Document Type Definitions

A Document Type Definition (DTD) is an optional part of an XML document that defines the document's exact layout and structure. Although not essential, there are several advantages to using a DTD, and these advantages become more obvious, and of greater value, as the size and complexity of the XML increases. Most non-trivial applications of XML benefit from a DTD, so most document authors need to know how to write them, and most software developers working with XML need to know how to take advantage of them. Among the more important things a DTD describes are:

- the elements that can appear in a document;
- the order in which they can appear;
- the ways some elements can be contained within instances of others;
- which elements are optional and which are mandatory;
- which elements attributes apply to;
- whether attributes they are mandatory or optional;
- whether attributes can have default values;
- where free text can be placed in the document

This article introduces DTDs, and explores some of the options available to free and open source developers who need to use them.

The Benefits of using DTDs

There are two main reasons for an XML author to write a DTD for their document. The first of these is documentation. A developer can look at a document with a DTD and immediately understand the structure of the data. A DTD can formally state that, for example, a product number contains a manufacturer's code, followed by a batch number, followed by a part number, followed by an optional colour code.

The second reason to use a DTD is to enable what is known as validation. The process of document validation involves passing an XML document through a processor which reads the DTD, then examines the XML data to ensure that elements appear in the right order, that mandatory elements and attributes are in place, that no other elements or attributes have been inserted where they shouldn't have been, and so on. Working with validated data makes life much easier for a developer. If data is known to be valid, it is completely predictable. For the product number example given above, a developer can write code to read each of the first 3 pieces of data from a validated document, then do a check to see if the optional colour code has been provided before attempting to read that. There is no need to clutter the code with error checks or assertions; if the document validates, one error check around the entire parsing code is all that's required. That error check can throw an "Internal Error" and the developer can be confident that he is the only one who is ever going to see it.

The Format of a DTD

A document's DTD is held near the top of the XML file, inside the document type declaration line. The actual DTD information can be embedded directly into the document type declaration itself, or stored in an external file which is referenced via a URI. An external file can be either on the local machine or somewhere on the Internet. An external DTD can be shared by several XML files, or made available for public reference. For the purposes of this article I'll show all my examples with the DTD internal to the XML document.

The format of the DTD is really quite simple, and is formally defined in the XML specification. A document type declaration (for a document of type printer) with a DTD embedded looks like this:

```
<!DOCTYPE printer [
dtd information here
]>
```

The actual DTD goes in between the square brackets, and as an example I'll expand this printer document with the lines of the DTD that define the elements.
An element declaration appears on an <!ELEMENT…> line, and contains information stating which other elements can appear inside it. So, for example:

```
<!DOCTYPE printer [
<!ELEMENT printer  (make, model, connection)>
...
]>
```

This line says that the printer element must contain – and can only contain – a make element, followed by a model element, followed by a connection element, like this:

```
<printer>
  <make>
    ...
  </make>
  <model>
    ...
  </model>
  <connection>
    ...
  </connection>
</printer>
```

The make, model and connection elements must all appear (i.e. none are optional) and they must appear in that order. No other elements are allowed. To make an element optional, follow it with a question mark in the DTD:

```
<!DOCTYPE printer [
<!ELEMENT printer  (make, model, connection?)>
...
]>
```

The question mark means the connection element can appear zero or one times, which, experienced developers will note, is the exact same meaning as it has in regular expressions – the preceding element may appear exactly once or not at all. This is no coincidence, as we shall see later.
The next two elements of our DTD might look like this:

```
<!DOCTYPE printer [
<!ELEMENT printer  (make, model, connection?)>
<!ELEMENT make     (#PCDATA)
<!ELEMENT model    (#PCDATA)
...
]>
```

It doesn't matter which order you declare your elements in inside a DTD. The make and model elements have been declared as containing #PCDATA which means parsed

character data – in other words, free text. The name parsed character data means just that: the characters will be parsed by the XML parser. That means if the text contains angle brackets and other XML significant characters, they must be escaped – coded as < and the like – otherwise the parser will consider them an error.
Our connection element might be defined like this:

```
<!ELEMENT connection (name+, alias*, option*)>
<!ELEMENT queuename  (#PCDATA)>
<!ELEMENT alias      EMPTY
<!ELEMENT option     (#PCDATA)>
```

which says a connection element must contain one or more name elements, followed by zero or more alias elements, followed by zero or more option elements. It's those regular expression characters again - + means one or more, * means zero or more. An element defined as EMPTY can have no text and no child elements, like XHTML's
 break element.
DTDs also control the names and types of the attributes which can be attached to the elements. A simple example of an attribute list might be:

```
<!ATTLIST connection
          type CDATA #required>
```

which says the connection element has a type attribute, that it contains character data, and that the attribute is mandatory. If the word #required was replaced with #implied then the attribute would be optional. It is also possible to define an attribute as having an enumeration, which is one of a set number of values:

```
<!ATTLIST connection
          type (usb|network) "usb">
```

The type attribute is now only valid if it contains one of two values: "usb" or "network". It defaults to "usb" if the attribute is not supplied. A connection element with a type attribute set to anything other than those two options would be rejected by an XML validator.
There are several other more specialised types of attribute, details of which are in the XML specification.
A complete, valid XML document might look like this:

```
<!DOCTYPE printer [
<!ELEMENT printer    (make, model, connection)>
<!ELEMENT make       (#PCDATA)
<!ELEMENT model      (#PCDATA)
<!ELEMENT connection (queuename+, alias*, option*)>
<!ELEMENT queuename  (#PCDATA)>
<!ELEMENT alias      EMPTY
<!ELEMENT option     (#PCDATA)>

<!ATTLIST connection
          type   (usb|network) "usb">
<!ATTLIST alias
          server CDATA         #implied
          name   CDATA         #required>
]>
<printer>
  <make>
    Canon
```

```
  </make>
  <model>
    i250
  </model>
  <connection type="usb">
    <queuename>
      canon_i250_color
    </queuename>
    <queuename>
      canon_i250_mono
    </queuename>
    <option>
      --with_A4_paper
    </option>
  </connection>
</printer>
```

In this case the make and model are supplied and contain only text, as required. The connection has 2 queuenames. Just one would be fine, but it must have at least one. There are no alias elements, which is fine because zero is an acceptable number of those. I also chose to supply a single option element.
Have another look at that alias definition - there isn't such an element in the XML document itself, so the only information we have about it is in the DTD. From this we know the element is optional, and we know that if supplied it will be an empty element with at least one attribute. The mandatory attribute is a name, and there's an optional attribute which specifies a server name. Given the context of an XML document which describes a printer, we can infer from the DTD quite a bit of information about how a printer might have an alternative name or access path. This is what I meant when I referred to the DTD as a documentation tool.

Validation using a DTD

Now that we've seen what a DTD looks like, and how it defines the structure of an XML document, we can look at how a DTD can be used to validate the document. Validation requires a tool that understands the format of a DTD, and that knows how to parse an XML document and compare its structure to that defined in the DTD. There are a number of free software tools which can do this, and which one you choose largely depends on what environment you're working in, and where in the process you need the validation to take place.
Let's start with a command line tool which simply checks an XML document against its DTD and returns a value indicating whether the document is valid or not. The xmllint tool from the LibXML2 project can do that with a command like:

```
xmllint –valid –noout printer.xml
```

which returns 0 if the document is valid, or an error code if it's not. Try adding an invalid attribute somewhere in the example XML document to see how xmllint reacts to it.
A typical validation requirement for an application written in code or script is to validate an XML document as the document is parsed. The objective is normally to read the document from a disk file or stream, parse it, then process the data as the application requires. As long as the parsing phase doesn't complain as the XML document is read in, the application can take it for granted that the XML document it is working with is valid.

Which parser package you use depends on which language you are working with. In many cases the LibXML2 package is a fine starting point. It contains library code as well as the xmllint binary used above, and runs on many platforms. The development package is included with almost every Linux distribution. There's a simple example of document validation from the library's native C here at the XMLSoft examples page. http://www.xmlsoft.org/examples/index.html#parse2.c. LibXML2 bindings also exist for C++, Perl, Python, Ruby, Tcl and others.
An alternative to LibXML2 is the Xerces package, which is the official XML package used and supported by the Apache project. Xerces has been implemented in both C++ and Java, and bindings are available for many languages. The C++ package has been ported to just about any platform you care to mention. Example code, including plenty that demonstrates validation, can be found on the Apache website. http://xml.apache.org/xerces-c/samples.html.
For script writers, native validation libraries are sometimes available. I showed how the Python based xmlproc parser from the PyXML package can be used to validate a document in my DOM article at DevChannel. If a native library isn't available, bindings for one or more of the popular library packages normally are. If you use Perl there's an official binding for the Xerces package (http://xml.apache.org/xerces-p/).
For Tcl scripters the TclXML package (http://tclxml.sourceforge.net/tclxml.html) provides several choices, including bindings to LibXML2.
It is important to realise that validation is rarely the real aim of using these sorts of development tools. The ultimate aim will normally be to process the data in the XML document in some application specific way, and that application logic would normally dictate which XML handling package is the most suitable for any particular task. This is why there are so many options available for document validation; it is simply one extremely useful feature these sorts of packages offer.
I should also point out that the popular expat parser does not do validation. Expat is used as the default XML parser by many languages, and is frequently bundled with language libraries and installations. Expat is invariably chosen because of its combination of maturity, speed and lightness, but developers needing XML document validation need to look elsewhere.

Conclusion

This article hasn't gone into the fine details of the Document Type Definition, and there are a few more features to explore. However, there is little which is more complicated than what has been presented here. DTDs are pretty simple and it doesn't take long to get the hang of writing them.
The free and open source software communities have plenty of facilities to hand for DTD based document validation, and considering the benefits DTDs bring to XML-based software developers, it makes sense to use them where possible.

Sidebar - Schemas

DTDs are a fine starting point when it comes to controlling XML document structure, but they are somewhat limited. A more powerful, more flexible and massively more complicated approach is provided by XML Schemas.
Schemas do much the same job as DTDs in that they provide a way of confirming that a document matches a certain definition, only they attempt to do the job in a much more comprehensive way. Schemas allow control over a document's content, as well as its structure. You can specify that an element be of a certain type, such that it must contain an integer, a date, a boolean value, or any one of a couple of dozen other built in types. You can also create your own complex types using combinations of built in

types and regular expressions. An XML validator can then restrict content to data which matches the pattern. Schemas also allow finer controls over elements and attributes, so you can control, for example, the exact number of times an element might appear.

Schemas are extremely powerful but have been somewhat slow to catch on. Their complicated nature makes them both difficult to use and difficult to implement from the tool developer's point of view. The Xerces C++ package has arguably the most mature support for Schemas in the free software world, with LibXML2 now offering similar facilities.