Google Web Toolkit

Taking the pain out of Ajax

Ed Burnette

The Pragmatic Bookshelf

Raleigh, North Carolina   Dallas, Texas

Useful Friday Links

- Source code from this book and other resources.
- Free updates to this PDF
- Errata and suggestions. To report an erratum on a page, click the link in the footer.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

To see what we're up to, please visit us at

http://www.pragmaticprogrammer.com

Copyright © 2006 The Pragmatic Programmers LLC.

All rights reserved.

This PDF publication is intended for the personal use of the individual whose name appears at the bottom of each page. This publication may not be disseminated to others by any means without the prior consent of the publisher. In particular, the publication must not be made available on the Internet (via a web server, file sharing network, or any other means).

Produced in the United States of America.

Lovingly created by gerbil #17 on 2006-11-28

Contents

1 Introduction **1**
- 1.1 Life before GWT . . . 1
- 1.2 What GWT does for you . . . 3
- 1.3 About this book . . . 3

2 Getting Started **5**
- 2.1 Supported platforms . . . 5
- 2.2 Installing . . . 5
- 2.3 Create scaffolding . . . 6
- 2.4 Running and debugging . . . 7

3 Hosted vs. Web Mode **11**
- 3.1 Hosted mode . . . 11
- 3.2 Web mode . . . 13
- 3.3 Obfuscation . . . 15
- 3.4 Deployment . . . 16

4 User Interface **17**
- 4.1 Tying into HTML . . . 17
- 4.2 Entry point . . . 19
- 4.3 Events . . . 20
- 4.4 Widgets . . . 21
- 4.5 Panels . . . 28

5 Remote Procedure Calls **32**
- 5.1 Where does your code live? . . . 32
- 5.2 Calling remote code . . . 33
- 5.3 Why a new protocol? . . . 33
- 5.4 GWT RPC basics . . . 34

5.5 Serialization 39

6 History and Bookmarks 43

6.1 The History Token 43
6.2 History Listener 44
6.3 How it Works 44
6.4 Example 45

7 JavaScript Native Interface 48

7.1 Declaring a Native Method 48
7.2 How it Works 49
7.3 Calling JSNI from Java 49
7.4 Calling Java from JSNI 50
7.5 Example 52

8 Internationalization (I18N) 55

8.1 Constants, Messages, and Dictionary 55
8.2 Creating the properties file 56
8.3 Creating the accessor class 58
8.4 Referring to messages 59
8.5 Making module changes 59
8.6 Running the example 60

9 Java Emulation 62

9.1 Language subset 62
9.2 Library subset 65
9.3 Supported packages 66
9.4 Regular Expressions 68

Chapter 1

Introduction

The Google Web Toolkit (GWT) was unveiled to an unsuspecting public on May 18th, 2006 at the annual JavaOne conference in San Francisco. The premise behind GWT is simple: make Ajax[1] development easier by hiding browser incompatibilities from the programmer and allowing the developer to work in a familiar Java development environment.

The announcement was one of the highlights of the conference and interest continues to grow. Developers have used GWT technology in everything from games to mortgage calculators. The gwtPowered community site lists over 130 examples, articles, widgets, and other resources. Why has the Google Web Toolkit become such a hot topic?

If you've ever written a non-trivial Ajax application before, then I'm sure you can sympathize with the need to make the process easier. If not, then a little background is in order.

1.1 Life before GWT

Dynamic web applications are typically written in several different languages across two or more tiers. On the client side (the part running in the browser), you have HTML markup of course,

[1] The term *Ajax* was famously coined in February 2005 by Jesse James Garrett. Originally it was an acronym for Asynchronous Javascript And Xml. The technology has actually been around for a few years—for example it was used in Outlook Web Access in 2000—but didn't get much attention until Google popularized it with applications such as GMail and Google Maps.

plus you have some logic written in JavaScript to perform tasks like client-side validation and manipulation of the HTML document object model (DOM).

> JavaScript is a red-headed step-child of a language that first appeared in the Netscape browser in 1995 as a way to script Java applets. It was adopted by Microsoft in the following year, becoming the de-facto standard for scripting inside the browser. Despite having Java in its name, it bears little resemblance to that language. The closest thing to JavaScript would be... well actually there's nothing quite like JavaScript. Some would count that as a good thing.

Unfortunately, slight differences in the JavaScript language between browsers, along with major differences in the DOM, make coding these clients a bit like walking through a mine field. Various libraries such as Dojo and Prototype were created to smooth out the rough edges but JavaScript/browser programming is still something of a black art. Some developers have abandoned HTML and JavaScript altogether in favor of Flash or other alternatives.

On the server side you have a web server tier and optionally a data tier. Commodity web servers such as Apache, Tomcat, Lighttpd, and IIS host your application logic, which is written in Java, PHP, Ruby, C#, Klingon (ok, maybe not Klingon), or other languages. JavaScript is not used on the server except by a few masochists. Data services are provided by databases such as MySQL, Oracle, Sql Server, and so forth. Often the actual database is hidden behind an Object/Relational (O/R) layer such as Hibernate.

Although this architecture is very flexible, its complexity makes it hard to manage. Frameworks such as Ruby on Rails grew up to reduce the complexity on the server side. Other frameworks like Java Server Faces (JSF) and Microsoft Atlas try to standardize and provide built-in implementations of client-side operations such as validation. However, substantial dynamic web applications are still much harder to write than the traditional desktop applications they're supposed to replace.

1.2 What GWT does for you

Google Web Toolkit unifies client and server code into a single application written in one language: Java. This has many advantages. For one thing, far more developers know Java than JavaScript or Flash. Another reason is that Java is blessed with an abundance of developer tools such as Eclipse, NetBeans, and IDEA. GWT lets you create a web application in much the same way as you would create a Swing application—creating visual components, setting up event handlers, debugging, and so forth—all within a familiar IDE.

By standardizing on one language you can share code on the client and server. For example you can run the same validation code—once on the client for immediate feedback, and once on the server for maximum security. You can even move code between tiers as you refactor your application to adapt to changing requirements.

GWT also abstracts the browser's DOM, hiding differences between browsers behind easy to extend object-oriented UI patterns. This helps make your code portable over all supported browsers.

If this sounds too good to be true, well, it is a little bit. You still have to be careful not to introduce browser-specific dependencies. As tech guru Joel Spolsky likes to say, all abstractions are leaky. Occasionally you may have to delve into CSS/DOM/JavaScript to address browser quirks in non-trivial programs. But with GWT this is the exception rather than the rule.

1.3 About this book

This book provides you with a thorough introduction to the Google Web Toolkit. From installation, through your first application, to UI

components and Remote Procedure calls, you'll learn the ins and outs of the framework. Some knowledge of Java programming and HTML is assumed, but you don't have to be an expert in web programming.

History

This section lists all the updates made to the first edition of this book.

- P1.1 (27nov2006): Updated for GWT 1.2.22. Added I18N chapter.
- P1 (11sep2006): Updated for GWT 1.1.10.
- P0 (23aug2006): Original for GWT 1.1.0.

Ok, enough talk—let's get started with your first GWT application!

Chapter 2

Getting Started

Getting started developing with Google Web Toolkit is easy. In this chapter I'll show you how to set up a few things, and then you can jump right in and create a working application using the scaffolding GWT provides.

2.1 Supported platforms

Development of GWT applications is supported on Windows, Linux, and MacOSX (as of GWT 1.2). All the examples in this book were done on Windows.

GWT applications may be deployed in web servers running on any operating system, and viewed on any modern desktop browser (IE6, IE7, Firefox, Opera, and so on).

2.2 Installing

Before you start coding you need to install Java, an IDE, and GWT itself.

Java 1.4.2+

First you need a copy of Java. Although GWT works with Java 1.4.2 and newer, you might as well get the latest Sun JDK 5.0 or 6.0 update from the Sun download site To verify you have the right version, run this command from your shell window:

```
C:\> java -version
java version "1.5.0_07"
```

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Java HotSpot(TM) Client VM (build 1.5.0_07-b03, mixed mode, sharing)
```

Sun has a taken a page from Microsoft's playbook and bundled their NetBeans IDE in the 5.0 JDK. However this is sometimes an older version of the JDK, and this kind of bundling should be discouraged anyway. Fortunately, you can still get just the plain JDK without NetBeans and save yourself 70MB of extra downloading at the same time. Unless you really want NetBeans of course.

Eclipse

Second, you need a copy of the Eclipse IDE. While you can use other Java IDEs such as NetBeans or IDEA, the Google developers use Eclipse and so do I, so that's what I'll be using for the remainder of this book. Go to the Eclipse downloads page, pick 3.2 (or later), and then get the Eclipse package for your platform (Windows, Linux, Mac, etc.). You can either get the full SDK (the one that has all the sources and programmer documentation), or for a smaller download you can just get the Platform Runtime Binary plus the JDT Runtime Binary.

For an easier Eclipse download experience, you could try the Eclipse on demand site, sponsored by Yoxos, or Easy Eclipse, sponsored by nexB.

GWT

Next, download the Google Web Toolkit SDK (1.2.22 or later). Unzip the Google Web Toolkit onto your machine. No special install is needed. Now you're ready to create your first project.

2.3 Create scaffolding

At a command prompt, run these commands (substituting the appropriate paths for your system):

```
C:\> mkdir c:\gwt-projects\MyProject
```

```
C:\> cd c:\gwt-projects\MyProject

C:\gwt-projects\MyProject> projectCreator -eclipse MyProject
Created directory C:\gwt-projects\MyProject\src
Created file C:\gwt-projects\MyProject\.project
Created file C:\gwt-projects\MyProject\.classpath

C:\gwt-projects\MyProject> applicationCreator -eclipse MyProject\
   com.xyz.client.MyApp
Created directory C:\gwt-projects\MyProject\src\com\xyz
Created directory C:\gwt-projects\MyProject\src\com\xyz\client
Created directory C:\gwt-projects\MyProject\src\com\xyz\public
Created file C:\gwt-projects\MyProject\src\com\xyz\MyApp.gwt.xml
Created file C:\gwt-projects\MyProject\src\com\xyz\public\MyApp.html
Created file C:\gwt-projects\MyProject\src\com\xyz\client\MyApp.java
Created file C:\gwt-projects\MyProject\MyApp.launch
Created file C:\gwt-projects\MyProject\MyApp-shell.cmd
Created file C:\gwt-projects\MyProject\MyApp-compile.cmd
```

The projectCreator and applicationCreator commands are two shell scripts that are supplied as part of GWT, so you'll need to specify the path to them or add the GWT directory to your system PATH variable. projectCreator builds the scaffolding for a generic GWT project, and applicationCreator adds a simple GWT application that you can build upon. MyProject, MyApp, and com.xyz are just example names; you can use anything you want. However the .client part of the package name is important; we'll come back to that later.

2.4 Running and debugging

At this point you're ready to try out the application.

Figure 2.1: Hello world GWT application

Running outside Eclipse

First, let's run the app outside of the IDE by using one of the handy shell scripts that the scaffolding provided:

```
C:\gwt-projects\MyProject> MyApp-shell
```

If everything is working correctly two windows will appear: The GWT development shell (this is kind of like a console window) and a web browser window. See Figure 2.1

Verify the application works by clicking the `Click me` button—the text *Hello World!* will appear. Congratulations, you've just created and run your first GWT application.

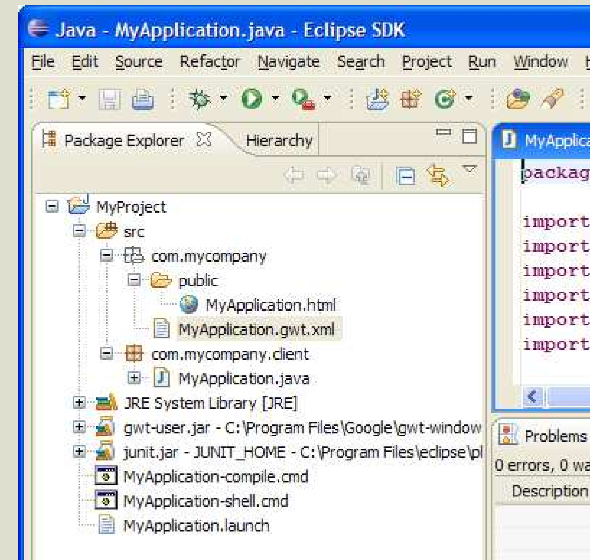


Figure 2.2: GWT project in Eclipse

Running inside Eclipse

Now close the two GWT windows, start up Eclipse, and import this project into your workspace (File → Import → Existing Projects Into Workspace). The project will build, and if all is successful you will end up with something like Figure 2.2 .

Now select Run → Debug..., and click on the launch configuration titled MyApp (under Java Application). Then click on Debug. The two GWT windows should appear again, just like in Figure 2.1, on the preceding page

Debugging

Ok, now for the neat part. Leave the application running and switch back to the Eclipse window. Set a breakpoint in the onClick() method

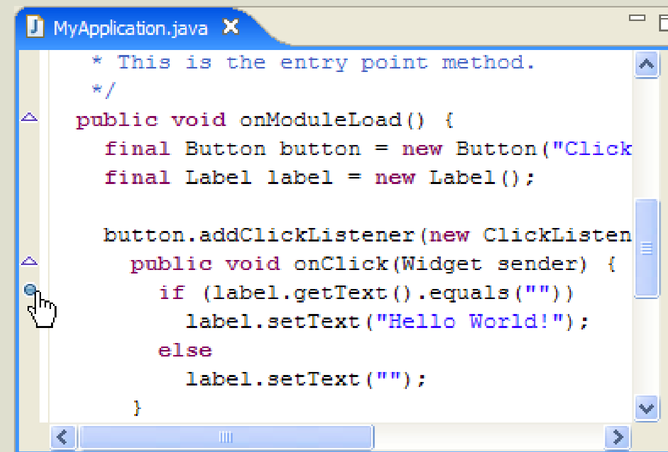


Figure 2.3: Setting a breakpoint

in MyApp.java by double-clicking the gutter area next to the line (see Figure 2.3).

Then switch to the application window and click the `Click me` button again. Eclipse will stop at the line in your Java code where you put the breakpoint. You can single step, examine variables, and so forth.

Hmm, that's Java code, yet you're writing an Ajax application that will eventually be deployed in pure JavaScript. All the power of your Java development environment—Eclipse, the debugger, refactoring, source management, and so on—is suddenly available in the Ajax world. Can you begin to see the potential of this technology? In the next chapter we'll take a look behind the curtain to reveal how the magic is done.

Chapter 3

Hosted vs. Web Mode

In the previous chapter, when you invoked a GWT application you were using what Google calls *hosted mode*. Hosted mode is only used during development. When in production, your application will be running in *web mode*. Before going any further in using GWT you need to understand the difference between the two. Note that as of this writing, hosted mode is only available on Windows and Linux.

3.1 Hosted mode

Think of hosted mode as training wheels for your GWT application. It's a hybrid development environment unique to GWT that lets your code run as real Java code, but still inside a browser. Execution in hosted mode is controlled by the Google Web Toolkit *development shell* (the background window in Figure 2.1, on page 8).

The development shell is actually an Eclipse Rich Client application, consisting of the shell console, a tomcat server, and one or more hosted browsers.

The *hosted browser* (the front window in Figure 2.1) has two connections back to the development shell. One is just a regular http connection to get the web pages, .css files, images, and other resources. All these are handled by the embedded Tomcat server using a servlet called com.google.gwt.dev.shell.GWTShellServlet.

The second connection is a back-door that intercepts all interactions inside the hosted browser and routes them not to JavaScript but to Java code in the shell. That Java code in turn calls your real client Java code, which was compiled to bytecode by your IDE. The exact

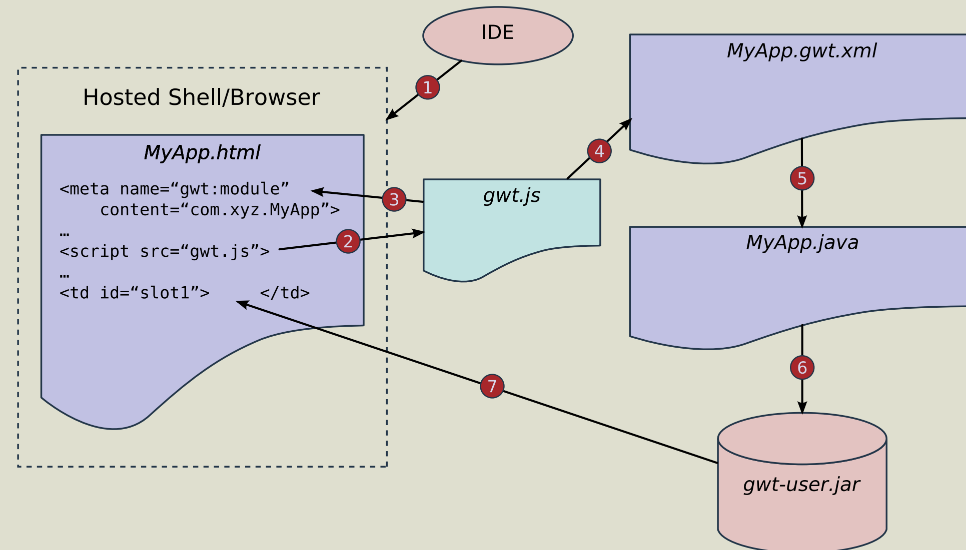


Figure 3.1: How a GWT page is loaded in hosted mode

details of how this is done are hidden in the shell code, which is not open source.

Figure 3.1 shows how a page is loaded in hosted mode:

1. The Shell program opens a hosted browser window, which loads MyApp.html.
2. MyApp.html loads gwt.js with a <*script*> tag.
3. gwt.js scans MyApp.html and parses out the <*meta name='gwt-module'*> to get the module name.
4. GWT reads the module file (MyApp.gwt.xml) to find the name of the EntryPoint class (MyApp).
5. The MyApp class is instantiated and its onModuleLoad() method

Joe Asks...

Is GWT open source?

The short answer is yes. All the libraries and JavaScript you need to deploy your code are covered under the Apache license. Two development time pieces – the development shell and the Java to JavaScript compiler – are not open source at this time. But according to GWT tech leader Bruce Johnson, even these parts may be opened in the future.

is called. Your application begins.

6. Your application code makes calls into the GWT user library (gwt-user.jar), which is also Java code.

7. Code in gwt-user.jar manipulates the hosted browser's DOM to add UI components to the web page, and redirects all browser events back to the Java application code using special hooks in the browser.

Because real Java code is running, you can use Java tools like the Eclipse debugger, findbugs, pmd, JUnit, and so forth. It's almost as if you were developing a rich client program with Swing or SWT because it's Java end-to-end.

Once you've debugged and unit tested your code the next step is to compile it into a form that can be run inside a regular browser (not one that has been hijacked by the development shell). That's where web mode comes in.

3.2 Web mode

When you click the Compile/Browse button in the hosted browser, the GWT compiler translates your .client package into JavaScript and opens a normal web browser to view the application. At this point pages are still served by the shell's Tomcat instance, but they could just as easily come from the file system or a normal web server.

Another way to invoke the GWT compiler is with the shell script provided by the scaffolding (MyApp-compile). You could also write an Ant script to do it if you prefer. For example to maintain the gwtpowered.org site I have an ant script that does the compile and then

copies everything to my hosting provider. You can find the source at http://code.google.com/p/gwtpowered.

However you invoke it, the GWT compiler combines your code with a JavaScript version of the GWT API (the equivalent of gwt-user.jar) in one JavaScript file. This code and several supporting files are placed in the www directory inside your project. Everything from your public directory is copied there as well. The table below explains what all the files do:

> The GWT compiler actually creates several different browser-specific versions of compiled JavaScript code, for several different classes of browsers (IE, Firefox, etc.). Google calls these cache files. The filenames use long unguessable hex codes. Only one of these is loaded, depending on your browser.
>
> The .cache file is cached by the client to improve load time for future visits. When the app is modified and recompiled the .cache file name will be different so the browser will download it again. Any old .cache files will be ignored.
>
> Future versions of GWT might create more or fewer of these cache files. As support for new browsers is added to GWT, all you have to do is recompile your application with the newer compiler to support them

Filename	**Description**
long-hex-name.cache.html	Compiled JavaScript
long-hex-name.cache.xml	Implementation defined
module-name.nocache.html	Cache file selection
gwt.js	Common GWT bootstrap code
history.html	Contents of history IFrame
MyApp.html	Main page, copied from public
tree*.gif	+/- images used by the Tree widget

The flow of execution during a page load in web mode (see Figure 3.2, on the following page) is a bit different than in hosted mode. Here's a breakdown of what happens:

1. The web browser loads MyApp.html.
2. MyApp.html loads gwt.js with a <*script*> tag.
3. gwt.js scans MyApp.html and parses out the <*meta name='gwt-module'*> to get the module name.
4. gwt.js modifies the page to include an <*iframe*> that causes the source file *module-name*.nocache.html to be loaded.

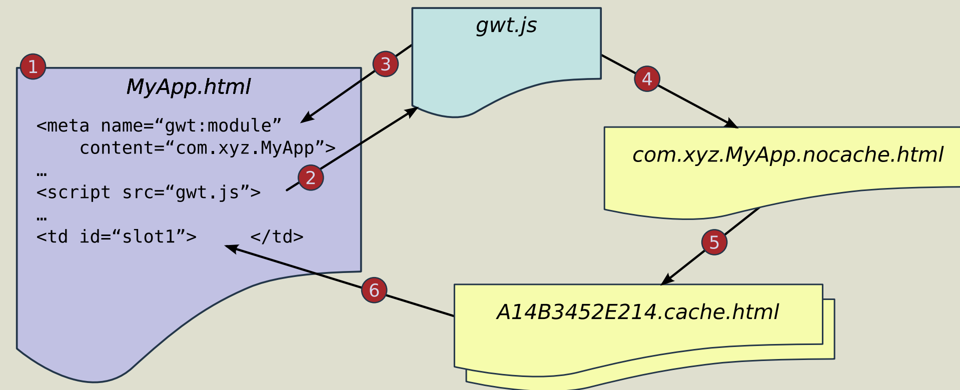


Figure 3.2: How a GWT page is loaded in web mode

5. JavaScript inside the file *module-name*.nocache.html looks at the browser's userAgent field to determine what kind of browser the user is running (IE6, Mozilla, Opera, etc.). Then it selects the correct code (cache file) for that browser type and redirects the <*iframe*> there.

6. The JavaScript equivalent of your onModuleLoad() method is executed, and the rest of your application goes from there. Manipulations to the browser DOM are performed with ordinary dynamic HTML calls in the compiled JavaScript.

3.3 Obfuscation

By default, the GWT Java to JavaScript compiler will produce obfuscated output. Code that has been *obfuscated* is smaller than human-readable code, and is harder to reverse-engineer. It's very difficult to debug, though. Should you ever need to debug the JavaScript that GWT produces, you can turn off obfuscation with command line

parameters on the GWT compiler (for example as arguments to the MyApp-compile.cmd script). Use the -style pretty option to produce good looking output with readable names and indentation. To see full Java types as part of the names, use the -style detailed option instead.

3.4 Deployment

All the examples up to now have been dependent on the hosted shell's Tomcat server to serve up all the application's files. However in web mode they can be delivered by any web server or even (for testing) the local file system. To try this out, copy the entire www directory to another location in your file system and bring up a regular browser on your starting HTML page. The application should work exactly the same as before.[1]

For simple programs like this there is no interaction with the server because none of your code is running there. We'll see some more complicated programs in Chapter 5, *Remote Procedure Calls*, on page 32 that do require more than copying a directory, but for now let's see what fun we can have with GWT's user interface components.

[1] History doesn't work on the local file system in IE6. But why are you using IE anyway?

Chapter 4

User Interface

One thing you'll notice when developing a GWT application is that it's much like developing a desktop application with Swing, SWT, or even Visual Basic. You create controls such as buttons, lists, and tables, you add them to parents, and you interact with them via listeners. You lay them out in a certain arrangement and try to make it look nice at any font size and screen resolution. The main difference is that your GWT app will appear in a web browser, so there has to be an HTML page involved somewhere.

Traditional web applications are structured as a series of HTML pages with some kind of navigation between them. For example you might have an inventory page, an ordering page, and a confirmation page. In a GWT application, however, you stay on one page the whole time. Instead of changing web pages, you change the contents of the one page to reflect the current state. For example you might have three different panels for inventory, ordering, and confirmation within the page, and show only one at any given time. This gives the user a smoother, more responsive experience compared to the old way.

4.1 Tying into HTML

If you look in your project under src/com/xyz/public you'll find a file called MyApp.html. This is the canvas in which the GWT user interface will be hosted.

Every GWT application lives inside a single HTML page. It could be a static page like this one, or a page generated with a server-

side framework like JSP, Struts, Ruby on Rails, etc.. To keep things simple we'll just look at static pages for the rest of this book.

The fact that MyApp.html is in the public directory means it will be copied verbatim into the final deployment area on the server (see Section 3.2, *Web mode*, on page 13). If you have any images, style sheets, etc., then they need to go somewhere in this same directory.

Near the top of the HTML page is a required meta tag that associates this page with a GWT *module*.

Download MyProject/src/com/xyz/public/MyApp.html

```
<meta name='gwt:module' content='com.xyz.MyApp'>
```

A GWT module is a collection of client-side application code and resources you supply. The module named `com.mycompany.MyApp` is defined in the module file src/com/mycompany/MyApp.gwt.xml.

Download MyProject/src/com/xyz/MyApp.gwt.xml

```
<module>

  <!-- Inherit the core Web Toolkit stuff.                    -->
  <inherits name='com.google.gwt.user.User' />

  <!-- Specify the app entry point class.                     -->
  <entry-point class='com.xyz.client.MyApp' />

</module>
```

Here you can see the name of your Java class. Logically, when the HTML page is loaded, GWT looks up the meta tag, reads the xml file to get the class name, and starts calling code in the EntryPoint class. As of GWT 1.1 you can also have GWT inject .css files and other resources with module directives.

4.2 Entry point

Your entry point class (MyApp) extends the EntryPoint interface and provides one method: onModuleLoad(). This method is responsible for constructing your GWT app's user interface.

Download MyProject/src/com/xyz/client/MyApp.java

```
/**
 * This is the entry point method.
 */
public void onModuleLoad() {
   final Button button = new Button("Click me");
   final Label label = new Label();
   //...
```

The scaffolding code creates two GWT user interface elements, a Button and a Label. These are examples of GWT *widgets*, which are similar to widgets in other Java GUI libraries like Swing and SWT. See Section 4.4, *Widgets*, on page 21 for a list of widgets supplied by GWT.

If you go back and look at the HTML file, near the bottom it references two placeholders for dynamic content:

Download MyProject/src/com/xyz/public/MyApp.html

```
<table align=center>
   <tr>
      <td id="slot1"></td><td id="slot2"></td>
   </tr>
</table>
```

Note the id='s given to these cells, *slot1* and *slot2*. In the Java code these two slots are referenced by their id=s and filled in with the Button and the Label you just created:

Download MyProject/src/com/xyz/client/MyApp.java

```
// Assume that the host HTML has elements defined whose
// IDs are "slot1", "slot2". In a real app, you probably would not want
// to hard-code IDs. Instead, you could, for example, search for all
// elements with a particular CSS class and replace them with widgets.
//
RootPanel.get("slot1").add(button);
RootPanel.get("slot2").add(label);
```

A *RootPanel* just wraps an HTML element on the page. They are created on demand. This code gets a RootPanel for each of the two <*td*> elements referenced by id=, and then adds the GWT widgets inside them.

This table defines how the widgets are laid out on the screen. A better way to define widget layout is to use a Panel. GWT panels are just widgets that can contain one or more other widgets, and arrange them in some predefined way. For example, you could have created a HorizontalPanel, added the button and label to that, and then added the panel to the RootPanel of the page (e.g., `RootPanel.get().add(hPanel)`). See Section 4.5, *Panels*, on page 28 for a list of panels predefined in the Google Web Toolkit user library.

Joe Asks...

What about memory leaks?

If you've done any Ajax programming before, you might be wondering about how the listeners get cleaned up, how leaks are prevented, and so on. You're only wondering that because you've learned this is a real hassle in JavaScript programming, especially in certain browsers. Well, stop biting your nails. As long as you follow its rules, GWT will take care of all that. Trust me.

4.3 Events

A web app would be pretty boring if you couldn't interact with it, so now let's look at the final ingredient—making that button do something. If you were programming a Swing app you would add a click listener on your JButton. In GWT, you do basically the same thing.

Download MyProject/src/com/xyz/client/MyApp.java

```
button.addClickListener(new ClickListener() {
   public void onClick(Widget sender) {
```

```
        if (label.getText().equals(""))
          label.setText("Hello World!");
        else
          label.setText("");
      }
    });
```

When the user clicks on the button, the onClick() method is called. The logic here could do anything, but in this case it just toggles the text of the Label widget to either be blank or say "*Hello World!*". See Figure 2.1, on page 8 for how the final application looks.

Joe Asks...

Do I still have to worry about browser differences in CSS?

After several years of abysmal CSS support in browsers we've now worked our way up to "adequate". Although GWT tries to hide differences between browsers, style sheets are still subject to different interpretations by browser writers. So the best advice is to keep it simple, rely on GWT layouts instead of CSS positioning, and test on different browsers.

4.4 Widgets

This section provides a list of most of the widgets built-in to GWT. For each widget, a description of what the widget does is provided, along with common methods you'll need to call. Many widgets include an image showing what the widget might look like. Of course, the actual display will depend on the user's browser and operating system. For information on laying out widgets on the page see Section 4.5, *Panels*, on page 28.

Colors, fonts, and other style information should be kept in standard Cascading Style Sheets (CSS). Often a widget will predefine a CSS class name for you (shown here after the description), but you can also add your own class names to the widget and reference either name in your .css files.

Download MyProject/src/com/xyz/client/MyApp.java

```
// Assume that the host HTML has elements defined whose
// IDs are "slot1", "slot2". In a real app, you probably would not want
// to hard-code IDs. Instead, you could, for example, search for all
// elements with a particular CSS class and replace them with widgets.
//
RootPanel.get("slot1").add(button);
RootPanel.get("slot2").add(label);
```

A *RootPanel* just wraps an HTML element on the page. They are created on demand. This code gets a RootPanel for each of the two <*td*> elements referenced by id=, and then adds the GWT widgets inside them.

This table defines how the widgets are laid out on the screen. A better way to define widget layout is to use a Panel. GWT panels are just widgets that can contain one or more other widgets, and arrange them in some predefined way. For example, you could have created a HorizontalPanel, added the button and label to that, and then added the panel to the RootPanel of the page (e.g., `RootPanel.get().add(hPanel)`). See Section 4.5, *Panels*, on page 28 for a list of panels predefined in the Google Web Toolkit user library.

Joe Asks...

What about memory leaks?

If you've done any Ajax programming before, you might be wondering about how the listeners get cleaned up, how leaks are prevented, and so on. You're only wondering that because you've learned this is a real hassle in JavaScript programming, especially in certain browsers. Well, stop biting your nails. As long as you follow its rules, GWT will take care of all that. Trust me.

4.3 Events

A web app would be pretty boring if you couldn't interact with it, so now let's look at the final ingredient—making that button do something. If you were programming a Swing app you would add a click listener on your JButton. In GWT, you do basically the same thing.

Download MyProject/src/com/xyz/client/MyApp.java

```
button.addClickListener(new ClickListener() {
   public void onClick(Widget sender) {
```

show() method to open the dialog and the hide() method to make it go away.

CSS Style Rules:

- `.gwt-DialogBox { }`
- `.gwt-DialogBox .Caption { }`

FileUpload

A widget that wraps an input file element. Can only be used inside a FormPanel. Set the name of the input element that will be submitted to the server using the setName() method.

FlexTable and Grid

sender	email
markboland05	mark@example.com
Hollie Voss	hollie@example.com
boticario	boticario@example.com
Emerson Milton	emerson@example.com
Healy Colette	healy@example.com
Brigitte Cobb	brigitte@example.com
Elba Lockhart	elba@example.com

A table that can contain text, HTML, or any type of widget. FlexTable creates cells on demand and can be ragged (i.e., each row can contain a different number of cells). Grid always has the same, fixed size.

Use the setText(), setHTML(), or setWidget() method to add cell items, and getCellFormatter() to customize the appearance of the cells. Cells in a FlexTable can span multiple rows or columns.

Frame and NamedFrame

A widget that wraps an HTML <*iframe*> element, which can contain an arbitrary web site. Note that if you are using History (see Chapter 6, *History and Bookmarks*, on page 43), any browser history items generated by the Frame will interleave with your application's history. Use the constructor or the setUrl() method to set the web page address.

CSS Style Rules: `.gwt-Frame {}`

HTML

A widget that can contain arbitrary HTML. If you only need simple text, then use the Label widget instead. Inserting HTML, especially if it contains user-supplied data, can lead to potential security issues with cross-site scripting if you're not careful.

CSS Style Rules: `.gwt-HTML { }`

Image

A widget that displays the image at a given URL. Use the constructor or the setUrl() method to specify the image address. You can also use addLoadListener() to be notified when the image is done loading (or an error occurs).

CSS Style Rules: `.gwt-Image { }`

Hyperlink

Info
Buttons
Menus
Images
Layouts

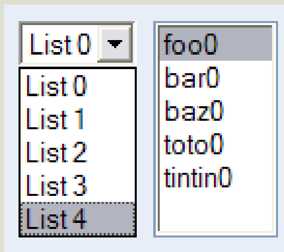
A widget that serves as an internal hyperlink, i.e., a link to another state of the running application. When clicked, it will create a new history frame using History.newItem(), but without reloading the page. To create a hyperlink to another page (for example, a different site), use the HTML widget instead.

CSS Style Rules: `.gwt-Hyperlink { }`

Label

A widget that contains arbitrary text, not interpreted as HTML. It supports word wrapping and arbitrary horizontal alignment. Use the setText() method to change the text.

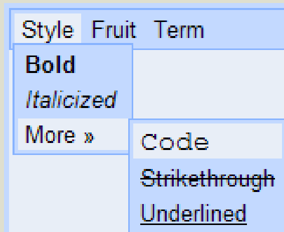
CSS Style Rules: `.gwt-Label { }`



ListBox

A widget that presents a list of choices to the user, either as a list box or as a drop-down list. Add items with the addItem() method, and set the height of the box with setVisibleItemCount(). If you set the height to 1, that turns it into a drop-down list. To tell which item(s) are selected call the getSelectedIndex() or isItemSelected() methods.

CSS Style Rules: `.gwt-ListBox { }`



MenuBar and MenuItem

A standard menu bar widget. A menu bar can contain any number of menu items, each of which can either fire a Command or open a cascaded menu bar. Use the addItem() method to add things to the menu bar.

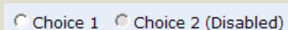
CSS Style Rules:

- `.gwt-MenuBar { the menu bar itself }`
- `.gwt-MenuItem { menu items }`
- `.gwt-MenuItem-selected { selected menu items }`

PasswordTextBox

Identical to a regular text box, except that the input is visually masked by the browser to prevent casual eavesdropping.

CSS Style Rules: `.gwt-PasswordTextBox { }`



RadioButton

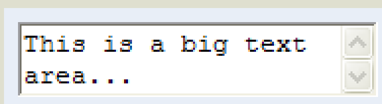
A mutually-exclusive selection radio button widget. Call the addClickListener() method to be notified when the user clicks on it, and call the isChecked() method to see if it's checked or not.

TabBar

A horizontal bar of folder-style tabs, most commonly used as part of a TabPanel. Call the addTab() method to add items to the bar, and addTabListener() to get notifications before (onBeforeTabSelected()) or after (onTabSelected()) a tab is selected.

CSS Style Rules:

- `.gwt-TabBar` {*the tab bar itself*}
- `.gwt-TabBarFirst` {*the left side spacer of the bar*}
- `.gwt-TabBarRest` {*the right side spacer of the bar*}
- `.gwt-TabBarItem` {*tabs*}
- `.gwt-TabBarItem-selected` {*additional style for selected tabs*}



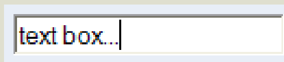
TextArea

A text box that allows multiple lines of text to be entered. Set the size of this area with the setCharacterWidth() and setVisibleLines() methods.

Call getCursorPos(), getSelectionLength(), and getSelectedText() to learn about the currently selected text. You can also move the cursor position or set the selection range programmatically. Call the addKeyboardListener() method to be notified of (and possibly suppress) key presses.

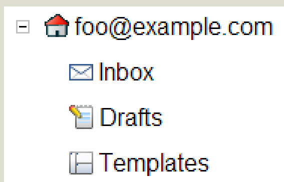
CSS Style Rules: `.gwt-TextArea { }`

TextBox



A text box that allows a single line of text to be entered. Set the size of this area with the setVisibleLength() method. Call getCursorPos(), getSelectionLength(), and getSelectedText() to learn about the currently selected text. You can also move the cursor position or set the selection range programmatically. Call the addKeyboardListener() method to be notified of (and possibly suppress) key presses.

Tree and TreeItem



A standard hierarchical tree widget. The tree contains a hierarchy of TreeItem objects that the user can open, close, and select. Call the addItem() method on Tree to add items to the root of the tree. Items can either be HTML strings or TreeItem objects containing nested branches of the tree. Call the addItem() method on TreeItem to add a nested item.

CSS Style Rules:

- `.gwt-Tree` {*the tree itself*}
- `.gwt-TreeItem` {*a tree item*}
- `.gwt-TreeItem-selected` {*a selected tree item*}

4.5 Panels

A panel is a widget that contains other widgets (including other panels). You use them to layout widgets in grids, decks, rows, or columns. This section describes the panels which are built-in to GWT. See also Section 4.4, *Widgets*, on page 21.

AbsolutePanel

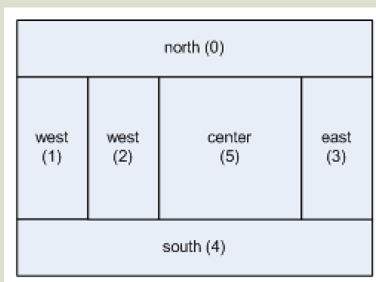
An absolute panel positions all of its children absolutely (in CSS, **position: absolute**), allowing them to overlap. Widgets are positioned in the document coordinate space. Note that this panel will not automatically resize itself to allow enough room for its absolutely-positioned children. It must be explicitly sized in order to make room for them.

DeckPanel

A panel that displays all of its child widgets in a 'deck', where only one can be visible at a time. All children reside simultaneously in the browser's memory, which could have memory implications for large decks.

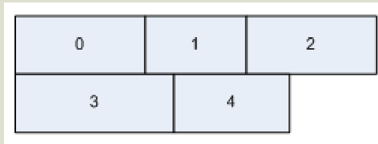
Call the add() method to add a widget to the deck, and showWidget() to make a particular one visible. DeckPanel is used for the body of TabPanel.

DockPanel



A panel that lays its child widgets out "docked" at its outer edges, with the central widget taking up the remaining space. Use the add() method to place widgets at the edges. You can also set the horizontal and vertical alignment of individual cells.

Note that this panel has some order dependent behavior. For example, if WEST is added first it will get the upper left corner, but if NORTH is added first it will get the upper left corner.



FlowPanel

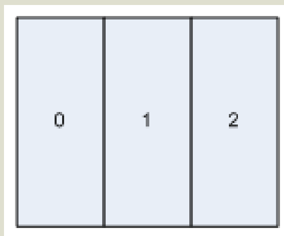
A panel that formats its child widgets using the default HTML layout behavior. Use the add() method to add a new child widget to the panel. Set the CSS rule **display: inline** on child widgets to make them flow horizontally in a FlowPanel.

FocusPanel

A simple panel that makes its contents focusable, and adds the capability to process mouse and keyboard events. Use the constructor or setWidget() method to set the panel's widget.

FormPanel

A panel that wraps its contents in an HTML <*form*> element. A FormPanel may only contain these elements: TextBox, PasswordTextBox, RadioButton, CheckBox, TextArea, ListBox, FileUpload. Use the setName() method on the containing elements to associate them with a form field passed to the server. Use setAction() to set the URL used to submit the form, and the submit() method to actually submit the form. Call addFormHandler() to get notified when the form is about to be submitted (cancelable) or when submission is complete.



HorizontalPanel

A panel that lays all of its widgets out in a single horizontal column. Use the add() method to place widgets in the panel. You can also set the horizontal and vertical alignment of individual cells.

Tip: set the alignment property before adding children so you don't have to set it on each child.

HTMLPanel

A panel that contains HTML with replaceable parts. Use the constructor to set the initial HTML string. Inside that string some elements have the id= attribute, allowing you to use the add() method to attach child widgets to those identified elements. When constructing the HTML string you should use the createUniqueId() method, because no two elements in a document should have the same id.

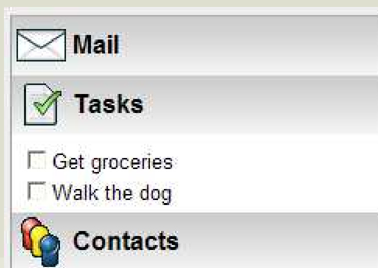
PopupPanel

A popup panel can "pop up" over other widgets. It overlays the browser's client area (and any previously-created popups). To use, create a new class that extends PopupPanel, and call the setWidget() method in the constructor. Call the show() method to display the popup and the hide() method to remove it. If you'd like to preview (and possibly suppress) keyboard events before they are passed to any other widget, override the onKeyDownPreview(), onKeyPressPreview(), or onKeyUpPreview() methods.

ScrollPanel

A simple panel that wraps its contents in a scrollable area. Use the constructor or the setWidget() function to define the widget to wrap.

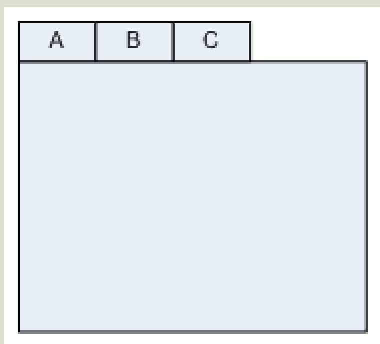
StackPanel



A panel that stacks its children vertically, displaying only one at a time, with a header for each that the user can click to display it. Use the add() method to add a widget and its header to the stack.

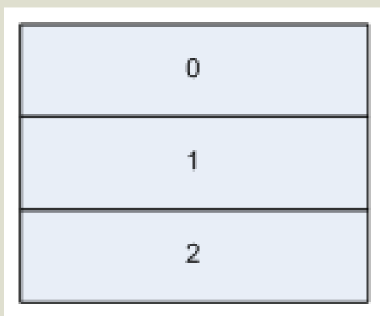
CSS Style Rules:

- `.gwt-StackPanel` {*the panel itself*}
- `.gwt-StackPanelItem` {*unselected items*}
- `.gwt-StackPanelItem-selected` {*selected items*}



TabPanel

A composite of TabBar and DeckPanel that shows a tabbed set of widgets. Only one of the widgets is visible at any given time, depending on which tab is selected. To use, create a new class that extends TabPanel, and call the add() method to add each widget to the panel. Implement onBeforeTabSelected() to preview (and possibly suppress) tab selection events, or the onTabSelected() method to get notified after the fact.



VerticalPanel

A panel that lays all of its widgets out in a single vertical column. Use the add() method to place widgets in the panel. You can also set the horizontal and vertical alignment of individual cells.

Tip: set the alignment property before adding children so you don't have to set it on each child.

This chapter has covered the basics you need to know to create GWT user interfaces. In the next chapter we'll delve into another powerful feature of GWT that lets your client and server side code talk to each other: Remote Procedure Calls.

Chapter 5

Remote Procedure Calls

For years, developers have struggled to find the perfect place for their code to run. Should it run on a server where it can be centrally controlled and secured, or should it run closer to the user on their desktop to take advantage of local processing power and interactivity? GWT lets part of your application run in both places, with seamless communication between them using Remote Procedure Calls (RPC).

5.1 Where does your code live?

Fat client applications, i.e., traditional desktop applications, run completely on the user's desktop. An example of this would be Microsoft Word. The application is installed on the user's machine, and all files produced live there too.

Thin client applications take the opposite approach. They run on some shared server machine, and the desktop is only used for the user interface. Most web applications fall into this category, for example Ebay and Amazon. Nothing is installed on the desktop except for a standard viewer (the web browser).

There are various other technologies that fit somewhere in between (rich, smart, client/server, etc.). Ajax applications in general, and GWT in particular, fall into a class called *Rich Internet Applications* (RIA). Some of the application code runs on the client desktop and some runs on a server machine.

Like thin client apps, RIA's don't require anything to be installed on the desktop other than a standard web browser. But like fat

clients, the user interface can be fairly rich and interactive, taking advantage of some of the processing power on the desktop.

Joe Asks. . .

How do I access standard web services?

Web services are APIs that are exposed by one program (on a server) and consumed by another, possibly independent program. These APIs are programming language independent, and are generally based on XML or JSON (JavaScript Object Notation).

GWT *does* support calling these APIs from your client side code. However usually it's better to make the calls from your server side code instead. The reasons include:

1. Standard web service wire protocols tend to be rather verbose (the worst offender being SOAP), requiring more code and more CPU time to support on the client.
2. JavaScript only allows calls back to the server that originated the JavaScript code. Thus if the web service lives on a different server (for example, at Amazon, EBay, or Google) it would have to go through some kind of HTTP proxy anyway to hop-scotch its way to the destination.

5.2 Calling remote code

Any time you have code running in two different places you'll need a way for them to communicate with each other. The simplest way is through a remote procedure call.

For our purposes, a *remote procedure call* is simply a way the client can execute some logic on the server and get a result back. For example, you might need to retrieve the current mortgage rate, or look up the map coordinates of an address. RMI, .NET Remoting, SOAP, REST, and XML-RPC are all remote procedure call protocols that you might be familiar with.

GWT doesn't use any of those.

5.3 Why a new protocol?

The first reason that GWT has its own protocol is that calls from the browser are *asynchronous*. That's the A in Ajax, remember? You make a request, and it may (or may not) give you back a response sometime in the future. While the request is pending, a GWT application needs to go about its business and adapt to not having that data as best it can.

For example, when you click on a message in GMail, the client requests the text of the message from the server and writes the string *Loading...* in the corner of your browser window. If you get

tired of waiting for the server, you can click on the back button, or select another message, or click on one of your other browser tabs. However, the UI is not locked up as it would be with a synchronous call. While GMail isn't written using GWT (currently), the idea is the same.

Second, GWT's RPC needs to be *simple*. The browser downloads all your client code the first time the user starts your application, and if it's more than 100K or so then people will notice a pause. Besides, JavaScript running in the browser tends to be fairly slow. So implementing a complex protocol like SOAP might just be too much code to download and too slow to be practical.

Finally, JavaScript doesn't support Java-style serialization or dynamic class loading so GWT's RPC can't rely on that. Since none of the existing protocols satisfied all these requirements efficiently, Google invented their own. Fortunately the implementation is open source so you can adapt it easily to meet your needs if necessary.

GWT's RPC wire protocol is not documented. Although you could figure it out from the source code, I recommend you don't do that because it can change without notice. GWT is designed so that your client and server sides are deployed at the same time, as two halves of the same application. If you need a protocol that will remain unchanged across releases then you should not use the GWT RPC.

5.4 GWT RPC basics

GWT is Java centric: Java on the client (converted into JavaScript for browser deployment) and Java on the server (running as a standard Java servlet). Although it's possible to use a different language

on the server side (such as PHP, python, or Ruby), you're fighting the design principles of GWT if you do that. So go with the flow, and just use Java everywhere.

The first step in making a remote procedure call is defining a template that describes the call. Unfortunately you're going to have to do this three times: once on the server, once on the client, and once in an interface that they both share. GWT uses a standard naming convention to connect them all.

Tool support is available to make this a bit easier. For example, GWT Designer from Instantiations and the GWT wizard in IDEA 6.0 let you define the procedure call once, then generate the boilerplate code for you.

Suppose your client needs to check the price of a stock symbol. To prevent chatty communication, you should allow it to request the price of several stocks at once. So, you'll need a method that takes an array of symbol names, and returns an array of doubles. Let's stick that method in an interface called StockService. The three interfaces or classes you need to define are:

Name	**Location**	**Purpose**
interface StockService	client and server	Describes the service (its signature)
class StockServiceImpl	server	Actual code goes here
interface StockServiceAsync	client	Lets you call the service

The only object here with any meat on it is the StockServiceImpl class, so let's look at that one first. If you're following along in Eclipse you should create a new RpcExample project using the regular scaffolding (or just download the samples for this book). Then add this class to the com.xyz.server package, because it will live on the server:

Download RpcProject/src/com/xyz/server/StockServiceImpl.java

```
public class StockServiceImpl extends RemoteServiceServlet
      implements StockService {
```

```
    public double[] getPrices(String[] symbols) {
        double[] result = new double[symbols.length];
        for (int i = 0; i < symbols.length; i++) {
            result[i] = getPrice(symbols[i]);
        }
        return result;
    }
}
```

For this demonstration let's just say that all stocks are priced at $400 and go up $1 every time we check the market (wow, I wish I'd bought some Sun stock when it was $4 a share). This will help illustrate another point later:

Download RpcProject/src/com/xyz/server/StockServiceImpl.java

```
private double price = 400.0;

private synchronized double getPrice(String symbol) {
    // Real code would retrieve the prices here
    return price++;
}
```

Back on the client side, here's the EntryPoint class for the test program. It should look pretty familiar by now, except for a few minor changes to implement the ClickListener interface and change the type of the label widget:

Download RpcProject/src/com/xyz/client/RpcExample.java

```
public class RpcExample implements EntryPoint, ClickListener {
    private Button button = new Button("Click me");
    private HTML label = new HTML();

    public void onModuleLoad() {
        button.addClickListener(this);

        RootPanel.get("slot1").add(button);
```

```
      RootPanel.get("slot2").add(label);
   }

   // ...
}
```

When the button is clicked, the onClick() method is called. This is where you actually do the call to the server:

Download RpcProject/src/com/xyz/client/RpcExample.java

```
   private String[] symbols = { "GOOG", "MSFT", "SUNW" };

   public void onClick(Widget sender) {
      // (1) Create the client proxy. Note that although you are
      // creating the service interface proper, you cast the
      // result to the async version of the interface. The cast
      // is always safe because the generated proxy implements
      // the async interface automatically.
      StockServiceAsync service = (StockServiceAsync) GWT
            .create(StockService.class);

      // (2) Specify the URL at which our service implementation is
      // running. Note that the target URL must reside on the same
      // domain and port from which the host page was served.
      ServiceDefTarget endpoint = (ServiceDefTarget) service;
      endpoint.setServiceEntryPoint(GWT.getModuleBaseURL() + "prices");

      // (3) Create an async callback to handle the result.
      AsyncCallback callback = new AsyncCallback() {
         public void onSuccess(Object result) {
            double[] prices = (double[]) result;
            updatePrices(symbols, prices);
         }

         public void onFailure(Throwable caught) {
            // do some UI stuff to show failure
         }
```

```
    };

    // (4) Make the call. Control flow will continue immediately
    // and later 'callback' will be invoked when the RPC completes.
    service.getPrices(symbols, callback);
  }
}
```

In a real application you'd probably want to do the first 3 steps once and save the results. These objects can be reused for the actual call.

See the references to StockServiceAsync and StockService in the first step? These interfaces can be derived manually or through some kind of tool (e.g., a Ruby script) from the StockServiceImpl class as follows:

Download RpcProject/src/com/xyz/client/StockServiceAsync.java

```
public interface StockServiceAsync {
  void getPrices(String[] symbols, AsyncCallback callback);
}
```

Download RpcProject/src/com/xyz/client/StockService.java

```
public interface StockService extends RemoteService {
  double[] getPrices(String[] symbols);
}
```

Now look at step 3 of the onClick() method. When the getPrices() method completes (assuming it does), then the onSuccess() method in your AsyncCallback is executed. This turns around and calls a method called updatePrices(), which changes the user interface to show the result. Here's the definition of that method:

Download RpcProject/src/com/xyz/client/RpcExample.java

```
private void updatePrices(String[] symbols, double[] prices) {
  String html = "";
  for (int i = 0; i < symbols.length; i++) {
    html += symbols[i] + ": " + prices[i] + "<br/>";
  }
  label.setHTML(html);
```

```
}
```

When you run this program and click the button nothing happens. The hosted shell will show an error message:

```
[TRACE] The development shell servlet received a request for
   'prices' in module 'com.xyz.RpcExample'
[WARN] Resource not found: prices
```

That's because we haven't deployed the servlet code yet. In Web mode you need to deploy the StockServiceImpl servlet to your container, for example Tomcat. See your servlet container's documentation for how to do that. In Hosted mode it's much easier. All you need to do is add this to your module definition (RpcExample.gwt.xml):

Download RpcProject/src/com/xyz/RpcExample.gwt.xml

```
<!-- Specify the servlet class. -->
<servlet path="/prices" class="com.xyz.server.StockServiceImpl" />
```

Now restart your application (click Refresh in the hosted browser), and you should see something like Figure 5.1, on the next page.

Server-based State

Every time you click the button, the prices increment, even if you do it from a different browser. That's because there is one servlet instance shared by all clients. To have different state for different users you'll need to implement some kind of session id, perhaps a random number passed around to all the RPC routines.

5.5 Serialization

In the previous example you passed a few strings and got back some doubles in response. Under the covers, this data transparently

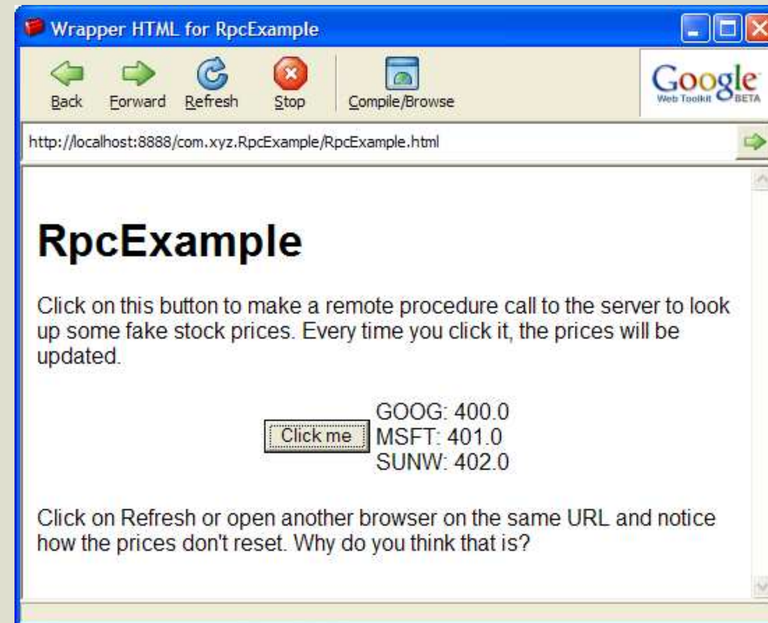


Figure 5.1: RPC example using fake stock prices

underwent *serialization*, a conversion from Java objects to bytes, and back to Java objects on the other side.

Like RMI and .NET Remoting, you are not just limited to primitive types. Any type that is serializable can be passed as a parameter or returned as a result from a remote call. Remember though that GWT's idea of serializable is different from Java's idea of serializable. A type is "GWT serializable" if it:

- is primitive, such as **char**, **byte**, **short**, **int**, **long**, **boolean**, **float**, or **double**;
- is a primitive wrapper (Character, Byte, etc.)
- is String or Date

- is an array of serializable types (including arrays of arrays)
- is a user defined class that contains only serializable fields, or
- implements the IsSerializable marker interface

Most simple types and classes you will want to use will automatically be serializable, for example:

Download RpcProject/src/com/xyz/client/Rect.java

```
public class Rect implements IsSerializable {
   private int height;
   private int width;

   // Must have a zero-arg constructor, or no constructor
   public Rect() {
   }

   public void setHeight(int height) {
      this.height = height;
   }

   // ...
}
```

Collection classes such as Set, List, Map, and HashMap are trickier; you have to use a special annotation in the JavaDoc to tell the GWT compiler what kind of objects will be in the Collection.[1] For example,

Download RpcProject/src/com/xyz/client/MyClass.java

```
public class MyClass implements IsSerializable {
   /**
    * This field is a Set that must always contain Strings.
    *
    * @gwt.typeArgs <java.lang.String>
```

[1]If GWT ever supports generics this will become easier.

```
    */
   public Set setOfStrings;
}
```

Similarly, to annotate parameters and return types:

Download RpcProject/src/com/xyz/server/MyService.java

```
public interface MyService extends RemoteService {
   /**
    * The first annotation indicates that the parameter named 'c' is
    * a List that will only contain Integer objects. The second
    * annotation indicates that the returned List will only contain
    * String objects (notice there is no need for a name, since it
    * is a return value).
    *
    * @gwt.typeArgs c <java.lang.Integer>
    * @gwt.typeArgs <java.lang.String>
    */
   List reverseListAndConvertToStrings(List c);
}
```

In the next chapter we'll change gears a bit to tackle a pesky client-side problem: handling the Back button.

> Here's a word of advice when using remote procedure calls: use them sparingly. Make as few calls as you can, and fetch the minimum amount of data you need to fetch on each call. Be mindful of both the CPU usage on your server, and the amount of data you're sending across the wire. This is just common sense, but it's easy to forget when you're just trying to get your application to work.

Chapter 6

History and Bookmarks

Ever since the first version of NCSA Mosaic was released, web browsers have been blessed with a Back button. Its function was pretty clear when all we had were static HTML pages, but when dynamic web applications appeared, it caused all kinds of headaches.

Bookmarks (sometimes called Favorites) are a particular problem for Ajax apps because the user is interacting with a single web page, and thus many different sections, or states, of the application all have the same URL.

Fortunately GWT solves both of these problems.

6.1 The History Token

The secret to handing the Back button and allowing the user to save useful bookmarks is the history token. The *history token* is just a string that you make up to store whatever state you want it to store. For example, you could use it to store the name of current tab of a multi-tabbed page, or it you could encode some kind of complex state in it.

Google doesn't document the maximum length of the history token, but I recommend keeping it short, say under 100 characters.

The current history token is changed when the user clicks on a Hyperlink object, or the Back or Forward button in the browser. You can also change the history token programmatically by calling the methods History.newItem(), History.back(), or History.forward().

6.2 History Listener

So how do you know when the history token changes? By registering a listener of course. Declare a class that implements HistoryListener and its onHistoryChanged() method, then pass it to History.addHistoryListener(). Many people just use their EntryPoint class for this purpose. All listeners are cleaned up when the application exits.

When the application first starts (when onModuleLoad() is called), the history listener is not called. This gives you a chance to see if there is an initial token or not, perform any special initialization you might need, and then call onHistoryChanged() yourself.

Joe Asks...

Does this mess up regular web anchors?

Web anchors like `<a name="stuff"></a>` and corresponding links like `<a href="#stuff">` do not affect the GWT history stack. However you should avoid using them in a GWT application. If the user bookmarked a URL with a web anchor, when they reloaded that page GWT would interpret the anchor as a history token. It would pass the token to your listener, which wouldn't know how to handle it. If you really want to change states, use the Hyperlink class instead.

6.3 How it Works

To the user, the history token appears in the browser address bar as part of the URL, as in `http://www.gwtpowered.org/#Xyzzy`. If the user browses to that address, the GWT application is loaded and can call History.getToken() to see what it's supposed to do.

Internally, history is managed by a special <*iframe*> tag that you put in your application's HTML page. The GWT scaffolding script adds this for you automatically:

Download HistoryProject/src/com/xyz/public/HistoryExample.html

```
<!-- OPTIONAL: include this if you want history support -->
<iframe id="__gwt_historyFrame" style="width:0;height:0;border:0"></iframe>
```

The browser's Back and Forward buttons simply keep a stack of URLs that you have visited. By carefully manipulating this stack,

GWT makes it correspond to the history tokens that have been set by your application.

As you've probably guessed by now, bookmarks just fall out of this implementation because they record the full URL (including anything after the #). You can also publish the URLs in articles or books as I've done above.

6.4 Example

This example shows how to handle the initial token, set up an event listener, and process events.

Download HistoryProject/src/com/xyz/client/HistoryExample.java

```
public class HistoryExample implements EntryPoint, HistoryListener {

   private Label lbl = new Label();

   public void onModuleLoad() {
      // Create three hyperlinks that change the application's
      // history.
      Hyperlink link0 = new Hyperlink("link to foo", "foo");
      Hyperlink link1 = new Hyperlink("link to bar", "bar");
      Hyperlink link2 = new Hyperlink("link to baz", "baz");

      // If the application starts with no history token, start it
      // off in the 'baz' state.
      String initToken = History.getToken();
      if (initToken.length() == 0)
         initToken = "baz";

      // onHistoryChanged() is not called when the application first
      // runs. Call it now in order to reflect the initial state.
      onHistoryChanged(initToken);
```

```
        // Add widgets to the root panel.
        Panel panel = new VerticalPanel();
        panel.add(lbl);
        panel.add(link0);
        panel.add(link1);
        panel.add(link2);
        RootPanel.get().add(panel);

        // Add history listener
        History.addHistoryListener(this);
    }

    public void onHistoryChanged(String historyToken) {
        // This method is called whenever the application's history
        // changes. Set the label to reflect the current token.
        lbl.setText("The current history token is: " + historyToken);
    }
}
```

When you run this it will show the current token plus three hyperlinks that let you set the token (see Figure 6.1, on the following page). Try clicking on the links and the Back and Forward buttons to satisfy yourself that it works as you'd expect.

Like many Ajax features, GWT makes history easy by hiding all the JavaScript code that is necessary to implement it. But as any programmer knows, sometimes you'll have to pop the hood and get your hands dirty. The next chapter will show you how.

Figure 6.1: History example

Chapter 7

JavaScript Native Interface

Sometimes when you're writing Java code (especially if you're doing systems programming) you find that you have to get closer to the metal and run code outside the Java virtual machine. For example, you might need to access a code library that's in a different language. To do that in Java, you would declare a method as native and then provide the implementation of that method in another language, such as C. This is called the Java Native Interface (JNI).

You can do the same thing with GWT Java code on the client, except instead of C code, the native language for the browser is JavaScript. That's how Google came up with the name JavaScript Native Interface (JSNI).

7.1 Declaring a Native Method

To declare a native method in JSNI, use Java's standard native keyword just like you would with JNI. In JNI, the native C code is in a separate file, compiled separately and dynamically loaded at run time. In JSNI, the native JavaScript code is embedded directly in your Java source in a specially formatted comment:

Download JSNIProject/src/com/xyz/client/Alert.java

```
public class Alert {
   public static native void alert(String msg) /*-{
    $wnd.alert(msg);
    }-*/;
}
```

A JSNI comment block begins with /*-{ and ends with }-*/.

As this example shows, when accessing the browser's window and document objects from JSNI, you must reference them as `$wnd` and `$doc`, respectively. Your compiled script runs in a nested frame, and `$wnd` and `$doc` are automatically initialized to correctly refer to the host page's window and document instead of the frame.

7.2 How it Works

As described in Section 3.2, *Web mode*, on page 13, the GWT compiler converts the client half of your Java program into JavaScript. So normally, when the compiler sees a method declaration, code inside the braces has to go through some kind of translation process. If it's a native method, however, the compiler's job is easier. All it has to do is copy the JavaScript native code directly into the compiled result.

If you've used Microsoft's Visual C++ or the GNU C++ compiler, the effect is much like inline assembler code, except with a much higher level language than assembler.

Since JavaScript is interpreted, any errors in the JavaScript code won't be evident until run-time.

7.3 Calling JSNI from Java

Calling a JSNI method from Java[1] is no different than calling a regular Java method. Here's an example:

---

[1]When I refer to calling to and from Java code on the client, what I really mean is JavaScript code that has been compiled from your Java code in the `.client` package. But you knew that, right?

Download JSNIProject/src/com/xyz/client/JSNIExample.java

```
button1.addClickListener(new ClickListener() {
   public void onClick(Widget sender) {
      Alert.alert("clicked!");
   }
});
```

The caller can't really tell if the method is native or not. This gives you some flexibility in changing your mind later about how the method is implemented.

7.4 Calling Java from JSNI

Going the other way is a little trickier. For example, suppose you pass an object to a JSNI method and you need to access a field or call a method in that object. You'll need to know how the GWT compiler mangles the Java field and method names so you can access them from your own JavaScript code.

Accessing Java fields

Object Oriented purists would say that you shouldn't access fields of a Java class directly because it makes it harder to change the implementation of that class later. But hey, we're writing native code here so we can cut a few corners. The syntax for accessing a Java field is:

obj.@*class*::*field*

where:

obj is the object instance being referenced. For static variables, leave off the instance expression and the trailing period.

class is the fully-qualified name of the class in which the field is declared (or a subclass thereof).

field is the name of the field being accessed.

Invoking Java methods

Calling methods uses a syntax similar to accessing fields, except you must also supply the signature of the method you're calling. The reason for that is that Java methods can be *overloaded*, i.e., two methods can have the same name but take different parameters. The syntax is:

obj`.@`*class*`::`*method*`(`*sig*`)(`*args*`)`

where:

obj is the object instance being referenced. For static methods, omit the instance expression and the trailing period.

class is the fully-qualified name of the class in which the method is declared (or a subclass thereof).

method is the name of the method being called.

sig is the internal Java method signature (see Section 7.4, *Method signatures*).

args is the actual argument list passed to the method.

Method signatures

JSNI method signatures are exactly the same as JNI method signatures except that the method return type is left off. That's because

it's not needed to figure out which overloaded method you're referring to. The following table shows these type signatures:

Type Signature	Java Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L*fully-qualified-class*;	*fully-qualified-class*
[*type*	*type*[]

For example, the Java method:

```
long f (int n, String s, int[] arr);
```

has the following type signature:

```
(ILjava/lang/String;[I)
```

Figure 7.1, on the next page shows the specific rules for how values passing in and out of JSNI code must be treated.

7.5 Example

This code shows some examples of accessing Java fields and methods from within JSNI. It demonstrates passing numbers, strings, booleans, and Java objects into JavaScript. It also shows how a JavaScript method can make a method call on a Java object that was passed in.

Java type	How it appears to JavaScript code
a Java numeric primitive	a JavaScript numeric value, as in `var x = 42;`
`String`	a JavaScript string, as in `var s = "my string";`
boolean	a JavaScript boolean value, as in `var b = true;`
JavaScriptObject	a JavaScriptObject that must have originated from JavaScript code, typically as the return value of some other JSNI method
Java array	an opaque value that can only be passed back into Java code
any other Java Object	an opaque value accessible through special syntax

Figure 7.1: Type Rules

Download JSNIProject/src/com/xyz/client/J2JS.java

```
public class J2JS {
   /** Pass a Java numeric primitive */
   public static void testJ2JSNumeric() {
      int x = 42;
      jsNumeric(x);
   }

   private static native void jsNumeric(int x) /*-{
    $wnd.alert("x is " + x);
    }-*/;

   /** Pass a Java String */
   public static void testJ2JSString() {
```

```
      String s = "my string";
      jsString(s);
   }

   private static native void jsString(String s) /*-{
    $wnd.alert("s is " + s);
    }-*/;

   /** Pass a boolean */
   public static void testJ2JSBoolean() {
      boolean b = true;
      jsBoolean(b);
   }

   private static native void jsBoolean(boolean b) /*-{
    $wnd.alert("b is " + b);
    }-*/;

   /** Pass an arbitrary Java Object */
   public static void testJ2JSObject() {
      MyJavaObject obj = new MyJavaObject();
      jsObject(obj);
   }

   private static native void jsObject(MyJavaObject obj) /*-{
    $wnd.alert("Calling getText(): " + obj.@MyJavaObject::getTextAt(I)(3));
    }-*/;
}
```

If you look at the source code for GWT you'll see that much of it is defined in terms of JSNI. Most GWT programmers will never need to define JSNI methods themselves, but it's nice to know the feature is there if you need it.

The next chapter discusses how to prepare your program for supporting national languages.

Chapter 8

Internationalization (I18N)

Internationalization (i18n for short) is the process of adding a framework to support different national languages to your program. Localization (l10n) occurs when you use that framework to customize the program for each language. National language support is only going to get more important over time as internet usage grows in countries like China, India, and Brazil. Luckily GWT provides full and flexible support as of version 1.1.10 of the toolkit.

Even if your application is not intended for a global audience, it pays to put all your human readable text in one place. You can more easily spell check your messages and ensure consistency between them if they're all in the same file. It also lets non-programmers change the messages to correct grammatical errors or trademark usages without having to modify the code.

The standard Java way to accomplish this is through resource bundles and property files. GWT lets you use these familiar concepts in your web applications as well.

8.1 Constants, Messages, and Dictionary

GWT provides four alternatives for localized text:

Constants

This type can only be used for text that has no substitutions, such as field labels or the names of menu items. It can also be used for numbers, Booleans, and Maps.

ConstantsWithLookup
: This is the same as the Constants interface except you can look up a constant with a dynamic string (more on this later).

Messages
: These are general purpose strings that can include placeholders for substitutions.

Dictionary
: The most flexible but least efficient of all the choices, the Dictionary interface supports dynamically specifying the locale.

Constants, ConstantsWithLookup, and Messages are more efficient than Dictionary because the locale can be determined ahead of time and compiled into the application. The GWT compiler produces a different .cache.html file for each locale, and the appropriate version is loaded at run time. Constants and Messages gain a little more efficiency by pruning resources that aren't actually used in the program.

The Messages interface is the best choice for most applications, so the rest of this chapter will show you how to use that. The API for the other interfaces is similar, and it's possible to use more than one style in the same application.

8.2 Creating the properties file

For this chapter we'll start with a simple program and convert it to use Messages instead of hard-coded strings. Here is the original program:

Download I18NProject/src/com/xyz/client/I18NOrig.java

```
public class I18NOrig implements EntryPoint {
   private Button m_clickMeButton;
```

```
    public void onModuleLoad() {
        RootPanel rootPanel = RootPanel.get();
        {
            m_clickMeButton = new Button();
            rootPanel.add(m_clickMeButton);
            m_clickMeButton.setText("Click me!");
            m_clickMeButton.addClickListener(new ClickListener() {
                public void onClick(Widget sender) {
                    Window.alert("Hello, GWT World!");
                }
            });
        }
    }
}
```

The first step is to find all the strings and copy them into a properties file. Use the standard name=value format. Comments (for example, notes to translators) start with a pound sign (#). Placeholders for substitution parameters are specified with {0}, {1}, and so forth. Here is the properties file for the converted program:

Download I18NProject/src/com/xyz/client/AppMessages.properties

```
m_clickMeButton_text=Click me!
m_helloAlert_text=Hello, {0} World!
```

The file will be needed on the client so put it in the com.xyz.client package. This is the default language file, which will be in English (or your native language). To provide translations for other languages, use ISO language and country code suffixes, for example "_fr" for French or "_fr_CA" for Canadian French. Here's the French version:

Download I18NProject/src/com/xyz/client/AppMessages_fr.properties

```
m_clickMeButton_text=Cliquez-moi!
m_helloAlert_text=Bonjour, Monde de {0}!
```

8.3 Creating the accessor class

If you're familiar with standard Java messages you know they're accessed by string. To get the translated message you call a function and pass it the key string, for example getString("m_clickMeButton_text"). There are two problems with this approach. First, the message keys may be as long as or longer than the message values. Second, if you make a mistake in the message key it can't be caught until run time.

To address these problems and make the web application as small as possible, GWT tweaks the way messages work. Instead of referring to the message using a string, you refer to it using a Java method. This requires you to create a new Java class that has a method corresponding to each message in your properties file. Here is the class for this simple example:

Download I18NProject/src/com/xyz/client/AppMessages.java

```
package com.xyz.client;

import com.google.gwt.i18n.client.Messages;

public interface AppMessages extends Messages {
   String m_clickMeButton_text();

   String m_helloAlert_text(String toolkit);
}
```

Note that there is only one accessor class, no matter how many languages you support in your application.

Maintaining the accessor class by hand is error prone but luckily tool support is available. For example the GWT Designer from Instantiations can take care of many of these housekeeping chores, including finding and extracting localizable strings, and keeping all the files in sync with each other.

8.4 Referring to messages

The next step is to replace the static strings with references to your new messages in the code. After getting a reference to the accessor class using GWT.create, you call the new methods to retrieve the message text. Here's the finished version:

Download I18NProject/src/com/xyz/client/I18N.java

```
public class I18N implements EntryPoint {
   private static final AppMessages MESSAGES = (AppMessages) GWT
         .create(AppMessages.class);

   private Button m_clickMeButton;

   public void onModuleLoad() {
      RootPanel rootPanel = RootPanel.get();
      {
         m_clickMeButton = new Button();
         rootPanel.add(m_clickMeButton);
         m_clickMeButton.setText(MESSAGES.m_clickMeButton_text());
         m_clickMeButton.addClickListener(new ClickListener() {
            public void onClick(Widget sender) {
               Window.alert(MESSAGES.m_helloAlert_text("GWT"));
            }
         });
      }
   }
}
```

8.5 Making module changes

Two minor changes are needed to your module file to complete the transition. First, you need to tell GWT that you are inheriting functionality from the I18N module, and second, you need to define

which languages you have provided translations for. Here is the final modules file after these changes have been made:

Download I18NProject/src/com/xyz/I18N.gwt.xml

```
<module>
        <inherits name="com.google.gwt.user.User"/>
        <entry-point class="com.xyz.client.I18N"/>
        <inherits name="com.google.gwt.i18n.I18N"/>
        <extend-property name="locale" values="fr,de" />
</module>
```

8.6 Running the example

Now it's time to try out the sample. Run it in hosted mode first. If you've imported the I18NProject sample project in Eclipse, select Run → Debug..., and click on the launch configuration titled I18NProject (under Java Application). Then click on Debug. The default (English) version should appear.

To try the French version, edit the URL in the browser to add the suffix ?locale=fr. Press return and the page should refresh, showing French instead of English. Click on the button to see French text in the dialog (see Figure 8.1, on the following page).

Another way to select the locale is to embed it as a meta tag in the HTML file, for example

```
<meta name='gwt:property' content='locale=ja_JP'>
```

This could be handy if the HTML was dynamically generated on the server, for example by a JSP file, based on session settings.

Some developers argue that internationalization should be done from the start of a project, while others say it should be saved until the

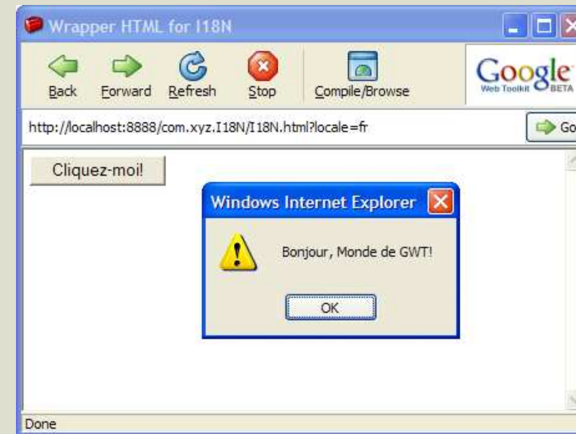


Figure 8.1: French locale

end. In practice it doesn't really matter. Given maintenance releases and newer versions, "the end" is never really the end. Converting an existing project to use message bundles is fairly quick and painless, especially if you have good tool support.

The next chapter discusses the language and library subsets implemented by GWT on the client side.

Chapter 9

Java Emulation

Although your entire GWT application can be written in Java, parts of it are translated into JavaScript for execution in the client browser. This has a couple of big implications:

1. Code targeted to the client must limit itself to the subset of the Java language that is supported by Google's Java to JavaScript translator.
2. In addition, code running in the client can only use a subset of Java Runtime Environment (JRE) library routines that have been ported to JavaScript.

Once you get used to them, these restrictions won't be as bad as you might think at first. It's a bit like writing mobile Java applications, where you have to limit yourself to a certain profile like MIDP. Through JSNI (see Chapter 7, *JavaScript Native Interface*, on page 48), you can also extend the library with native JavaScript code to round out any areas that you might find missing.

9.1 Language subset

The GWT Java to JavaScript translator parses your source code just like a Java compiler would, but instead of compiling it into bytecode, the translator outputs JavaScript code.

Despite having Java in its name, the JavaScript language is actually quite different than Java. For example, it doesn't use classes or type checking. However it's flexible enough that a large subset of the Java 1.4 language can be emulated.

If you use a language feature that isn't supported by GWT, then your code may work fine in hosted mode, but when you run the Java to JavaScript compiler to prepare for web mode you will get an error.

Language level

GWT compiles Java source that is compatible with J2SE 1.4.2 or earlier. Java 5 language features such as enhanced **for** loops, generics, and annotations, are not supported in the current version, but may be supported in the future.

Intrinsic types

byte, **char**, **short**, **int**, **long**, **float**, **double**, Object, String, and arrays are supported. However, there is no 64-bit integral type in JavaScript, so variables of type **long** are mapped onto JavaScript double-precision floating point values. To ensure maximum consistency between hosted mode and web mode, Google recommends that you use **int** variables.

Exceptions

try, **catch**, **finally** and user-defined exceptions are supported as normal, although Throwable.getStackTrace() is not supported for web mode.

Assertions

The GWT compiler parses Java **assert** statements, but it does not emit JavaScript code for them. Asserts are processed in hosted mode, if enabled as a VM argument.

Multithreading and synchronization

JavaScript interpreters are single-threaded, so while GWT silently accepts the **synchronized** keyword, it has no real effect. Synchronization-related library methods are not available, including Object.wait(), Object.notify(), and Object.notifyAll().

Reflection

For maximum efficiency, GWT compiles your Java source into a monolithic script, and does not support subsequent dynamic loading of classes. This and other optimizations preclude general support for reflection. It is possible to query an object for its class name using GWT.getTypeName().

Finalization

JavaScript does not support object finalization during garbage collection, so GWT isn't able to be honor Java finalizers in web mode. Most Java experts advise you not to use finalizers anyway, so this is no great loss.

Strict floating-point

The Java language specification precisely defines floating-point support, including single-precision and double-precision numbers as well as the **strictfp** keyword. GWT does not support the **strictfp** keyword and can't ensure any particular degree of floating-point precision in translated code, so you may want to avoid calculations in client-side code that require a guaranteed level of floating-point precision.

9.2 Library subset

GWT supports a relatively small subset of the JRE library for code targeted to the client. One reason is that the JRE libraries are huge, and another is that many of those features are not supported within the JavaScript sandbox. So forget about file I/O, for example—that has to be done on the server.

If you use a library feature that isn't supported by GWT, then your code will appear to compile correctly in your IDE but when you try to run it in hosted mode you'll get an error in the development shell. So run early and often to catch these problems from the beginning.

Here are some specific areas in which GWT emulation differs from the standard Java runtime. For more information on specific classes and methods, see Section 9.3, *Supported packages*, on the following page.

Regular expressions

The syntax of Java regular expressions is similar, but not identical, to JavaScript regular expressions. So, you'll probably want to be careful to only use Java regular expressions that have the same meaning in JavaScript. See Section 9.4, *Regular Expressions*, on page 68 for a subset supported by both.

Serialization

Java serialization relies on a few mechanisms that are not available in compiled JavaScript, such as dynamic class loading and reflection. As a result, GWT does not support standard Java serialization. Instead, GWT has a RPC facility (described in Chapter 5, *Remote Procedure Calls*, on page 32) that provides automatic object serialization to and from the server for the purpose of invoking remote methods.

9.3 Supported packages

For client code, GWT implements a subset of the `java.lang` and `java.util` packages from JRE 1.4. This section lists the classes and interfaces that are supported, along with any notes on their use.

> As new versions of GWT come out, this list is likely to change. For example Google might add support for some JRE 5.0 and 6.0 methods and classes at some point. But there is plenty of functionality already supported, and an argument can be made for keeping the subset small so the JavaScript download size will be small.

java.lang package

These are the supported classes and interfaces for the `java.lang` package:

Classes:

Boolean, Byte, Character, Class, Double(1), Float(1), Integer, Long(1), Math, Number, Object, Short, String(2), StringBuffer, and System.

Notes:

1. Avoid using as a map key (performance).
2. Regular expressions vary from the standard implementation (see Section 9.4, *Regular Expressions*, on page 68).

Errors and Exceptions:

ArrayStoreException, AssertionError, ClassCastException, Error, Exception, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, StringIndexOutOfBoundsException, Throwable(1), and UnsupportedOperationException.

Notes:

1. Stack traces are not currently supported.

Interfaces:

CharSequence, Cloneable, and Comparable.

java.util package

These are the supported classes and interfaces for the `java.util` package:

Classes:

AbstractCollection, AbstractList, AbstractMap, AbstractSet, ArrayList, Arrays, Collections, Date, HashMap, HashSet, Stack, and Vector(1).

Notes:

1. Does not check for index validity.

Errors and Exceptions:

EmptyStackException, NoSuchElementException, and TooManyListenersException.

Interfaces:

Collection, Comparator, EventListener, Iterator, List, Map, RandomAccess, and Set.

9.3 Supported packages

For client code, GWT implements a subset of the `java.lang` and `java.util` packages from JRE 1.4. This section lists the classes and interfaces that are supported, along with any notes on their use.

As new versions of GWT come out, this list is likely to change. For example Google might add support for some JRE 5.0 and 6.0 methods and classes at some point. But there is plenty of functionality already supported, and an argument can be made for keeping the subset small so the JavaScript download size will be small.

java.lang package

These are the supported classes and interfaces for the `java.lang` package:

Classes:

Boolean, Byte, Character, Class, Double(1), Float(1), Integer, Long(1), Math, Number, Object, Short, String(2), StringBuffer, and System.

Notes:

1. Avoid using as a map key (performance).
2. Regular expressions vary from the standard implementation (see Section 9.4, *Regular Expressions*, on page 68).

Errors and Exceptions:

ArrayStoreException, AssertionError, ClassCastException, Error, Exception, IllegalArgumentException, IllegalStateException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, NumberFormatException, RuntimeException, StringIndexOutOfBoundsException, Throwable(1), and UnsupportedOperationException.

Boundary matchers

Expression	Meaning
^ (caret)	Beginning of line. (As far as I can tell, multi-line mode is not supported.)
$	End of line
\b	Word boundary
\B	Non-word boundary

Quantifiers

Expression	Meaning
*	Zero or more. All matches are greedy.
+	One or more
?	Zero or one
{*n*}	Exactly n times
{*n*,}	n or more times
{*n*,*m*}	Between n and m times inclusive

Miscellaneous

Expression	Meaning
.	Any character
(*x*)	Capturing group
(?:*x*)	Non-capturing group
x(?=*y*)	Zero-width positive look-ahead. (Extra credit if you know what this means.)
x(?!*y*)	Zero-width negative look-ahead
x|*y*	Either *x* or *y*
n	Back reference to captured group. (Don't use '\0' because its meaning is different in Java and JavaScript.)

By following a few simple restrictions described in this chapter, you can pretend that your Java code is running directly in the browser. Code can be shared between the client and server halves of your program, so you don't have to implement the same algorithms in two different languages.

Well, that wraps it up for this book on the Google Web Toolkit. I hope you've found it to be helpful. Now quit reading and go create something great!

Pragmatic Fridays

Timely and focused PDF-only books. Written by experts for people who need information in a hurry. No DRM restrictions. Free updates. Immediate download. Visit our web site to see what's happening on Friday!

More Online Goodness

GWT
Source code from this book and other resources. Come give us feedback, too!

Free Updates
Visit the link, identify your book, and we'll create a new PDF containing the latest content.

Errata and Suggestions
See suggestions and known problems. Add your own. (The easiest way to report an errata is to click on the link at the bottom of the page.

Join the Community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
Check out the latest pragmatic developments in the news.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	`orders@pragmaticprogrammer.com`
Non-English Versions:	`translations@pragmaticprogrammer.com`
Pragmatic Teaching:	`academic@pragmaticprogrammer.com`
Author Proposals:	`proposals@pragmaticprogrammer.com`