaver.timelineInspector.recordPathOfLayer() 616
dreamweaver.timelineInspector.removeBehavior() 617
dreamweaver.timelineInspector.removeFrame() 617
dreamweaver.timelineInspector.removeKeyframe() 617
dreamweaver.timelineInspector.removeObject() 618
dreamweaver.timelineInspector.removeTimeline() 618
dreamweaver.timelineInspector.renameTimeline() 618
dreamweaver.timelineInspector.setAutoplay() 618
dreamweaver.timelineInspector.setCurrentFrame() 619
dreamweaver.timelineInspector.setLoop() 619
dreamweaver.toggleFloater() 650
dreamweaver.undo() 490
dreamweaver.updatePages() 526
dreamweaver.updateReference() 651
dreamweaver.useTranslatedSource() 641
dreamweaver.validateFlash() 447
dropdown tag 86
duplicateSelectedStyle() 405, 495
dw object 371
dw.arrangeFloatingPalettes() 515
dw.assetPalette.addToFavoritesFromDocument() 372
dw.assetPalette.addToFavoritesFromSiteAssets() 373
dw.assetPalette.addToFavoritesFromSiteWindow() 373
dw.assetPalette.canEdit() 418
dw.assetPalette.canInsertOrApply() 418
dw.assetPalette.copyToSite() 373
dw.assetPalette.edit() 374
dw.assetPalette.getSelectedCategory() 374
dw.assetPalette.getSelectedItems() 374
dw.assetPalette.getSelectedView() 375
dw.assetPalette.insertOrApply() 375
dw.assetPalette.locateInSite() 376
dw.assetPalette.newAsset() 376
dw.assetPalette.newFolder() 376
dw.assetPaletterecreateLibraryFromDocument() 377
dw.assetPalette.refreshSiteAssets() 377
dw.assetPalette.removeFromFavorites() 377
dw.assetPalette.renameNickname() 378
dw.assetPalette.setSelectedCategory() 378
dw.assetPalette.setSelectedView() 378
dw.beep() 480
dw.behaviorInspector.getBehaviorAt() 384
dw.behaviorInspector.getBehaviorCount() 384
dw.behaviorInspector.getSelectedBehavior() 384
dw.behaviorInspector.moveBehaviorDown() 385
dw.behaviorInspector.moveBehaviorUp() 386
dw.behaviorInspector.setSelectedBehavior() 387
dw.browseDocument() 441
dw.browseForFileURL() 449
dw.browseForFolderURL() 449
dw.canClipCopy() 418
dw.canClipCut() 419
dw.canClipPaste() 419
dw.canExportCSS() 420

dw.canExportTemplateDataAsXML() 420
dw.canFindNext() 420
dw.canOpenInFrame() 420
dw.canPlayRecordedCommand() 421
dw.canPopupEditTagDialog() 421
dw.canRedo() 421
dw.canRevertDocument() 422
dw.canSaveAll() 422
dw.canSaveDocument() 422
dw.canSaveDocumentAsTemplate() 423
dw.canSaveFrameset() 423
dw.canSaveFramesetAs() 423
dw.canSelectAll() 424
dw.canShowFindDialog() 424
dw.canUndo() 424
dw.clipCopy() 390
dw.clipCut() 391
dw.clipPaste() 391
dw.closeDocument() 450
dw.codeHints.addFunction() 398
dw.codeHints.addMenu() 397
dw.codeHints.resetMenu() 399
dw.codeHints.showCodeHints() 400
dw.createDocument() 450
dw.createResultsWindow() 537
dw.createXHTMLDocument() 451
dw.createXMLDocument() 452
dw.cssStyle.canEditSelectedStyle() 425
dw.cssStylePalette.applySelectedStyle() 404
dw.cssStylePalette.canApplySelectedStyle() 424
dw.cssStylePalette.canDeleteSelectedStyle() 425
dw.cssStylePalette.canDuplicateSelectedStyle() 425
dw.cssStylePalette.deleteSelectedStyle() 405
dw.cssStylePalette.duplicateSelectedStyle() 405
dw.cssStylePalette.editSelectedStyle() 405
dw.cssStylePalette.editStyleSheet() 406
dw.cssStylePalette.getSelectedStyle() 406
dw.cssStylePalette.getSelectedTarget() 406, 653
dw.cssStylePalette.getStyles() 407
dw.cssStylePalette.newStyle() 408
dw.cssStylePallette.canEditStyleSheet() 425
dw.dbi.getDataSources() 408
dw.debugDocument() 500
dw.deleteSelection() 477
dw.doURLDecoding() 484
dw.doURLEncoding() 585
dw.editCommandList() 400
dw.editFontList() 478
dw.editLockedRegions() 640
dw.exportCSS() 452
dw.exportEditableRegionsAsXML() 653
dw.exportTemplateDataAsXML() 452, 525
DWfile.copy() 271
DWfile.createFolder() 272
DWfile.exists() 272
DWfile.getAttributes() 273
DWfile.getCreationDate() 274
DWfile.getCreationDateObj() 275
DWfile.getModificationDate() 274
DWfile.getModificationDateObj() 275
DWfile.getSize() 276
DWfile.listFolder() 276
DWfile.read() 277
DWfile.remove() 277
DWfile.setAttributes() 278
DWfile.write() 278
dw.findNext() 460
dw.getActiveWindow() 644
dw.getBehaviorElement() 381
dw.getBehaviorTag() 382
dw.getBrowserList() 442
dw.getConfigurationPath() 532
dw.getDebugBrowserList() 500
dw.getDocumentDOM() 453
dw.getDocumentList() 644
dw.getDocumentPath() 532
dw.getElementRef() 484
dw.getExtDataArray 182
dw.getExtDataValue() 182
dw.getExtensionEditorList() 442
dw.getExternalTextEditor() 442
dw.getExtGroups() 183
dw.getExtParticipants() 182
dw.getFlashPath() 443
dw.getFloaterVisibility() 644
dw.getFocus() 645
dw.getFontList() 478
dw.getFontStyles() 478
dw.getHideAllFloaters() 630
dw.getIsAnyBreakpoints() 501
dw.getKeyState() 479
dw.getLiveDataInitTags() 526
dw.getLiveDataMode() 527
dw.getLiveDataParameters () 527
dw.getMenuNeedsUpdating() 530
dw.getParticipants() 155
dw.getPreferenceInt() 485
dw.getPreferenceString() 486
dw.getPrimaryBrowser() 443
dw.getPrimaryExtensionEditor() 443

dw.getPrimaryView() 646
dw.getRecentFileList() 454
dw.getRedoText() 488
dw.getSecondaryBrowser() 444
dw.getShowDialogsOnInsert() 480
dw.getShowStatusBar() 630
dw.getSiteRoot() 533
dw.getSnapDistance() 646
dw.getSystemFontList() 480
dw.getTokens() 585
dw.getTranslatorList() 641
dw.getUndoText() 488
dw.historyPalette.clearSteps() 490
dw.historyPalette.copySteps() 490
dw.historyPalette.getSelectedSteps() 491
dw.historyPalette.getStepCount() 492
dw.historyPalette.getStepsAsJavaScript() 492
dw.historyPalette.getUndoState() 493
dw.historyPalette.replaySteps() 493
dw.historyPalette.saveAsCommand() 493
dw.historyPalette.setSelectedSteps() 494
dw.historyPalette.setUndoState() 494
dw.htmlInspector.getShowAutoIndent() 631
dw.htmlInspector.getShowHighlightInvalidHTML() 631
dw.htmlInspector.getShowLineNumbers() 631
dw.htmlInspector.getShowSyntaxColoring() 631
dw.htmlInspector.getShowWordWrap() 632
dw.htmlInspector.setShowAutoIndent() 632
dw.htmlInspector.setShowHighlightInvalidHTML() 632
dw.htmlInspector.setShowLineNumbers() 633
dw.htmlInspector.setShowSyntaxColoring() 633
dw.htmlInspector.setShowWordWrap() 633
dw.htmlStylePalette.canEditSelection() 426
dw.htmlStylePalette.deleteSelectedStyle() 495
dw.htmlStylePalette.duplicateSelectedStyle() 495
dw.htmlStylePalette.editSelectedStyle() 496
dw.htmlStylePalette.getSelectedStyle() 496
dw.htmlStylePalette.getStyles() 496
dw.htmlStylePalette.newStyle() 496
dw.htmlStylePalette.setSelectedStyle() 497
dw.importXMLIntoTemplate() 454
dw.isRecording() 426
dw.isReporting() 536
dw.latin1ToNative() 586
dw.libraryPalette.deleteSelectedItem() 656
dw.libraryPalette.getSelectedItem() 656
dw.libraryPalette.newFromDocument() 657
dw.libraryPalette.recreateFromDocument() 657
dw.libraryPalette.renameSelectedItem() 657
dw.liveDataTranslate() 528
dw.loadSitesFromPrefs() 558
dw.minimizeRestoreAll() 647
dw.nativeToLatin1() 586
dw.newDocumentDOM() 454
dw.newFromTemplate() 455
dw.nodeExists() 549
dw.notifyMenuUpdated() 531
dw.openDocument() 455
dw.openDocumentFromSite() 455
dw.openInFrame() 456
dw.openWithApp() 445
dw.openWithBrowseDialog() 446
dw.openWithExternalTextEditor() 446
dw.openWithImageEditor() 446
dw.outputResultsPanel.clearItems() 541
dw.outputResultsPanel.clipCopy() 541
dw.outputResultsPanel.clipCut() 542
dw.outputResultsPanel.clipPaste() 542
dw.outputResultsPanel.debugWindow.addDebugContextData() 544
dw.outputResultsPanel.openInBrowser() 542
dw.outputResultsPanel.openInEditor() 543
dw.outputResultsPanel.save() 543
dw.outputResultsPanel.selectAll() 543
dw.playRecordedCommand() 489
dw.popupAction() 383
dw.popupEditTagDialog() 609
dw.popupInsertTagDialog() 608
dw.popupServerBehavior() 551
dw.PrintCode() 534
dw.quitApplication() 481
dw.redo() 489
dw.referencePalette.getFontSize() 379
dw.referencePalette.setFontSize() 379
dw.refreshExtData() 183
dw.relativeToAbsoluteURL() 533
dw.releaseDocument() 456
dw.reloadMenus() 531
dw.removeAllBreakpoints() 501
dw.replace() 460
dw.replaceAll() 460
dw.resultsPalette.canClear() 426
dw.resultsPalette.canClipCopy() 427
dw.resultsPalette.canClipCut() 427
dw.resultsPalette.canClipPaste() 427
dw.resultsPalette.canOpenInBrowser() 427
dw.resultsPalette.canOpenInEditor() 428
dw.resultsPalette.canSave() 428

dw.resultsPalette.canSelectAll() 428
dw.revertDocument() 457
dw.runCommand() 400
dw.saveAll() 457
dw.saveDocument() 457
dw.saveDocumentAs() 458
dw.saveDocumentAsTemplate() 458
dw.saveFrameset() 459
dw.saveFramesetAs() 459
dw.saveSitesToPrefs() 558
dw.scanSourceString() 586
dwscripts functions
  applySB() 157
  deleteSB() 157
  findSBs() 156
dw.selectAll() 550
dw.serverBehaviorInspector.getServerBehaviors() 551
dw.serverComponents.getSelectedNode() 401
dw.serverComponents.refresh() 401
dw.setActiveWindow() 647
dw.setFloaterVisibility() 647
dw.setHideAllFloaters() 634
dw.setLiveDataError() 528
dw.setLiveDataMode() 529
dw.setLiveDataParameters () 529
dw.setPreferenceInt() 486
dw.setPreferenceString() 487
dw.setPrimaryView() 648
dw.setShowStatusBar() 634
dw.setSnapDistance() 649
dw.setUpComplexFind() 461
dw.setUpComplexFindReplace() 461
dw.setUpFind() 462
dw.setUpFindReplace() 463
dw.showAboutBox() 481
dw.showDynamicData() 481
dw.showFindDialog() 463
dw.showFindReplaceDialog() 464
dw.showGridSettingsDialog() 515
dw.showLiveDataDialog() 530
dw.showPreferencesDialog() 482
dw.showProperties() 649
dw.showQuickTagEditor() 536
dw.showReportsDialog() 536
dw.showTagChooser() 482, 609
dw.showTagLibraryEditor() 609
dw.snippetPalette.editSnippet() 583
dw.snippetPalette.insert() 583
dw.snippetPalette.insertSnippet() 584
dw.snippetPalette.newFolder() 583
dw.snippetPalette.remove() 584
dw.snippetPalette.rename() 584
dw.startDebugger() 501
dw.startRecording() 489
dw.stopRecording() 490
dw.stylePalette.attachExternalStylesheet() 404
dw.tagInspector.deleteTags() 613
dw.tagInspector.editTagName() 613
dw.tagInspector.tagAfter() 612
dw.tagInspector.tagBefore() 612
dw.tagInspector.tagInside() 612
dw.tagLibrary.getImportedTagList() 611
dw.tagLibrary.getSelectedLibrary() 610
dw.tagLibrary.getSelectedTag() 610
dw.tagLibrary.getTagLibraryDOM() 610
dw.tagLibrary.importDTDOrSchema() 611
dw.templatePalette.deleteSelectedTemplate() 659
dw.templatePalette.getSelectedTemplate() 658
dw.templatePalette.renameSelectedTemplate() 660
dw.timelineInspector.addBehavior() 614
dw.timelineInspector.addFrame() 614
dw.timelineInspector.addKeyframe() 614
dw.timelineInspector.addObject() 615
dw.timelineInspector.addTimeline() 615
dw.timelineInspector.canAddFrame() 430
dw.timelineInspector.canAddKeyFrame() 431
dw.timelineInspector.canChangeObject() 431
dw.timelineInspector.canRemoveFrame() 432
dw.timelineInspector.canRemoveKeyFrame() 432
dw.timelineInspector.canRemoveObject() 432
dw.timelineInspector.changeObject() 615
dw.timelineInspector.getAutoplay() 615
dw.timelineInspector.getCurrentFrame() 616
dw.timelineInspector.getLoop() 616
dw.timelineInspector.recordPathOfLayer() 616
dw.timelineInspector.removeBehavior() 617
dw.timelineInspector.removeFrame() 617
dw.timelineInspector.removeKeyframe() 617
dw.timelineInspector.removeObject() 618
dw.timelineInspector.removeTimeline() 618
dw.timelineInspector.renameTimeline() 618
dw.timelineInspector.setAutoplay() 618
dw.timelineInspector.setCurrentFrame() 619
dw.timelineInspector.setLoop() 619
dw.toggleFloater() 650
dw.undo() 490
dw.updatePages() 526
dw.updateReference() 651
dw.useTranslatedSource() 641
dw.validateFlash() 447

Dynamic Data dialog box 191
dynamic menus
  sample code 75
  user experience 67
dynamic templates 27
Dynamic Text dialog box 191

E

Edit Format List Plus (+) pop-up menu 201
editAttribute() 467
editColumns() 563
editCommandList() 400
editcontrol tag 87
editDynamicSource() 194
editFontList() 478
editLockedRegions() 640
editSelectedStyle() 405, 496
editStyleSheet() 406
EDML
  definition 145
EDML file tags
  attributes 180
  closeTag 181
  dataSource attribute 161
  delete 175
  deleteType attribute 176
  display 181
  group 160
  groupParticipant tag 163
  groupParticipants 162
  insertText 166
  isOptional attribute 172
  limitSearch attribute 172, 178
  location attribute 166
  name attribute 163
  nodeParamName attribute 168
  openTag 179
  paramName attribute 175
  paramNames attribute 171
  participant 165
  partType attribute 164
  quickSearch 165
  searchPatterns 169, 177
  selectParticipant attribute 163
  serverBehavior attribute 160
  subType attribute 161
  title 162
  translation 177
  translations 177
  translationType attribute 178
  translator 176
  updatePattern 174
  updatePatterns 173
  version attribute 160, 165
  whereToSearch attribute 169, 178
EDML files 146
  editing 158
  EDML structure 159
  group file tags 160
  using regular expressions 158
element node 45
enabled attribute 90
enablers
  return value 409
  using 372
endOfDocument() 504, 592
endOfLine() 504, 592
endPage() 593
endReporting() 105
errata 12
escape() 42
event handlers
  in behavior dialog boxes 135
  in extension files 21
  returning a value from 142
events
  in extension files 42
  role in behaviors 135
execJsInFireworks() 300
exists() 272
exitBlock() 467
exportCSS() 452
exportEditableRegionsAsXML() 653
exportSite() 563
exportTemplateDataAsXML() 452, 525
extensible document types 22
extension
  user interface 31
extension APIs, types of 20
Extension Data Manager 181
Extension Data Markup Language (EDML) 146
extension folders 18
Extension Manager
  guidelines 31
  working with 22
extensions, Dreamweaver 17
Extensions.txt file 28
external
  application functions 441
  JavaScript files 21

F

file (field) object 42
file attribute 90
file I/O API 271
  DWfile.copy() 271
  DWfile.createFolder() 272
  DWfile.exists() 272
  DWfile.getAttributes() 273
  DWfile.getCreationDate() 274
  DWfile.getCreationDateObj() 275
  DWfile.getModificationDate() 274
  DWfile.getModificationDateObj() 275
  DWfile.getSize() 276
  DWfile.listFolder() 276
  DWfile.read() 277
  DWfile.remove() 277
  DWfile.setAttributes() 278
  DWfile.write() 278
file manipulation functions
  in JavaScript API 447
FilePathToLocalURL() 293
files
  CodeHints.xml 392
  insertbar.xml 51
  snippets 582
  toolbars.xml 77
  XML 42
files on disk
  copying 271
  creating (HTML files) 450
  creating (non-HTML files) 278
  creating (XHTML files) 451
  creating (XML files) 452
  reading 277
  removing 277
  writing to 278
findConnection() 339
findDynamicSources() 194
findLinkSource() 565
findNext() 460
findSBs() 156
Fireworks integration API 299
  bringDWToFront() 299
  bringFWToFront() 299
  example 304
  execJsInFireworks() 300
  getJsResponse() 300
  mayLaunchFireworks() 301
  optimizeInFireworks() 302
  validateFireworks() 302
Flash integration 16
Flash object API
  SWFFile.createFile() 307
  SWFFile.getNaturalSize() 309
  SWFFile.getObjectType() 309
  SWFFile.readFile() 309
Flash objects
  creating 307
floating panel API
  displayHelp() 127
  documentEdited() 127
  getDockingSide() 128
  initialPosition() 128
  initialTabs() 129
  isATarget() 129
  isAvailableInCodeView() 130
  isResizable() 130
floating panels
  API 126
  performance issues 131
  sample code 133
  selectionChanged() 130
  user experience 125
focus() 42
folders
  objects 51
form object 42
formatDynamicDataRef() 203
format.Range() 588
formats 199
formatSelection() 589
FTP logging 541
function
  object 42
function tag 396
FWLaunch.bringDWToFront() 299
FWLaunch.bringFWToFront() 299
FWLaunch.execJsInFireworks() 300
FWLaunch.getJsResponse() 300
FWLaunch.mayLaunchFireworks() 301
FWLaunch.optimizeInFireworks() 302
FWLaunch.validateFireworks() 302

G

generateDynamicDataRef() 195
generateDynamicSourceBindings() 196
get() 565
getActiveWindow() 644
getAppServerAccessType() 565
getAppServerPathToFiles() 566
getAttachedTemplate() 522

getAttribute() 45
getAttributes() 273
getAutoplay() 615
getBehavior() 380
getBehaviorAt() 384
getBehaviorCount() 384
getBehaviorElement() 381
getBehaviorEvent() 653
getBehaviorTag() 382
getBreakpoint() 498
getBrowserList() 442
getCharSet() 467
getCheckOutUser() 566
getCheckOutUserForFile() 567
getClasses() 346
getClassesFromPackage() 347
getClickedHeaderColumn() 518
getClipboardText() 391
getCloakingEnabled() 567
getCodeViewDropCode() 212
getColdFusionDsnList() 313
getColumnAndTypeList() 325
getColumnList() 325
getColumns() 326
getColumnsOfTable() 327
getComponentChildren() 208
getConfigurationPath() 532
getConnection() 313
getConnectionList() 314
getConnectionName() 315
getConnectionState() 567
getConnectionString() 315
getContextMenuId() 210
getCreationDate() 274
getCreationDateObj() 275
getCurrentFrame() 616
getCurrentLines() 593
getCurrentSite() 568
getCurrentValue() 94
getDataSources() 408
getDebugBrowserList() 500
getDelimiters() 553
getDockingSide() 128
getDocumentDOM() 453
getDocumentList() 644
getDocumentPath() 532
getDriverName() 316
getDriverUrlTemplateList() 316
getDynamicContent() 69, 94
getEditableRegionList() 522
getEditableRetionList() 523
getEditNoFramesContent() 620
getElementRef() 484
getElementsByTagName()
  for document objects 44
  for tag objects 45
getErrorMessage() 347
getEvents() 346
getExtDataArray() 182
getExtDataValue() 182
getExtensionEditorList() 442
getExternalTextEditor() 442
getExtGroups() 183
getExtParticipants() 182
getFile() 282
getFileCallback() 284
getFileExtensions() 218
getFlashPath() 443
getFloaterVisibility() 644
getFocus() 568, 642, 645
getFontList() 478
getFontMarkup() 468
getFontStyles() 478
getFrameNames() 464
getHideAllFloaters() 630
getHideAllVisualAids() 620
getImportedTagList() 611
getIndexedProperties() 346
getIsAnyBreakpoints() 501
getIsLibraryDocument() 523
getIsTemplateDocument() 523
getIsXHTMLDocument() 449
getJsResponse() 300
getKeyState() 479
getLanguageSignatures() 219
getLineFromOffset() 498, 594
getLinkHref() 468
getLinkTarget() 468
getLinkVisibility() 568
getListTag() 468
getLiveDataInitTags() 526
getLiveDataMode() 527
getLiveDataParameters () 527
getLocalDsnList() 317
getLocalPathToFiles() 569
getLoop() 616
getMenuID() 95
getMenuNeedsUpdating() 530
getMethods() 345
getModificationDate() 274

getModificationDateObj() 275
GetNote() 293
GetNoteLength() 294
GetNotesKeyCount() 294
GetNotesKeys() 294
getObjectRefs() 654
getObjectTags() 655
getParticipants() 155
getPassword() 317
getPreferenceInt() 485
getPreferenceString() 486
getPreventLayerOverlaps() 620
getPrimaryBrowser() 443
getPrimaryExtensionEditor() 443
getPrimaryKeys() 327
getPrimaryView() 646
getProcedures() 328
getProperties() 345
getRdsPassword() 318
getRdsUserName() 318
getRecentFileList() 454
getRedoText() 488
getRemoteDsnList() 318
getRulerOrigin() 511
getRulerUnits() 511
getRuntimeConnectionType() 319
getSecondaryBrowser() 444
getSelectedBehavior() 384
getSelectedEditableRegion() 523
getSelectedItem() 656
getSelectedLibrary() 610
getSelectedNode() 401, 546
getSelectedStyle() 406, 496
getSelectedTag() 610
getSelectedTarget() 406, 653
getSelectedTemplate() 658
getSelection() 546, 569, 593
  dreamweaver.getSelection() 656
getServerBehaviors() 551
getServerExtension() 219, 554
getServerInfo() 220
getServerLanguage() 556
getServerLanguages() 220
getServerModelDelimiters() 221
getServerModelDisplayName() 222
getServerModelExtDataNameUD4() 221
getServerModelFolderName() 222
getServerName() 556
getServerSupportsCharset() 222
getServerVersion() 557
getSetupSteps() 212
getShowAutoIndent() 620
getShowDependents() 634
getShowDialogsOnInsert() 480
getShowFrameBorders() 621
getShowGrid() 621
getShowHeadView() 621
getShowHiddenFiles() 634
getShowImageMaps() 622
getShowInvalidHTML() 622
getShowInvisibleElements() 622
getShowLayerBorders() 622, 628
getShowLayoutTableTabs() 518
getShowLayoutView() 518
getShowLineNumbers() 623
getShowNoscript 589
getShowPageTitles() 635
getShowRulers() 623
getShowStatusBar() 630
getShowSyntaxColoring() 623
getShowTableBorders() 623
getShowToolbar() 624
getShowToolbarIconLabels() 640
getShowToolTips() 635
getShowTracingImage() 624
getShowWordWrap() 624
getSiteForURL() 569
getSiteRoot() 533
GetSiteRootForFile() 295
getSites() 570
getSize() 276
getSnapDistance() 646
getSnapToGrid() 625
getSPColumnList() 329
getSPColumnListNamedParams() 329
getSPParameters() 330
getSPParamsAsString() 331
getStepCount() 492
getStepsAsJavaScript() 492
getStyles() 407, 496
getSystemFontList() 480
getTableExtent() 604
getTables() 332
getTagLibraryDOM() 610
getTagSelectorTag() 608
getText() 284, 594
getTextAlignment() 469
getTextCallback() 285
getTextFormat() 469
getTokens() 585

getToolbarIdArray() 638
getToolbarLabel() 639
getToolbarVisibility() 637
getTracingImageOpacity() 511
getTranslatedAttribute() 45
getTranslatorInfo() 226
getTranslatorList() 641
getUndoState() 493
getUndoText() 488
getUpdateFrequency() 96
getUserName() 319
getVersionArray() 223
GetVersionName() 296
GetVersionNum() 296
getView() 642
getViews() 332
getWindowTitle() 643
global application functions 480
group file tags 160
group files 146
groupParticipant 163
groupParticipants tag 162

H

handleDoubleClick() 213
hasCharacterMarkup() 469
hasChildNodes()
  for comment objects 46
  for document objects 44
  for tag objects 45
  for text objects 46
hasConnectionWithName() 320
hasTracingImage() 418
hasTranslatedAttributes() 45
Help Book extensions
  definition 20
helper functions, in behaviors 136
hidden (field) object 42
history functions 487
hline 119
hotspot functions 508
HTML
  apply style 495
  Cascading Style Sheets 402
  converting to XHTML 448
  creating new document 450
  inner/outer properties 45
  inserting 470
  show invalid 622, 631
  style palette object 495
htmlInspector.getShowAutoIndent() 631
htmlInspector.getShowHighlightInvalidHTML() 631
htmlInspector.getShowLineNumbers() 631
htmlInspector.getShowSyntaxColoring() 631
htmlInspector.getShowWordWrap() 632
htmlInspector.setShowAutoIndent() 632
htmlInspector.setShowHighlightInvalidHTML() 632
htmlInspector.setShowLineNumbers() 633
htmlInspector.setShowSyntaxColoring() 633
htmlInspector.setShowWordWrap() 633
HTTP API 281
  MMHttp.clearTemp() 282
  MMHttp.getFile() 282
  MMHttp.getFileCallback() 284
  MMHttp.getText() 284
  MMHttp.getTextCallback() 285
  MMHttp.postText() 286
  MMHttp.postTextCallback() 286

I

id attribute 88
identifyBehaviorArguments() 140
image
  attributes 88
  map functions 508
  object 42
image map functions 508
importDTDOrSchema() 611
importSite() 570
importXMLIntoTemplate() 454
include files
  connection type definition 343
include files, generated 341
include/ tag 82
increaseColspan() 604
increaseRowspan() 604
indent() 470
indentTextView() 594
InfoPrefs 295
initialPosition() 128
initialTabs() 129
innerHTML property 45
Insert bar
  adding objects 52
  defining 52
  insertbar xml tag 54
Insert menu
  adding objects 56
insert() 595
insertbar.xml file 51
insertHTML() 470
insertLibraryItem() 524

insertObject() 51, 58, 471
insertTableColumns() 605
insertTableRows() 605
insertText tag 166
insertText() 471
inspectBehavior() 141
inspectConnection() 341
inspectDynamicDataRef() 197
inspectFormatDefinition() 204
Inspector extensions, definition 20
inspectTag() 117
installing an extension 11
instrumentDocument() 243, 498
invertSelection() 570
isATarget() 129
isAvailableInCodeView() 130
isCloaked() 571
isColumnAutostretch() 518
isCommandChecked() 70, 97
isDesignViewUpdated() 589
isDocumentInFrame() 465
isDOMRequired() 98
isDomRequired() 58
isOptional attribute 172
isRecording() 426
isReporting() 536
isResizable() 130
isSelectionValid() 590
item tags, in toolbars 83
item() 42
itemInfo struct 357
itemref/ tag 82
itemtype/ tag 82

J

JavaBeans API 345
  MMJB.getClasses() 346
  MMJB.getClassesFromPackage() 347
  MMJB.getErrorMessage() 347
  MMJB.getEvents() 346
  MMJB.getIndexedProperties() 346
  MMJB.getMethods() 345
  MMJB.getProperties() 345
JavaScript
  controls 33
  URLs 21
JavaScript (core) API 371
JavaScript debugger module API 245
  getBodyInstrument() 246
  getFunctionEndInstrument() 245
  getFunctionStartInstrument() 246
  getHeadInstrument() 246
  getIncludedFileList() 247
  getOnUnloadInstrument() 247
  getStepInstrument() 248
  reportError() 248
  reportWarning() 249
  startBlock() 249
JDBC drivers 316
JS_BooleanToValue() 257
JS_DefineFunction() 254
JS_DoubleToValue() 257
JS_ExecuteScript() 259
JS_GetArrayLength() 258
JS_GetElement() 259
JS_IntegerToValue() 257
JS_NewArrayObject() 258
JS_ObjectToValue() 257
JS_ObjectType() 258
JS_ReportError() 260
JS_SetElement() 259
JS_StringToValue() 256
JS_ValueToBoolean() 256
JS_ValueToDouble() 255
JS_ValueToInteger() 255
JS_ValueToObject() 256
JS_ValueToString() 255
JSBool 254
JSContext 253
JSNative 254
JSObject 253
jsval 253

L

label attribute 89
language information 47
latin1ToNative() 586
layer object 42
layers to tables 402
library and template functions 521
limitSearch attribute 172, 178
link-checking 541
listFolder() 276
live data functions 526
liveDataTranslate() 528
liveDataTranslateMarkup function() 228
loadSitesFromPrefs() 558
loadTracingImage() 512
localized strings 24
LocalURLToFilePath() 296
locateInSite() 571
location attribute 166

locked content, inspecting 239
LOCKED keyword 239

M

makeCellWidthsConsistent() 519
makeEditable() 571
makeNewDreamweaverFile() 572
makeNewFolder() 572
makeSizesEqual() 509
manipulating tree control content 39
markSelectionAsEditable() 524
math object 42
mayLaunchFireworks() 301
menu command
  API 68
  sample code 73
  user experience 67
menu tag 395
menu_ID attribute 90
menubutton tag 85
menugroup tag 394
menuitem tag 395
MENU-LOCATION 150
mergeTableCells() 606
minimizeRestoreAll() 647
MM
  TREECOLUMN 38
  TREENODE 38
MM_ConfigFileExists() 263
MM_GetConfigFileAttributes() 264
MM_GetConfigFolderList() 262
mm_jsapi.h file
  including 253
  sample 267
MM_OpenConfigFile() 263
MM_returnValue 142
MMDB.deleteConnection() 312
MMDB.getColdFusionDsnList() 313
MMDB.getColumnAndTypeList() 325
MMDB.getColumnList() 325
MMDB.getColumns() 326
MMDB.getColumnsOfTable() 327
MMDB.getConnection() 313
MMDB.getConnectionList() 314
MMDB.getConnectionName() 315
MMDB.getConnectionString() 315
MMDB.getDriverName() 316
MMDB.getDriverUrlTemplateList() 316
MMDB.getLocalDsnList() 317
MMDB.getPassword() 317
MMDB.getPrimaryKeys() 327
MMDB.getProcedures() 328
MMDB.getRdsPassword() 318
MMDB.getRdsUserName() 318
MMDB.getRemoteDsnList() 318
MMDB.getRuntimeConnectionType() 319
MMDB.getSPColumnList() 329
MMDB.getSPColumnListNamedParams() 329
MMDB.getSPParameters() 330
MMDB.getSPParamsAsString() 331
MMDB.getTables() 332
MMDB.getUserName() 319
MMDB.getViews() 332
MMDB.hasConnectionWithName() 320
MMDB.needToPromptForRdsInfo() 320
MMDB.needToRefreshColdFusionDsnList() 320
MMDB.popupConnection() 321
MMDB.setRdsPassword() 321
MMDB.setRdsUserName() 322
MMDB.showColdFusionAdmin() 322
MMDB.showConnectionMgrDialog() 322
MMDB.showOdbcDialog() 323
MMDB.showRdsUserDialog() 323
MMDB.showRestrictDialog() 323
MMDB.showResultset() 333
MMDB.showSPResultset() 334
MMDB.showSPResultsetNamedParams() 334
MMDB.testConnection() 324
MMDocumentTypes.xml document type definition file 23
MMHttp.clearTemp() 282
MMHttp.getFile() 282
MMHttp.getFileCallback() 284
MMHttp.getText() 284
MMHttp.getTextCallback() 285
MMHttp.postText() 286
MMHttp.postTextCallback() 286
MMJB*() functions 345
MMJB.getClasses() 346
MMJB.getClassesFromPackage() 347
MMJB.getErrorMessage() 347
MMJB.getEvents() 346
MMJB.getIndexedProperties() 346
MMJB.getMethods() 345
MMJB.getProperties() 345
MMNotes object 288
MMNotes.close() 288
MMNotes.filePathToLocalURL() 288
MMNotes.get() 288
MMNotes.getKeyCount() 289
MMNotes.getKeys() 289

MMNotes.getSiteRootForFile() 290
MMNotes.getVersionName() 290
MMNotes.getVersionNum() 290
MMNotes.localURLToFilePath() 290
MMNotes.open() 291
MMNotes.remove() 291
MMNotes.set() 292
moveBehaviorDown() 385
moveBehaviorUp() 386
moveSelectionBy() 509
multiple configurations 14
multiuser platforms
  Configuration folder 28

N

name attribute 163, 582
nativeToLatin1() 586
navigator object 42
needToPromptForRdsInfo() 320
needToRefreshColdFusionDsnList() 320
new features 13
newBlock() 472
newDocumentDOM() 454
newEditableRegion() 524
newFromDocument() 657
newFromTemplate() 455
newHomePage() 572
newSite() 573
newStyle() 408, 496
nextParagraph() 505
nextWord() 505, 595
node constants 42
Node.COMMENT_NODE 42
Node.DOCUMENT_NODE 42
Node.ELEMENT_NODE 42
nodeExists() 549
nodelist object 42
nodeParamName attribute 168
nodes 42
Node.TEXT_NODE 42
nodeToOffsets() 546
  dreamweaver.nodeToOffsets() 658
nodeType property
  of comment objects 46
  of document objects 44
  of tag objects 45
  of text objects 46
_notes folder 287
notifyFlashObjectChanged() 472
notifyMenuUpdated() 531
number object 42

O

object extensions
  definition 20
object object 42
objects
  adding to Insert bar 52
  adding to Insert menu 56
  API 57
  folder 51
  how files work 51
  user experience 51
objects API
  canInsertObject() 57
  displayHelp() 57
  insertObject() 58
  isDomRequired 58
  windowDimensions() 60
objectTag() 51, 59
offsetsToNode() 547
  dreamweaver.offsetsToNode() 658
onBlur 42
onChange 42
onClick 42
onFocus 42
onLoad 42
onMouseOver event 42
onResize 42
open() 291, 573
openDocument() 455
openDocumentFromSite() 455
openInBrowser() 542
openInEditor() 543
openInFrame() 456
OpenNotesFile() 297
OpenNotesFilewithOpenFlags() 297
openTag attribute 179
openWithApp() 445
openWithBrowseDialog() 446
openWithExternalTextEditor() 446
openWithImageEditor() 446
operating system, user's 47
optimizeInFireworks() 302
option object 42
outdent() 473
outdentTextView() 595
outerHTML property 45
outputResultsPanel.clearItems() 541
outputResultsPanel.clipCopy() 541
outputResultsPanel.clipCut() 542
outputResultsPanel.clipPaste() 542

outputResultsPanel.debugWindow.addDebugContext Data() 544
outputResultsPanel.openInBrowser() 542
outputResultsPanel.openInEditor() 543
outputResultsPanel.save() 543
outputResultsPanel.selectAll() 543
overImage attributes 89

P

pageDown() 505, 596
pageUp() 506, 596
Panel extensions
  definition 20
paramName attribute 175
paramNames attribute 171
parentNode property
  of comment objects 46
  of document objects 44
  of tag objects 45
  of text objects 46
parentWindow property 44
participant files 146
participant tag 165
participants 145
partType attribute 164
password (field) object 42
passwords, database connection 317
playAllPlugins() 512
playPlugin() 512
playRecordedCommand() 489
popupAction() 383
popupCommand() 659
popupConnection() 321
popupEditTagDialog() 609
popupInsertTagDialog() 608
popupServerBehavior() 551
postText() 286
postTextCallback() 286
preview attribute 582
previousParagraph() 506
previousWord() 507, 596
PrintCode() 534
processFile() 104
Property inspector API 121
  canInspectSelection() 121
  displayHelp() 121
  inspectSelection() 122
property inspectors
  LOCKED keyword 239
  comment at top of file 119
  custom 119
  file structure 119
  for locked content 239
  lightning bolt icon 233
  overview 119
  sample code 123
  translated attributes in 233
  user experience 120
put() 573

Q

quickSearch tag 165, 183
quitApplication() 481

R

radio object 42
radiobutton tag 85
read() 277
reapplyBehaviors() 381
receiveArguments() 99
  in menu commands 70
recordPathOfLayer() 616
recreateCache() 574
recreateFromDocument() 657
redo() 487, 489
referencePalette.getFontSize() 379
referencePalette.setFontSize() 379
refresh() 401, 574
refreshExtData() 183
regexp object 42
regular expressions in EDML files 158
relativeToAbsoluteURL() 533
releaseDocument() 456
reloadMenus() 531
Remote Development Services (RDS) 318
remoteIsValid() 574
remove() 277, 291
removeAllBreakpoints() 501
removeAllSpacers() 519
removeAllTableHeights() 606
removeAllTableWidths() 606
removeAttribute() 45
removeBehavior() 381, 617
removeCharacterMarkup() 473
removeCSSStyle() 403
removeEditableRegion() 525
removeFontMarkup() 473
removeFrame() 617
removeKeyframe() 617
removeLink() 474, 575
RemoveNote() 297
removeObject() 618

removeSpacerFromColumn() 519
removeTimeline() 618
renameSelectedItem() 657
renameSelectedTemplate() 660
renameSelection() 575
renameTimeline() 618
replace() 460
replaceAll() 460
replaceRange() 597
replaySteps() 493
report API
  beginReporting() 105
  commandButtons() 105
  configureSettings() 106
  endReporting() 105
  processfile() 104
  windowDimensions() 106
reports 541
  site 103
  stand-alone 104
reset object 42
resizeSelection() 474
resizeSelectionBy() 509
resizeTo() 42
Results window functions 537
resultsPalette.canClear() 426
resultsPalette.canClipCopy() 427
resultsPalette.canClipCut() 427
resultsPalette.canClipPaste() 427
resultsPalette.canOpenInBrowser() 427
resultsPalette.canOpenInEditor() 428
resultsPalette.canSave() 428
resultsPalette.canSelectAll() 428
resWin.addItem() 537
resWin.addResultItem() 538
resWin.setCallbackCommands() 539
resWin.setColumnWidths() 539
resWin.setFileList() 539
resWin.setTitle() 540
resWin.startProcessing() 540
resWin.stopProcessing() 540
revertDocument() 457
runCommand() 400
runTranslator() 640
runValidation() 483, 575

S

save() 543
saveAll() 457
saveAllFrames() 465
saveAsCommand() 493
saveAsImage() 576
saveDocument() 457
saveDocumentAs() 458
saveDocumentAsTemplate() 458
saveFrameset() 459
saveFramesetAs() 459
saveSitesToPrefs() 558
scanSourceString() 586
SCRIPTING-LANGUAGE statement 220
scrollEndFile() 597
scrollLineDown() 597
scrollLineUp() 598
scrollPageDown() 598
scrollPageUp() 598
scrollTopFile() 599
SCS 364
SCS_AfterPut() 367, 368
SCS_BeforeGet() 366
SCS_BeforePut() 367
SCS_canCheckin() 364
SCS_canCheckout() 364
SCS_canConnect() 363
SCS_canDelete() 366
SCS_canGet() 363
SCS_canNewFolder() 365
SCS_canPut() 364
SCS_canRename() 366
SCS_CanUndoCheckout() 365
SCS_Checkin() 357
SCS_Checkout() 358
SCS_Connect() 350
SCS_Delete() 354
SCS_Disconnect() 351
SCS_Get() 353
SCS_GetAgentInfo() 350
SCS_GetCheckoutName() 357
SCS_GetConnectionInfo() 355
SCS_GetDesignNotes() 361
SCS_GetErrorMessage() 360
SCS_GetErrorMessageLength() 360
SCS_GetFileCheckoutList() 359
SCS_GetFolderList() 352
SCS_GetFolderListLength() 352
SCS_GetMaxNoteLength() 361
SCS_GetNewFeatures() 356
SCS_GetNoteCount() 360
SCS_GetNumCheckedOut() 359
SCS_GetNumNewFeatures() 356
SCS_GetRootFolder() 352
SCS_GetRootFolderLength() 351

SCS_IsConnected() 351
SCS_IsRemoteNewer() 362
SCS_ItemExists() 355
SCS_NewFolder() 354
SCS_Put() 353
SCS_Rename() 354
SCS_SetDesignNotes() 362
SCS_SiteDeleted() 356
SCS_SiteRenamed() 356
SCS_UndoCheckout() 358
search pattern resolution 187
searches 541
searchPatterns tag 169, 177
select color control for Javascript extensions 40
select object 42
select() 42
selectAll() 543, 548, 550, 576
selectChild() 534
selectHomePage() 576
selection functions 546
selection, exact vs. within 119
selectionChanged() 130
selectNewer() 577
selectParent() 535
selectParentTag() 599
selectParticipant attribute 163
selectTable() 548
separator tag 54, 83
server
  components functions 401
  debugging 544
server behavior
  deleting 189
  dwscripts functions 156
  example 147
  extension 145
  finding 183
  group files 146
  instance 145
  overview 145
  participant files 146
  participants 145
  runtime code 145
  search pattern resolution 187
  techniques 183
  updating 188
server behavior API 151
  analyzeServerBehavior() 151
  applyServerBehavior() 152
  canApplyServerBehavior() 152
  copyServerBehavior() 153
  deleteServerBehavior() 153
  displayHelp() 154
  findServerBehaviors() 154
  inspectServerBehavior() 154
  pasteServerBehavior() 155
Server Behavior extensions, definition 20
server model API 217
  canRecognizeDocument() 217
  getFileExtensions() 218
  getLanguageSignatures() 219
  getServerExtension() 219
  getServerInfo() 220
  getServerLanguages() 220
  getServerModelDelimiters() 221
  getServerModelDisplayName() 222
  getServerModelExtDataNameUD4() 221
  getServerModelFolderName() 222
  getServerSupportsCharset() 222
  getVersionArray() 223
server models
  definition 217
  extensibility 15
server models extensions
  definition 21
serverBehavior attribute 160
serverdebuginfo 544
service component, adding 206
set() 292
setActiveWindow() 647
setAsHomePage() 577
setAttribute() 45
setAttributes() 278
setAttributeWithErrorChecking() 474
setAutoplay() 618
setBreakpoint() 499
setCallbackCommands() 539
setCloakingEnabled() 577
setColumnAutostretch() 520
setColumnWidths() 539
setConnectionState() 578
setCurrentFrame() 619
setCurrentLine() 599
setCurrentSite() 578
setEditNoFramesContent() 625
setFileList() 539
setFloaterVisibility() 647
setFocus() 578
setHideAllFloaters() 634
setHideAllVisualAidst() 625

setInterval() 42
setLayerTag() 510
setLayout() 579
setLinkHref() 474
setLinkTarget() 475
setLinkVisibility() 579
setListBoxKind() 475
setListTag() 476
setLiveDataError() 528
setLiveDataMode() 529
setLiveDataParameters () 529
setLoop() 619
setMenuText() 71
SetNote() 298
setPreferenceInt() 486
setPreferenceString() 487
setPreventLayerOverlaps() 625
setPrimaryView() 648
setRdsPassword() 321
setRdsUserName() 322
setRulerOrigin() 512
setRulerUnits() 513
setSelectedBehavior() 387
setSelectedNode() 548
setSelection() 549, 579
  dreamweaver.setSelection() 659
setShowDependents() 635
setShowFrameBorders() 626
setShowGrid() 626
setShowHeadView() 626
setShowHiddenFiles() 636
setShowImageMaps() 627
setShowInvalidHTML() 627
setShowInvisibleElements() 627
setShowLayerBorders() 627
setShowLayoutTableTabs() 520
setShowLayoutView() 520
setShowLineNumbers() 628
setShowNoscript 590
setShowPageTitles() 636
setShowRulers() 628
setShowStatusBar() 634
setShowSyntaxColoring() 628
setShowTableBorders() 629
setShowToolbar() 629
setShowToolbarIconLabels() 639
setShowToolTips() 636
setShowTracingImage() 629
setShowWordWrap() 629
setSnapDistance() 649
setSnapToGrid() 630
setTableCellTag() 606
setTableColumns() 607
setTableRows() 607
setTextAlignment() 476
setTextFieldKind() 476
setTextFormat() 477
setTimeout() 42
  in floating panels 131
  using with FWLaunch) 304
setTitle() 540
setToolbarPosition() 638
setToolbarVisibility() 637
setTracingImageOpacity() 513
setTracingImagePosition() 513
setUndoState() 494
setUpComplexFind() 461
setUpComplexFindReplace() 461
setUpFind() 462
setUpFindReplace() 463
setupStepsCompleted() 213
setView() 643
share-in-memory 190
showAboutBox() 481
showColdFusionAdmin() 322
showConnectionMgrDialog() 322
showDynamicData() 481
showFindDialog() 463
showFindReplaceDialog() 464
showFontColorDialog() 477
showGridSettingsDialog() 515
showif attribute 88
showIf() 99
showInsertTableRowsOrColumnsDialog() 607
showListPropertiesDialog() 475
showLiveDataDialog() 530
showOdbcDialog() 323
showPagePropertiesDialog() 484
showPreferencesDialog() 482
showProperties() 649
showQuickTagEditor() 536
showRdsUserDialog() 323
showReportsDialog() 536
showRestrictDialog() 323
showResultset() 333
showSPResultset() 334
showSPResultsetNamedParams() 334
showTagChooser() 482, 609
showTagLibraryEditor() 609
shutdown commands 22

Shutdown folder 22
site object 48
  methods of 371
  properties of 47
site reports 103, 541
site.addLinkToExistingFile() 558
site.addLinkToNewFile() 559
site.browseDocument() 432
site.canAddLinkToFile() 433
site.canChangeLink() 433
site.canCheckIn() 433
site.canCheckOut() 434
site.canCloak() 434
site.canConnect() 435
site.canEditColumns() 559
site.canFindLinkSource() 435
site.canGet() 435
site.canLocateInSite() 436
site.canMakeEditable() 436
site.canMakeNewFileOrFolder() 436
site.canOpen() 437
site.canPut() 437
site.canRecreateCache() 437
site.canRefresh() 438
site.canRemoveLink() 438
site.canSelectAllCheckedOutFiles() 438
site.canSelectNewer() 439
site.canSetLayout() 438
site.canSynchronize() 439
site.canUncloak() 440
site.canUndoCheckOut() 440
site.canViewAsRoot() 440
site.changeLink() 560
site.changeLinkSitewide() 559
site.checkIn() 560
site.checkLinks() 560
site.checkOut() 561
site.checkTargetBrowsers() 561
site.cloak() 562
site.defineSites() 562
site.deleteSelection() 562
site.editColumns() 563
site.exportSite() 563
site.findLinkSource() 565
site.get() 565
site.getAppServerAccessType() 565
site.getAppServerPathToFiles() 566
site.getCheckOutUser() 566
site.getCheckOutUserForFile() 567
site.getCloakingEnabled() 567
site.getConnectionState() 567
site.getCurrentSite() 568
site.getFocus() 568
site.getLinkVisibility() 568
site.getLocalPathToFiles() 569
site.getSelection() 569
site.getShowDependents() 634
site.getShowHiddenFiles() 634
site.getShowPageTitles() 635
site.getShowToolTips() 635
site.getSiteForURL() 569
site.getSites() 570
site.importSite() 570
site.invertSelection() 570
site.isCloaked() 571
site.locateInSite() 571
site.makeEditable() 571
site.makeNewDreamweaverFile() 572
site.makeNewFolder() 572
site.newHomePage() 572
site.newSite() 573
site.open() 573
site.put() 573
site.recreateCache() 574
site.refresh() 574
site.remoteIsValid() 574
site.removeLink() 575
site.renameSelection() 575
site.runValidation() 575
site.saveAsImage() 576
site.selectAll() 576
site.selectHomePage() 576
site.selectNewer() 577
site.setAsHomePage() 577
site.setCloakingEnabled() 577
site.setConnectionState() 578
site.setCurrentSite() 578
site.setFocus() 578
site.setLayout() 579
site.setLinkVisibility() 579
site.setSelection() 579
site.setShowDependents() 635
site.setShowHiddenFiles() 636
site.setShowPageTitles() 636
site.setShowToolTips() 636
site.synchronize() 580
site.uncloak() 580
site.uncloakAll() 580
site.undoCheckOut() 581
site.viewAsRoot() 581

snapTracingImageToSelection() 514
snippet tag, attributes 582
snippetPalette.editSnippet() 583
snippetPalette.insert() 583
snippetPalette.insertSnippet() 584
snippetPalette.newFolder() 583
snippetPalette.remove() 584
snippetPalette.rename() 584
snippets
  description attribute 582
  name attribute 582
  preview attribute 582
  type attribute 582
Source Control Integration API
  SCS_AfterGet() 367
  SCS_AfterPut() 368
  SCS_BeforeGet() 366
  SCS_BeforePut() 367
  SCS_canCheckin() 364
  SCS_canCheckout() 364
  SCS_canConnect() 363
  SCS_canDelete() 366
  SCS_canGet() 363
  SCS_canNewFolder() 365
  SCS_canPut() 364
  SCS_canRename() 366
  SCS_CanUndoCheckout() 365
  SCS_Checkin() 357
  SCS_Checkout() 358
  SCS_Connect() 350
  SCS_Delete() 354
  SCS_Disconnect() 351
  SCS_Get() 353
  SCS_GetAgentInfo() 350
  SCS_GetCheckoutName() 357
  SCS_GetConnectionInfo() 355
  SCS_GetDesignNotes() 361
  SCS_GetErrorMessage() 360
  SCS_GetErrorMessageLength() 360
  SCS_GetFileCheckoutList() 359
  SCS_GetFolderList() 352
  SCS_GetFolderListLength() 352
  SCS_GetMaxNoteLength() 361
  SCS_GetNewFeatures() 356
  SCS_GetNoteCount() 360
  SCS_GetNumCheckedOut() 359
  SCS_GetNumNewFeatures() 356
  SCS_GetRootFolder() 352
  SCS_GetRootFolderLength() 351
  SCS_IsConnected() 351
  SCS_IsRemoteNewer() 362
  SCS_ItemExists() 355
  SCS_NewFolder() 354
  SCS_Put() 353
  SCS_Rename() 354
  SCS_SetDesignNotes() 362
  SCS_SiteDeleted() 356
  SCS_SiteRenamed() 356
  SCS_UndoCheckout() 358
source validation 541
source view functions 588
splitFrame() 465
splitTableCell() 608
SQL statements
  getting columns from 325
  showing results of 333
stand-alone reports 104
startDebugger() 501
startOfDocument() 507, 600
startOfLine() 507, 600
startProcessing() 540
startRecording() 489
startup commands 22
Startup folder 22
status codes 281
statusCode property 281
stopAllPlugins() 514
stopPlugin() 514
stopProcessing() 540
stopRecording() 490
stored procedures 324
  getting columns from 329
  getting parameters for 331
  showing results of 334
string object 42
stripTag() 535
submit object 42
subType attribute 161
SWFFile.createFile() 307
SWFFile.getNaturalSize() 309
SWFFile.getObjectType() 309
SWFFile.readFile() 309
synchronize() 580
synchronizeDocument() 601
systemScript property 47

T

tables 332
  getting columns of 327
tables to layers 402
tag attributes 54

Tag Chooser 111
Tag Dialog extensions
definition 20
Tag editor API
applyTag() 118
inspectTag() 117
validateTag() 117
Tag editor functions 608
Tag editor, creating 117
tag inspector functions 612
tag libraries 107
tag library functions 608
tag object 45
tagInspector.deleteTags() 613
tagInspector.editTagName() 613
tagInspector.tagAfter() 612
tagInspector.tagBefore() 612
tagInspector.tagInside() 612
tagName property 45
testConnection() 324
text (field) object 42
text node 46
text objects 46
textarea object 42
title tag 162
toggle functions 620
toggleFloater() 650
toolbar command API
canAcceptCommand() 93
getCurrentValue() 94
getDynamicContent() 94
getMenuID() 95
getUpdateFrequency() 96
isCommandChecked() 97
isDOMRequired() 98
receiveArguments() 99
showIf() 99
toolbar extensions
definition 20
toolbar tag 80
toolbarControls() 214
toolbars
button tag 83
checkbutton tag 84
colorpicker tag 87
combobox tag 86
command API 93
controls 77
creating 77
docking 78
dropdown tag 86
editcontrol tag 87
file definition 79
how commands work 79
how work 77
include/ tag 82
item tags 83
itemref/ tag 82
itemtype/ tag 82
menubutton tag 85
radiobutton tag 85
separator tag 83
simple command file 100
tag attributes 88
toolbar tag 80
toolbars.xml file 77
toolbars.xml file 77
definition 79
tooltip attribute 89
topPage() 600
translated attributes
finding in tags 45
individual 229
inspecting 233
multiple 230
translated tags, inspecting 239
translateMarkup() 228
translation functions 640
translation tag 177
translations tag 177
translationType attribute 178
translator tag 176
translators
attribute 229
block/tag 234
debugging 241
tree control content, manipulating 39
tree controls 36, 38
Tree View
XML 42
TREECOLUMN 38
TREENODE 38
type attribute 582

U

uncloak() 580
uncloakAll() 580
undo() 488, 490
undoCheckOut() 581
unescape() 42
update attribute 91

updateCurrentPage() 525
updatePages() 526
updatePattern tag 174
updatePatterns tag 173
updateReference() 651
URL property 44
user configurations 14
user interface, enhancements 13
user names 319
useTranslatedSource() 641

v

validateFireworks() 302
validateFlash() 447
validateTag() 117
validator 541
value attribute 91
variable grid controls 37
VBScript 135
version attribute 160, 165
versioning 47
view tables 332
viewAsRoot() 581
vline 119

w

W3C 42
whereToSearch attribute 169, 178
width attribute 89
window object 42
window.close() 42
windowDimensions() 51, 60, 106
  in behavior actions 142
  in menu commands 72
workspace
  Dreamweaver 4 13, 78
  Dreamweaver MX 13, 78
wrapSelection() 601
wrapTag() 535
write() 278

x

XHTML
  cleaning up 447
  converting to 448
  creating 451
  testing document 449
XML
  files 42
  Tree View 42
XML files
  CodeHints.xml 392
  insertbar.xml file 51
  snippets 582
  toolbars.xml 77
XML structure 159
XML Tag
  button 53
  category 53
  checkbutton 53
  codehints 393
  insertbar 52
  separator 54
  toolbar 80
XML Tree View 42