IBM www.ibm.com/ XML

Education: Papers

Building an XML Application, Step 1: Writing a DTD

Doug Tidwell
IBM XML Technical Strategy Group, SmartGuide Development
October 1998

Abstract: One of the main tasks in creating an XML application is writing a Document Type Definition (DTD). The DTD lets us define the different pieces of data we plan to model, along with the relationships between them. The ability to include this semantic information is the source of XML's power, and its main advantage over HTML. In this article, we'll build a DTD as the first step in building an XML application; future articles will expand on this work.

Mary, Mary, quite contrary, how does your data grow?

When writing a DTD, the first place to start is with your data in its current form. How are your data items arranged currently? Are they in a relational database? In a flat file? On yellow sticky notes plastered to your wall? If you don't have much data, or your data is in an easily manipulatable format, you may be able to restructure your data source to make the job of writing your XML application easier. On the other hand, if you have lots of data or your current format is inconvenient, you'll probably have to use your data as is.

Sample database structure

For our examples in this article, we'll create an XML application for the Xtreme Travel Agency, a company specializing in outdoor tours for the active/foolhardy set. This article involves creating a DTD for the data specified in the `flights` database. The structure of the database is shown in the following table:

Structure of Database Table `flights`

Column Name	Sample Data	Column Name	Sample Data
DepartFrom_1	Chicago	DepartFrom_2	Palm Springs
DepartTime_1	January 10 6:30 AM	DepartTime_2	January 15 11:50 AM
ArriveIn_1	Palm Springs	ArriveIn_2	Chicago
ArriveTime_1	January 10 11:03 AM	ArriveTime_2	January 15 9:24 PM
Airline_1	American #303	Airline_2	American #1250

Taking a look at our database, there are several troubling things. One is that several fields contain more than one piece of information. For example, the `DepartTime_1` field contains both the date and time of the departing flight. If we want to look at the starting time and ending time of the flight to calculate the duration of the flight, there's no reliable way to do that. There also seems to be some redundant information (`DepartFrom_2` and `ArriveIn_1` always have the same value, for example).

One of the first questions we would ask is whether any of these concerns affect the data we want to model in our application. If so, it's worth finding out if we can change the structure of the database.

Most likely, however, changes to the database won't be allowed, and we'll have to live with what we have.

Modeling our data

Assuming that we can't change the database, our first step will be to create a simple DTD that mirrors the structure of the database. We've already listed the fields that make up each record in the `flights` table. There are some other rules we can state about the database:

- The table we're concerned with is called `flights`.
- Each record in the `flights` database represents a complete itinerary: an outbound flight and a returning flight.
- For each itinerary, there are ten fields, the names of which are listed in the previous table.

As a first pass at a DTD, we'll create the tags `<flights>`, `<itinerary>`, etc., specifying the relationships among items we just outlined. Before we do that, we'll discuss the basics of DTD syntax.

DTD basics

Each statement in a DTD uses the `<!XML DTD>` syntax. This syntax begins each instruction with a left angle bracket and an exclamation point, and ends it with a right angle bracket. (The fact that an XML DTD isn't specified in XML is being addressed by several proposals, including the Document Content Description specification from IBM, Textuality, and Microsoft.) As we mentioned earlier, our first pass at a tag set will look like this:

```
<flights>
  <itinerary>
    <departFrom_1 />
    <departTime_1 />
    <arriveIn_1 />
    <arriveTime_1 />
    <airline_1 />
    <departFrom_2 />
    <departTime_2 />
    <arriveIn_2 />
    <arriveTime_2 />
    <airline_2 />
  </itinerary>
</flights>
```

Defining a document element

Our document element, the outermost tag, will be the `<flights>` tag:

```
<!ELEMENT flights (itinerary)+>
```

The element declaration defines the name of the tag (`flights`, in this case), and the *content model* for the tag. The `+` notation above means the `<flights>` tag can contain one or more `<itinerary>` tags.

XML occurrence indicators

In addition to the plus sign from our previous example, there are other occurrence indicators:

XML Occurrence Indicators	
Indicator	**Meaning**
?	The content must appear either once, or not at all.
*	The content can appear one or more times, or not at all.
+	The content must appear at least once, and may appear more than once.
[none]	The content must appear once, exactly as described.

These indicators can be combined with parentheses in any order to create complex expressions. For example, an element defined with the content model

```
(a, (b | c | d), e)*
```

can contain any of the following content:

```
<a /><b /><e />
<a /><d /><e />
<a /><c /><e /><a /><c /><e />
<a /><b /><e /><a /><c /><e /><a /><d /><e />
```

Defining more tags

We've already decided that for our first DTD, we'll simply describe the format of the database. That means we'll define a tag for each field in the row. Because rows are represented by the `<itinerary>` tag, we'll include all of the field tags:

```
<!ELEMENT itinerary (departFrom_1, departTime_1, arriveIn_1,
                     arriveTime_1, airline_1, departFrom_2,
                     departTime_2, arriveIn_2, arriveTime_2,
                     airline_2)>
```

Notice that the definition of the `itinerary` element doesn't use any occurrence indicators; this means that the elements must occur in exactly this order, and will occur only once.

Now that we've defined the tag that contains the data for each row in the database, we can start defining the individual tags. Each of these ten tags will look like this:

```
<!ELEMENT departFrom_1        (#PCDATA)>
<!ELEMENT departTime_2        (#PCDATA)>
...
```

The `#PCDATA` keyword above means that the tag contains parsed character data; this means that the XML parser will find only character data, no tags or entity references (more about these in a minute). There are other keywords, such as `EMPTY`, which means the tag can't contain anything, and `ANY`, which means the tag can contain text, other tags, entity references, etc.

Our DTD - Version 1

Our completed DTD is shown below.

```
<!-- flights.dtd  -->
<!ELEMENT flights (itinerary)+>

<!ELEMENT itinerary (departFrom_1, departTime_1, arriveIn_1,
                   arriveTime_1, airline_1, departFrom_2,
                   departTime_2, arriveIn_2, arriveTime_2,
                   airline_2)>

<!ELEMENT departFrom_1     (#PCDATA)>
<!ELEMENT departTime_1     (#PCDATA)>
<!ELEMENT arriveIn_1       (#PCDATA)>
<!ELEMENT arriveTime_1     (#PCDATA)>
<!ELEMENT airline_1        (#PCDATA)>
<!ELEMENT departFrom_2     (#PCDATA)>
<!ELEMENT departTime_2     (#PCDATA)>
<!ELEMENT arriveIn_2       (#PCDATA)>
<!ELEMENT arriveTime_2     (#PCDATA)>
<!ELEMENT airline_2        (#PCDATA)>
```

Other things you can put in a DTD

There are a number of other things you can put in a DTD; the most common are attribute declarations and entity references.

Attribute declarations

Attribute declarations allow you to define the attributes that can appear inside a tag, as well as the kinds of data the attributes can contain.

As an example, let's say that we don't like the structure of the `<airline_1>` and `<airline_2>` tags. These tags typically contain data about the airline and the flight number. We've decided that we can reliably parse out the airline and the flight number; the airline number will be the text of the tag, and the flight number will be an attribute inside the tag itself. Here are the definitions for the new tags and their attributes:

```
<!ELEMENT airline_1        (#PCDATA)>
<!ATTLIST airline_1      flightNum CDATA #REQUIRED>
<!ELEMENT airline_2        (#PCDATA)>
<!ATTLIST airline_2      flightNum CDATA #REQUIRED>
```

The `#REQUIRED` keyword in the attribute definition means that this attribute must be coded for each and every `<airline_1>` and `<airline_2>` tag in your document. If an attribute isn't required, you can use the `#IMPLIED` keyword.

Another type of attribute definition allows you to specify a set of valid values, along with a default. As an example, let's say Xtreme Travel only deals with three airlines, and we've decided that the airline name should be an attribute of the `<airline_1>` and `<airline_2>` tags. Here are the definitions for the new tags:

```
<!ELEMENT airline_1        (EMPTY)>
<!ATTLIST airline_1      flightNum CDATA #REQUIRED
                         carrierName (Delta | American | United) "American">
<!ELEMENT airline_2        (EMPTY)>
<!ATTLIST airline_2      flightNum CDATA #REQUIRED
                         carrierName (Delta | American | United) "American">
```

In this somewhat ill-conceived example, the attribute `carrierName` can have the values `Delta`, `American`, or `United`. If no value is specified, the default is `American`. Also notice that because all of the data contained in these tags is now in the attributes, we used the `EMPTY` keyword to specify that these tags don't have any content.

Tags or attributes?

One common question in DTD writing is whether something should be a tag or an attribute. A third approach to our current example would be to create `<flightNumber>` and `<carrierName>` tags inside the

`<airline_1>` and `<airline_2>` tags:

```
<airline_1>
  <flightNumber>330</flightNumber>
  <carrierName>American</carrierName>
</airline_1>
```

In most cases, the tags versus attributes decision doesn't make any difference. However, if the data we're modelling needs to be reused, data in tags is easier to access. As an example, say the flight number returned by a database query needs to be used as input into another query. Finding and reusing a `<flightNumber>` tag is much easier than finding and reusing the `flightNumber` attribute of the `<airline_1>` tag.

Entity declarations

The last thing we'll add to our DTD is an entity declaration. Entities allow you to define symbols that are replaced by other text before they're displayed to the user. Here's an entity that defines the symbol `&xt;` as equivalent to the name "Xtreme Travel."

```
<!ENTITY xt "Xtreme Travel" >
```

Markup such as `Welcome to &xt;!` will replace the entity name with its value. If you use an entity for a common word or phrase, such as a product name, you can change all occurrences of that word or phrase simply by changing the entity declaration.

Our final DTD

Here's our final DTD:

```
<!-- flights.dtd  -->
<!ELEMENT flights (itinerary)+>

<!ELEMENT itinerary (departFrom_1, departTime_1, arriveIn_1,
                    arriveTime_1, airline_1, departFrom_2,
                    departTime_2, arriveIn_2, arriveTime_2,
                    airline_2)>

<!ELEMENT departFrom_1      (#PCDATA)>
<!ELEMENT departTime_1      (#PCDATA)>
<!ELEMENT arriveIn_1        (#PCDATA)>
<!ELEMENT arriveTime_1      (#PCDATA)>
<!ELEMENT airline_1         (EMPTY)>
<!ATTLIST airline_1         flightNum CDATA #REQUIRED
                         carrierName (Delta | American | United) "American">
<!ELEMENT departFrom_2      (#PCDATA)>
<!ELEMENT departTime_2      (#PCDATA)>
<!ELEMENT arriveIn_2        (#PCDATA)>
<!ELEMENT arriveTime_2      (#PCDATA)>
<!ELEMENT airline_2         (EMPTY)>
<!ATTLIST airline_2         flightNum CDATA #REQUIRED
                         carrierName (Delta | American | United) "American">

<!ENTITY xt "Xtreme Travel">
```

Summary

This article has covered the basics of creating a DTD. The most important part of this task is understanding the structure of our source data and the data relationships we want our XML tags to convey. As mentioned earlier, the XML tags we've created add semantic meaning and let us process our data in much more flexible ways. These benefits will be more apparent as we continue to develop our XML application.

What's next?

In our next article, we'll discuss the next step in building our application: Generating XML from our database. Other topics in the coming weeks include:

- Generating and manipulating a Document Object Model (DOM) tree
- Using XSL to transform XML documents
- XML and Reuse
- Dynamically generating DTDs based on database schemas

Please send any comments or questions to:

Doug Tidwell
dtidwell@us.ibm.com