IBM www.ibm.com/ XML

Education: Papers

Building an XML Application, Step 3: Converting XML into HTML with the Document Object Model (DOM)

Doug Tidwell
IBM XML Technical Strategy Group, TaskGuide Development
Updated January 1999

Step 1: Writing a DTD
Step 2: Generating XML from a Data Store

Abstract: In this paper, we'll take the XML document we generated in the previous paper and parse it into a DOM tree. Building a DOM tree is typically the first step in processing an XML document. Once the DOM tree is built, we'll look at the information in it and convert it into HTML. The DOM tree is a very useful data structure that allows us to manipulate the contents of the XML document. This paper focuses on building the tree, navigating through its contents, and generating HTML based on the nodes in the tree.

Technologies We'll be Using

We'll be using the methods of the Level 1 Specification of the Document Object Model. Documentation on these methods is provided with the IBM XML Parser for Java (XML4J parser). The DOM is a recommendation of the World Wide Web Consortium (W3C); for the complete DOM Level 1 specification, see http://www.w3.org/TR/REC-DOM-Level-1.

Sample XML Document

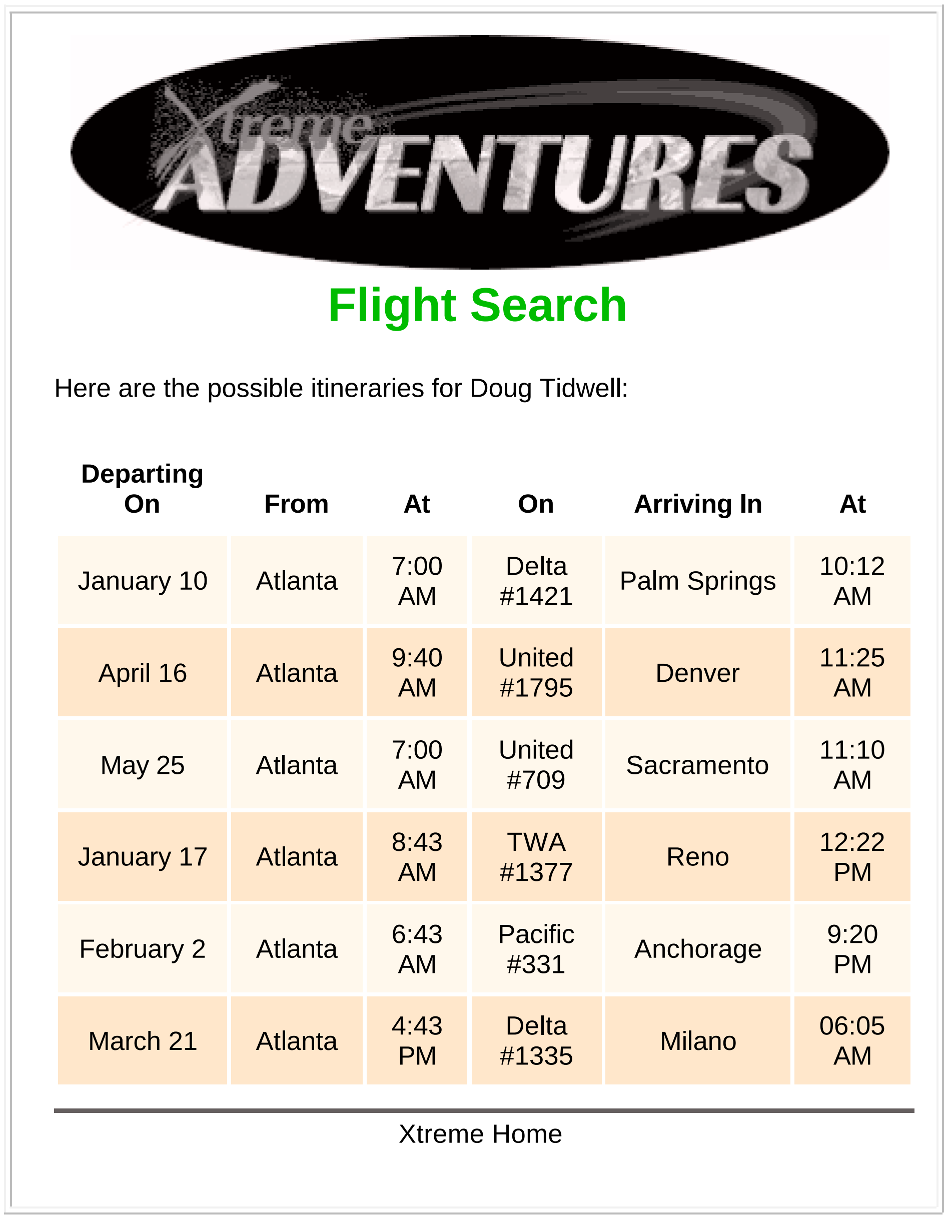
To refresh your memory, here's a look at a sample document produced by the servlet we built in our last paper:

```
<?xml version="1.0" ?>
<!DOCTYPE travelplans SYSTEM "flights.dtd">
  <travelplans>
    <itinerary>
      <outbound-depart-from>Chicago</outbound-depart-from>
      <outbound-depart-time year="1999" month="1" day="10" hour="6" minute="30" />
      <outbound-arrive-in>Palm Springs</outbound-arrive-in>
      <outbound-arrive-time year="1999" month="1" day="10" hour="11" minute="3" />
      <outbound-airline carrierName="American" flightNum="303" />
      <returning-depart-from>Palm Springs</returning-depart-from>
      <returning-depart-time year="1999" month="1" day="15" hour="11" minute="50" />
      <returning-arrive-in>Chicago</returning-arrive-in>
      <returning-arrive-time year="1999" month="1" day="15" hour="21" minute="24" />
      <returning-airline carrierName="American" flightNum="1250" />
    </itinerary>
    <itinerary>
      <outbound-depart-from>Atlanta</outbound-depart-from>
      <outbound-depart-time year="1999" month="1" day="10" hour="7" minute="0" />
      <outbound-arrive-in>Palm Springs</outbound-arrive-in>
      <outbound-arrive-time year="1999" month="1" day="10" hour="10" minute="12" />
      <outbound-airline carrierName="Delta" flightNum="1421" />
      <returning-depart-from>Palm Springs</returning-depart-from>
      <returning-depart-time year="1999" month="1" day="15" hour="16" minute="0" />
      <returning-arrive-in>Atlanta</returning-arrive-in>
      <returning-arrive-time year="1999" month="1" day="15" hour="22" minute="38" />
      <returning-airline carrierName="Delta" flightNum="5906" />
    </itinerary>
    . . .
  </travelplans>
```

Our code will build on our previous examples. This means that when we start to build a DOM tree, we'll have an XML document that exists as a Java `String` object; we'll build the DOM tree by parsing that `String`.

Our Goal

Our goal is to produce a Web page that looks like this:



Xtreme ADVENTURES

Flight Search

Here are the possible itineraries for Doug Tidwell:

Departing On	From	At	On	Arriving In	At
January 10	Atlanta	7:00 AM	Delta #1421	Palm Springs	10:12 AM
April 16	Atlanta	9:40 AM	United #1795	Denver	11:25 AM
May 25	Atlanta	7:00 AM	United #709	Sacramento	11:10 AM
January 17	Atlanta	8:43 AM	TWA #1377	Reno	12:22 PM
February 2	Atlanta	6:43 AM	Pacific #331	Anchorage	9:20 PM
March 21	Atlanta	4:43 PM	Delta #1335	Milano	06:05 AM

Xtreme Home

In the sample page above, the user's name (Doug Tidwell) and home city (Atlanta) are provided as input by an HTML form. Our code must customize the database query, the XML document, and the final HTML document accordingly.

Let's Get Started!

Now that we've reviewed our document format, let's start building the code that will create a DOM tree from the XML document. As with our previous examples, we'll discuss the main tasks our code has to accomplish, with a link to the complete code at the end of this paper.

Creating an XML Parser Object and Parsing our XML Document

Our first step is to create an XML Parser. We use a `com.ibm.xml.parser.Parser` object to do this:

```
Parser parser = new Parser("xslparse.err");
```

The argument to the constructor is the default input stream. Because we have an XML document (stored as a Java `String`), this filename will be the default log file for any errors that occur while we're parsing our XML document. To parse the XML document, we'll have to convert it to an `InputStream`:

```
ByteArrayInputStream bais = new ByteArrayInputStream(xmlString.getBytes());
```

Now that the XML document is in a form the parser can handle, we'll set some properties of the parser. The four lines below tell the parser to:

- Ignore any missing `<?xml ... ?>` processing instructions
- Ignore any missing `<!DOCTYPE ... >` declarations
- Ignore any comments that occur in the XML document
- Assume that no namespace markup is used

```
parser.setWarningNoXMLDecl(false);
parser.setWarningNoDoctypeDecl(false);
parser.setKeepComment(false);
parser.setProcessNamespace(false);
```

Now that our XML document is in the correct format, and our XML parser is set up just the way we want it, we'll parse the document and close the `InputStream`:

```
doc = parser.readStream(bais);
bais.close();
```

Pretty simple, huh? The `TXDocument` object `doc` should contain the DOM tree at this point.

DOM Tree Methods

Before we convert the data from our DOM tree into HTML, let's discuss the DOM methods we can use to navigate through the tree. Most methods are defined in the interface `org.w3c.dom.Node`, the primary datatype for the Document Object Model. The most commonly used methods are:

`getFirstChild()`
: Returns a `Node` object that is the first child of this `Node`.

`getNextSibling()`
: Returns a `Node` object that is the next sibling of this `Node`.

`getLastChild()`

Returns a Node object that is the last child of this Node.

getPreviousSibling()

Returns a Node object that is the previous sibling of this Node.

getAttribute(java.lang.String attrName)

Returns a String object representing the value of the requested attribute. If the attribute doesn't exist, the returned String is null.

Most often, we'll start with the root element of the DOM tree (getDocumentElement()), get its first child (getFirstChild()), then look at each of that child's siblings in turn (getFirstChild().getNextSibling()). To process the children of the root element in reverse order, use the getLastChild() and getPreviousSibling() methods.

Although our sample tag set doesn't use attributes, the getAttribute() method is commonly used to process the attributes of an XML element. Through the TXDocument class, the IBM XML parser provides other useful methods (such as getAttributeArray()) that make it easy to work with attributes.

Although these are the most commonly used DOM methods, be aware that there are many other methods available. For example, the DOM defines the getElementsByTagName() method that returns all child elements that have a particular name, regardless of how deeply nested in the tree they may be. As always, the documentation provided with the IBM parser has more information. The DOM Level 1 specification itself (available at http://www.w3.org/TR/REC-DOM-Level-1) also provides useful information about DOM methods.

Navigating the DOM Tree

The first thing we'll do is check our TXDocument object to make sure it isn't empty. Assuming it isn't, we'll look at the root element (the first element in the XML document, also known as the document element) and begin to convert our XML data into HTML. There are two methods for working with the root element: getRootName(), which returns the tag name of the root element, and getDocumentElement(), which returns a reference to the Node object itself.

Here's the code that makes sure our document isn't empty, then looks at the root element:

```
if (doc != null)
{
  String root = doc.getRootName();
  if (root != null)
  {
    if (root.equalsIgnoreCase("travelplans"))
    {
```

Next, we'll write a bunch of boilerplate HTML to set up the Web page. An alternate way of doing this would be to have our servlet simply convert our XML document into an HTML table; the table could then be embedded in a variety of Web pages. To keep it brief, we'll skip the lines of code that generate the HTML.

Once we've written the top part of the HTML page, we have to generate the table that contains the actual data we care about. This means we'll need to look at each <itinerary> element inside the <travelplans> element.

For aesthetic reasons, we've decided to alternate the color of each row in the HTML table. This means we'll need a flag to determine which color each row should be.

```
boolean papayaRow = true;
```

We'll also use one of Java's built-in classes to retrieve the names of the months for the current location.

The values we'll get from the XML document will be of the form month="0". Month 0 is January (or whatever localized string refers to the first month of the year). We'll also use a Java DateFormat object to convert the hour and minute attributes into a localized time. Here's how we create these objects:

```
DateFormatSymbols dfs = new DateFormatSymbols();
String[] monthNames = dfs.getMonths();

DateFormat df2 = DateFormat.getTimeInstance(DateFormat.SHORT);
```

Now we'll set up our for loop. As we mentioned earlier, we'll use the getFirstChild() and getNextSibling() methods to walk through the DOM tree. Because we're going to process all the child elements of the <itinerary> tag before we write out the row of the HTML table, we'll define some String variables to hold the values for the table.

```
for (Node nextKid = doc.getDocumentElement().getFirstChild();
     nextKid != null;
     nextKid = nextKid.getNextSibling())
{
  String deptOn = "", deptFrom = "", deptAt = "",
         deptAir = "", arrvIn = "", arrvAt = "";
```

The first thing we have to do is make sure the current child of the <travelplans> tag is an <itinerary> element. Assuming that's the case, we'll write out the HTML table row tag, using the appropriate color.

```
if (nextKid.getNodeName().equalsIgnoreCase("itinerary"))
{
  if (papayaRow)
  {
    out.print("<tr bgcolor=\"papayawhip\">\n");
    papayaRow = false;
  }
  else
  {
    out.print("<tr bgcolor=\"ffcc99\">\n");
    papayaRow = true;
  }
```

At this point, we're inside an <itinerary> element, and we're ready to start processing its children. We'll use a for loop similar to the one above. An important difference here is that we need to look at the first child element of the <itinerary> element to see if it is a TXText or TXElement element. Without this test, we'll get an exception when we try to cast the TXText element to a TXElement. Consider this example:

```
<itinerary>
  Here's another trip:
  <outbound-depart-time year="1999" month="1" day="10" hour="7" minute="0" />
  ...
```

The first child of the <itinerary> element is a TXText element that contains the text of the <itinerary> tag. Getting back to our code, we'll check the type of the child element and get ready to process it:

```
for (Node nG = (Node) nextKid.getFirstChild();
     nG != null;
     nG = (Node) nG.getNextSibling())
{
  if (nG instanceof TXElement)
  {
    TXElement nextGrandkid = (TXElement) nG;
```

Now we're simply going to look at the tag name of the child element. If it's an element we care about, we'll process its data. If it's not an element we want, we just ignore it. Here is the processing we do for all of the different tags:

```
// We use the Java Calendar and DateFormat classes to handle the
// date and time information.
if (nextGrandkid.getTagName().equals("outbound-depart-time"))
{
  deptOn = monthNames[Integer.parseInt(nextGrandkid.getAttribute("month"))] +
           " " + nextGrandkid.getAttribute("day");
  cal.set(Calendar.HOUR,
          Integer.parseInt(nextGrandkid.getAttribute("hour")));
  cal.set(Calendar.MINUTE,
          Integer.parseInt(nextGrandkid.getAttribute("minute")));
  deptAt = df2.format(cal.getTime());
}
else if (nextGrandkid.getTagName().equals("outbound-depart-from"))
  deptFrom = nextGrandkid.getText();
else if (nextGrandkid.getTagName().equals("outbound-airline"))
  deptAir = nextGrandkid.getAttribute("carrierName") + " " +
            nextGrandkid.getAttribute("flightNum");
else if (nextGrandkid.getTagName().equals("outbound-arrive-in"))
  arrvIn = nextGrandkid.getText();

// Again, we use the Calendar and DateFormat classes to format the time.
else if (nextGrandkid.getTagName().equals("outbound-arrive-time"))
{
  cal.set(Calendar.HOUR,
          Integer.parseInt(nextGrandkid.getAttribute("hour")));
  cal.set(Calendar.MINUTE,
          Integer.parseInt(nextGrandkid.getAttribute("minute")));
  arrvAt = df2.format(cal.getTime());
}
```

Because of the design of the underlying database, there are some elements that we have to process differently: <outbound-depart-time>, <outbound-arrive-time>, and <outbound-airline>. We have to look at the attributes of these tags to find the information we need. Notice that because we parsed the original, unstructured data, all the pieces of information we need (airline, month, etc.) are easy to access. Once we've processed all of the children of the <itinerary> tag, we're ready to write out the rest of the row of the HTML table:

```
out.print("<td align=\"center\">" + deptOn + "</td>\n");
out.print("<td align=\"center\">" + deptFrom + "</td>\n");
out.print("<td align=\"center\">" + deptAt + "</td>\n");
out.print("<td align=\"center\">" + deptAir + "</td>\n");
out.print("<td align=\"center\">" + arrvIn + "</td>\n");
out.print("<td align=\"center\">" + arrvAt + "</td>\n");
out.print("</tr>\n");
```

Once we've gone through all the elements in the DOM tree, we simply close the HTML table, write out any boilerplate markup that goes at the bottom of the HTML page, and we're done.

Sample Code

To study this code, see the html version of this file on the XML web site.

flights.dtd

The DTD for our sample data

parseXML.java

Java source file for the servlet that parses the XML-tagged data and generates an HTML from it.

mainxtremehead.gif

redbar2.gif

The graphics files embedded in the Web page produced by our servlet.

Summary

In this paper, we learned how to parse an XML document, create a DOM tree, then navigate through the DOM tree. We also learned how to convert the XML markup into HTML by looking for certain elements and attributes within the DOM tree.

What's Next?

Our next paper is using the Extensible Style Language (XSL) as an alternate way to transform XML documents. XSL is an XML vocabulary that lets you define rules for converting XML documents into some other markup. Although XSL is an emerging standard that is not yet defined, the next paper will give you a sense of the design goals of XSL and the types of applications it will enable.

Step 1: Writing a DTD
Step 2: Generating XML from a Data Store

Please send any comments or questions to:
Doug Tidwell
dtidwell@us.ibm.com