The Fundamentals of DTD Design

By Vikram Vaswani

Table of Contents

Running Scared

If you've been playing with XML for a while, you probably already know that XML documents come in two flavours: well−formed and valid.

A well−formed document is one which meets the specifications laid down in the XML recommendation − that it, it follows the rules for element and attribute names, contains all essential declarations, and has properly−nested elements.

A valid document is one which, in addition to being well−formed, adheres to the rules laid out in a DTD or XML Schema. By imposing some structure on an XML document, a DTD makes it possible for documents to conform to some standard rules, and for applications to avoid nasty surprises in the form of incompatible or invalid data.

If you're serious about developing your XML skill set, you're going to bump your head up against DTDs sooner or later − and the arcane commands and symbols you find will make you want to weep and beg for Mommy. Unless, of course, you're armed with your own secret weapon...

This article.

Over the course of the next few pages, I'm going to find out just what makes a DTD tick, with examples, explanations and illustrations that will demystify this simple yet surprisingly−scary piece of the XML puzzle. Strap yourself in, and prepare to meet the beast!

DTD Who?

Let's start with the basics: what's a DTD when it's home, and why do you care?

The first part of the question is easy enough to answer. A DTD, or document type definition, is a lot like a blueprint. Unlike most blueprints, however, it doesn't tell you where the kitchen goes or how the capsule containing the plutonium is to be wired up. Nope, this blueprint is a lot more boring − it tells you exactly how an XML document should be structured, complete with lists of allowed values, permitted element and attribute names, and predefined entities.

DTDs are essential when managing a large number of XML documents, as they immediately make it possible to apply a standard set of rules to different documents and thereby demand conformance to a common standard. However, for smaller, simpler documents, a DTD can often be overkill, adding substantially to download and processing time.

Most XML documents start out as well−formed data − they meet the basic syntactical rules described in the XML specification, and are correctly structured (no overlapping, badly−nested elements or illegal values). However, an XML document which additionally meets all the rules, conditions and structural guidelines laid down in a DTD qualifies for the far cooler "valid" status. Think of it like a free airline upgrade from business to first...except, of course, without the complimentary drinks.

Why do you need to know about this? Well, you don't.

If your day job involves carrying out covert operations for an unnamed intelligence agency or building houses, you'd be better off studying the other sort of blueprint. If, on the other hand, your job involves developing and using XML applications and data, you need to have at least a working knowledge of how DTDs are constructed, so that you can roll your own whenever required.

How's The Weather Up There?

In order to illustrate how DTDs work, consider this simple XML document:

```
<?xml version="1.0"?>
<weather>
<city>New York</city>
<high>26</high>
<low>18</low>
<forecast>rain</forecast>
</weather>
```

As of now, this file is merely well−formed − it hasn't yet been compared to a DTD and declared valid. In order to perform this comparison, I need to link it to a DTD − which I can do by adding a document type declaration referencing the DTD.

```
<?xml version="1.0"?>
<!DOCTYPE weather SYSTEM
"http://www.somedomain.com/weather.dtd">
<weather>
<city>New York</city>
<high>26</high>
<low>18</low>
<forecast>rain</forecast>
</weather>
```

As a result of this addition, the document would be validated against the DTD located at http://www.somedomain.com/weather.dtd

Let's see what this DTD looks like:

```
<!ELEMENT weather (city, high, low, forecast)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT high (#PCDATA)>
<!ELEMENT low (#PCDATA)>
<!ELEMENT forecast (#PCDATA)>
```

To the untrained eye − gibberish. But give it a couple of minutes...

Rainy Days

Once you have an XML document and a DTD linked together, an XML parser can verify the document against the DTD and let you know if it finds errors. A number of tools are available online to perform this validation − my favourite is the XML Spy editor, available at http://www.xmlspy.com/ , although you can also try out expat, at http://sourceforge.net/projects/expat/, and rxp, at http://www.cogsci.ed.ac.uk/~richard/rxp.html

Here's what rxp has to say when I run it on the XML document above.

```
$ rxp −V weather.xml
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE weather SYSTEM "weather.dtd">
<weather>
<city>New York</city>
<high>26</high>
<low>18</low>
<forecast>rain</forecast>
</weather>
```

In other words − no error.

Let's suppose I altered the XML document instance a little, by modifying one of the element names and adding a new element.

```
<?xml version="1.0"?>
<!DOCTYPE weather SYSTEM "weather.dtd">
<weather>
<city>New York</city>
<high>26</high>
<low>18</low>
<weekly_mean>21</weekly_mean>
<daily_forecast>rain</daily_forecast>
</weather>
```

While this version of the document is still well−formed, it no longer follows the rules laid down in "weather.dtd" and hence cannot be considered valid − which is why rxp barfs and generates a list of errors.

```
$ rxp −V weather.xml
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE weather SYSTEM "weather.dtd">
<weather>
<city>New York</city>
```

```
<high>26</high>
<low>18</low>
Warning: Start tag for undeclared element weekly_mean
in unnamed entity at line 7 char 14 of weather.xml
Warning: Content model for weather does not allow element
weekly_mean here
in unnamed entity at line 7 char 14 of weather.xml
<weekly_mean>21</weekly_mean>
Warning: Start tag for undeclared element daily_forecast
in unnamed entity at line 8 char 17 of weather.xml
<daily_forecast>rain</daily_forecast>
</weather>
```

Incidentally, it's also possible to place the DTD within the XML document itself. Although this is quite rare − the DTD is usually stored in a central place so that it can be referenced by different XML documents − you should know how to do it in case you're ever home on a Saturday evening and feel like experimenting.

```
<?xml version="1.0"?>
<!DOCTYPE weather [

<!ELEMENT weather (city, high, low, forecast)>

<!ELEMENT city (#PCDATA)>

<!ELEMENT high (#PCDATA)>

<!ELEMENT low (#PCDATA)>

<!ELEMENT forecast (#PCDATA)>

]>

<weather>
<city>New York</city>
<high>26</high>
<low>18</low>
<forecast>rain</forecast>
</weather>
```

Simply Elementary

Now that you know the basics of linking and validating XML data against DTDs, let's focus in on the different components that actually go into a DTD.

All XML documents consist of some combination of elements, attributes, entities and character data. In case you've forgotten what these are, here are some quick definitions:

An element, which is the basic unit of XML, consists of textual content (character data), enhanced with descriptive tags.

```
<dinosaur>Stegosaurus</dinosaur>
```

An attribute is a name−value pair which provides additional descriptive parameters or default values to an element.

```
<person sex="male">Spiderman</person>
```

An entity is an XML construct, referenced by name, which stores text, images and file references; it is primarily used as a mechanism to store and reuse content which appears in multiple places within an XML document.

```
<!ENTITY copyright "This material copyright Melonfire, 2001.
All rights
reserved.">
```

Each of these basic constructs can be defined in a DTD. I'll begin with element declarations, which typically look like this:

```
<!ELEMENT elementName (contentType)>
```

As an example, consider the "forecast" element from the previous example:

```
<!ELEMENT forecast (#PCDATA)>
```

In English, this declares an element with name "forecast" and content of the form "parsed character data" (in case you're wondering, this means that the parser will parse the contents of the "forecast" element, automatically processing its child elements and entities).

The alternative to parsed character data is regular character data, which will be treated as literal text by the parser without any further processing. Here's an example of this type of element declaration:

```
<!ELEMENT greeting (#CDATA)>
```

In case you don't want to specify a content type, you can escape without making a decision by allowing any content.

```
<!ELEMENT address ANY>
```

Of course, doing this kinda negates the purpose of having a DTD in the first place...

If an element contains nested child elements, it's necessary to specify these element names within the declaration. In the following example,

```
<?xml version="1.0"?>

<book>
<author>Stephen King</author>
<title>Bag Of Bones</title>
<price>$9.99</price>
</book>
```

the "book" element contains four child elements nested within it − which is why its element declaration in the DTD looks like this:

```
<!ELEMENT book (author, title, price)>
```

XML also allows for so−called empty elements − essentially, elements which have no content and therefore do not require a closing tag. Such elements are closed by adding a slash (/) to the end of their opening tag. Consider the following XML snippet

```
<?xml version="1.0"?>
<rule>Every sentence ends with a <period /></rule>
```

and then take a look at the corresponding empty element declaration:

```
<!ELEMENT period EMPTY>
```

What's The Frequency, Bobby?

A number of special symbols can be added to an element declaration in order to define its frequency and order, or the frequency and order of its child elements. Here's a quick list:

```
symbol description
------------------------------------------------------
+ one or more occurrence(s)
* zero or more occurrence(s)
? zero or one occurrence(s)
| choice
```

If you're familiar with regular expressions, you'll feel right at home with these symbols − they're almost identical to the symbols used to build regular expression patterns.

Let's take this for a quick spin. Consider the following revised XML document

```
<?xml version="1.0"?>
<!DOCTYPE weather SYSTEM "weather.dtd">
<weather>

<city>New York</city>
<high>26</high>
<low>18</low>
<forecast>rain</forecast>

<city>Boston</city>
<forecast>snow</forecast>

<city>London</city>
<high>32</high>
<forecast>sun</forecast>

</weather>
```

and then take a look at its associated DTD

```
<!ELEMENT weather (city, high*, low*, forecast)+>
<!ELEMENT city (#PCDATA)>
<!ELEMENT forecast (#PCDATA)>
<!ELEMENT high (#PCDATA)>
<!ELEMENT low (#PCDATA)>
```

How did I come up with this? It's simple − you just have to take it step by step.

The first thing to do is allow for more than one "city" block within the "weather" element.

```
<!ELEMENT weather (city, high, low, forecast)+>
```

Next, the "high" and "low" elements must be made optional.

```
<!ELEMENT weather (city, high*, low*, forecast)+>
```

And Bob's your uncle!

The | operator sets up a list of alternatives, and comes in handy when an element must contain any one of a finite list of alternatives. Consider the following XML document,

```
<?xml version="1.0"?>

<addressbook>

<record>
<name>John Smith</name>
<street>24, Main Street</street>
<city>Poodle Springs</city>
<zip>16628</zip>
<country>USA</country>
<tel>
<home>947 3838</home>
</tel>
</record>

<record>
<name>Sherlock Holmes</name>
<tel>
<home>827 3483</home>
</tel>
<fax>
<home>364 2929</home>
</fax>
<email>
<home>holmes@greatdetectives.org</home>
</email>
</record>

<record>
```

```
<name>Jane Doe</name>
<email>
<work>jane@somedomain.com</work>
</email>
</record>

</addressbook>
```

and take a look at the corresponding DTD, specifically at the declarations for the "tel", "fax" and "email" elements, which may contain either "home" or "work" nested child elements.

```
<!ELEMENT addressbook (record+)>
<!ELEMENT record (name, street?, city?, zip?, country?, tel?,
fax?, email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT tel (home | work)>
<!ELEMENT fax (home | work)>
<!ELEMENT email (home | work)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT work (#PCDATA)>
```

The | operator also comes in handy when defining elements which are of so−called "mixed" type − they can contain either data or other elements. Here's an example:

```
<?xml version="1.0"?>
<surrealism>
The elongated <color>blue</color> <animal>fox</animal> jumped
over the
<color>green</color> <vegetable>pumpkin</vegetable> and
morphed into
<personality>Richard VIII</personality>
</surrealism>
```

Pay close attention to the "surrealism" element, which can contain either character data or any one of the listed elements:

```
<!ELEMENT animal (#PCDATA)>
<!ELEMENT color (#PCDATA)>
<!ELEMENT personality (#PCDATA)>
```

```
<!ELEMENT surrealism (#PCDATA | animal | color | personality |
vegetable)*>
<!ELEMENT vegetable (#PCDATA)>
```

Obviously, all these symbols can also be combined to create weird and wonderful rules for the document to follow. An example awaits you at the end of the article...but first, attributes.

Turning Up The Heat

Just as you can declare elements, a DTD also allows you to define the attributes attached to each element. An attribute declaration typically looks like this:

```
<!ATTLIST elementName attributeName contentType modifier>
```

In order to demonstrate, consider the following XML document, which adds a couple of attributes to the previously declared XML elements.

```
<?xml version="1.0"?>
<weather>

<city state="NY">New York</city>
<temperature>
<high units="celsius">23</high>
<low units="celsius">10</low>
</temperature>
<forecast>sun</forecast>

</weather>
```

Here's the corresponding DTD:

```
<!-- element declarations -->
<!ELEMENT weather (city, temperature, forecast)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT forecast (#PCDATA)>
<!ELEMENT high (#PCDATA)>
<!ELEMENT low (#PCDATA)>
<!ELEMENT temperature (high, low)>

<!-- attribute declarations -->
<!ATTLIST city state CDATA #REQUIRED>
<!ATTLIST high units CDATA #REQUIRED>
<!ATTLIST low units CDATA #REQUIRED>
```

Let's examine the first one a little more closely:

```
<!ATTLIST city state CDATA #REQUIRED>
```

In English, this declares that the element "city" has an attribute named "state" containing character data. The additional #REQUIRED modifier indicates that this is a required attribute − failure to include it will render the XML document invalid.

A number of different content types are available for attributes. You're already familiar with character data − here's a quick list of the others:

```
type description
----------------------------------------
CDATA character data
ID unique identifier
IDREF reference to unique identifier of another element
IDREFS list of identifiers
NMTOKEN string literal or token
NMTOKENS list of tokens
ENTITY entity
ENTITIES list of entities
```

You can also specify a list of allowed attribute values by enclosing them in parentheses and separating them with the | operator. The following attribute declaration does just that, limiting the list of allowed values for the "units" attribute to either "celsius" or "fahrenheit".

```
<!ATTLIST high units (celsius | fahrenheit) #REQUIRED>
```

A number of modifiers are available, each applying a special characteristic to the attribute. For example, you can specify a default value for the attribute by enclosing it in quotes; this default value is used if the attribute is absent.

```
<!ATTLIST high units (celsius | fahrenheit) "fahrenheit">
```

The #IMPLIED modifier is used to declare a particular attribute as optional.

```
<!ATTLIST high units #IMPLIED>
```

And finally, the #FIXED keyword is used to fix an attribute value to something specific, allowing the XML document author no choice in the matter.

```
<!ATTLIST high units (celsius | fahrenheit) #FIXED
"fahrenheit">
```

If an element has more than one attribute, you can declare them all within the same attribute declaration. Consider the following XML document,

```
<?xml version="1.0"?>
<movie id="42" genre="sci-fi">
<title>Star Wars</title>
<cast>Mark Hamill, Carrie Fisher, Harrison Ford</cast>
<director>George Lucas</director>
</movie>
```

and its associated DTD, which demonstrates how this works.

```
<!-- element declarations -->
<!ELEMENT movie (title, cast, director)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT cast (#PCDATA)>
<!ELEMENT director (#PCDATA)>

<!-- attribute declarations -->
<!ATTLIST movie
id CDATA #REQUIRED
genre CDATA #REQUIRED
```

Let's move on to entities.

An Entity In The Attic

XML entities are a bit like variables in other programming languages − they're XML constructs which are referenced by a name and store text, images and file references. Once an entity has been defined, XML authors may call it by its name at different places within an XML document, and the XML parser will replace the entity name with its actual value.

XML entities come in particularly handy if you have a piece of text which recurs at different places within a document − examples would be a name, an email address or a standard header or footer. By defining an entity to hold this recurring data, XML allows document authors to make global alterations to a document by changing a single value.

An entity declaration typically looks like this:

```
<!ENTITY entityName entityValue>
```

Consider the following XML document,

```
<?xml version="1.0"?>
<article>

<title>The Fundamentals Of DTD Design)</title>
<abstract>A discussion of DTD syntax and
construction</abstract>
<body>

Article body goes here

</body>

</article>
```

and then take a look at its DTD, which sets up the entity.

```
<!-- element declarations -->
<!ELEMENT abstract (#PCDATA)>
<!ELEMENT article (title, abstract, body)>
<!ELEMENT body (#PCDATA)>
<!ELEMENT title (#PCDATA)>

<!-- entity declarations -->
<!ENTITY copyright "This material copyright Melonfire, 2001.
All rights
```

```
reserved.">
```

Entity declarations can contain XML markup in addition to ordinary text − the following is a perfectly valid entity declaration:

```
<!ENTITY copyright "This material copyright
<link>Melonfire</link>,
<publication_year>2001</publication_year>. All rights
reserved.">
```

Entities may be "nested"; one entity can reference another. Consider the following entity declaration, which illustrates this rather novel concept.

```
<!ENTITY company "Melonfire">
<!ENTITY year "2001">
<!ENTITY copyright "This material copyright , . All rights
reserved.">
```

Note, however, that an entity cannot reference itself, either directly or indirectly, as this would result in an infinite loop (most parsers will warn you about this.) And now that I've said it, I just know that you're going to try it out to see how much damage it causes.

The Old Popcorn Trick

And that just about covers everything I have to say on the topic. Before I go, though, I'd like to run through a composite example illustrating everything you've learned thus far.

Take a look at the following XML document

```
<?xml version="1.0"?>
<!DOCTYPE review SYSTEM "movie.dtd">
<review id="42">

<header>
<title>Pearl Harbor</title>
<cast>Ben Affleck, Josh Hartnett and Kate Beckinsale</cast>
<director>Michael Bay</director>
<duration units="m">167</duration>
<genre>Drama</genre>
<slug>War Games</slug>
<author>J. Doe</author>
<date>2001-08-08</date>
</header>

<body>
<para>On December 7, 1941, Japan unexpectedly attacked the
American naval
base at Pearl Harbor, hoping to gain the initiative in the war
against
Europe. As it turned out, the attack had the effect of
galvanizing the
<quote>sleeping American giant</quote>, resulting in the utter
rout of the
Japanese and German armies and bolstering America's dominant
role in world
politics. </para>

<para>While <title>Pearl Harbor</title>'s love story may seem
unbelievably
trite, the effects are most certainly not. The Japanese attack
on the naval
port is described in tremendous detail, and is perhaps the
most compelling
reason to watch this film. With over forty minutes of reel
time devoted to
the attack, you've probably never seen anything like it
before; it's a
visual spectacle that hits home more than any written
description ever
```

```
will. Bay's direction is superb − he knows just where to put
the camera,
and he always gets the money shot − and the cinematography and
visuals −
especially those shot in the train station, with steam
billowing out in the
background − simply gorgeous. </para>

<para>While I think the love story embedded within
<title>Pearl
Harbor</title> isn't really all that compelling −
<title>Moulin
Rouge</title> did it better − this is still a film worth
watching, if only
to understand a little bit of history! </para>
</body>

</review>
```

and then see if you can put together a DTD for it. Here's my version:

```
<!−− element declarations −−>
<!ELEMENT review (header,body) >
<!ELEMENT header
(title,cast,director,duration,genre,slug,author,date) >
<!ELEMENT title (#PCDATA) >
<!ELEMENT cast (#PCDATA) >
<!ELEMENT director (#PCDATA) >
<!ELEMENT duration (#PCDATA) >
<!ELEMENT genre (#PCDATA) >
<!ELEMENT slug (#PCDATA) >
<!ELEMENT author (#PCDATA) >
<!ELEMENT date (#PCDATA) >
<!ELEMENT body (para+) >
<!ELEMENT para (#PCDATA|quote|title)* >
<!ELEMENT quote (#PCDATA) >

<!−− attribute declarations −−>
<!ATTLIST review id CDATA #REQUIRED >
<!ATTLIST duration units (m | h) "m" >
```

And that's about it from me. In case you're interested in finding out about the more arcane aspects of DTDs − notations, parameter entities and overrides − you should consider checking out the following links.

XML Basics, at http://www.devshed.com/Client_Side/XML/XMLBasic1/

The W3C's XML specification, at http://www.w3.org/TR/2000/REC−xml−20001006

The DocBook DTD, a specification for technical manuals and material, at http://www.oasis−open.org/docbook/xml/4.1.2/index.shtml

A collection of sample DTDs, at http://www.dtd.com/

An interesting article on DTD construction, at http://www.xml.com/pub/a/2000/06/xmleurope/schemas.html

If, on the other hand, all you were looking for was a working knowledge of DTDs to get you though your day, I hope you found it here. I'll be back soon with another article on a related technology, XML Schema, which is quickly gaining followers on account of its ease of use and powerful data−validation capabilities (think of it as DTDs on steroids, but without the nasty symbols). Until then, though...be good!