XML Data Binding with Castor

By Scott L. Bain
Senior Consultant
Net Objectives
(www.netobjectives.com)

XML is popping up everywhere, permeating technologies at just about every layer of software development. Whether you are building applications, utilities, tools, or frameworks, you're probably using XML, considering XML, or anticipating the need to integrate with other packages that do.

From the standpoint of the developer, there are three issues to be dealt with: how do we read XML, how do we process it, and how do we write it. Put another way, we need to find an easy and flexible way to make good use of XML without significantly increasing development time or over-complicating our projects.

Enter data binding.

As good Java developers we would naturally prefer to deal with our data in an object model, instantiating data beans to represent each entity and populating their state. And, ideally, we'd like the model to be structured in such a way as to make our use of it the simplest and most straightforward task possible. The process of getting the state information "into" our data model before we make use of it, and "back out" to XML when we are done (if we've made any changes, for instance), is called "binding".

At first blush this would seem to be a parsing task, and one certainly could tackle it that way. The purpose of this article, however, is to point out a much better alternative for some applications.

Why not SAX?

If you're familiar with SAX (the **S**imple **A**PI for **X**ML), it's easy to see how you could code an implementation of the ContentHandler interface, which could then instantiate objects in an object model, populate their state, and link them together in a hierarchy. Consider the following XML structure:

```
<customer ID = "fbs0001">
      <firstName>Fred</firstName>
      <MI>B.</MI>
      <lastName>Scerbo</lastName>
</customer>
```

The object model here would be a simple one: the Customer class has one attribute, a string to hold an ID, and three member objects, each a reference to an instance of one of the other three classes FirstName, MI, and LastName. Add the proper accessor methods

(getters and setters) to make them proper javabeans and our classes would look something like this:

```
public class Customer {
	private String myID;
	private FirstName myFirstName;
	private MI myMI;
	private LastName myLastName;

	public void setID (String anID) {myID = anID;}
	public String getID(){return myID;}
	public void setFirstName(FirstName aFirstName){
		myFirstName = aFirstName;
	}
	public FirstName getFirstName(){return myFirstName;}
	public void setMI(MI aMI){
		myMI = aMI;
	}
	public MI getMI(){return myMI;}
	public void setLastName(LastName aLastName){
		myLastName = aLastName;
	}
	public LastName getLastName(){return myLastName;}
}

public class FirstName {
	private String myText;
	public void setText(String text){myText = text;}
	public String getText(){return myText;}
}

public class MI {
	private String myText;
	public void setText(String text){myText = text;}
	public String getText(){return myText;}
}

public class LastName {
	private String myText;
	public void setText(String text){myText = text;}
	public String getText(){return myText;}
}
```

SAX uses a callback strategy to parse XML. You instantiate a copy of the parser (let's say, the Xerces parser from Apache), hand it the XML source to parse, and give it a reference to a class of your own design that implements the ContentHandler interface. Then the parser "calls back" to that ContentHandler implementation as it parses the XML.

This interface includes methods for each sort of thing the parser will run into as it parses the XML – startElement(), endElement(), characters(), and so on. By designing these methods to instantiate the proper objects in our model, we can successfully bind the XML to the object model, at least on the "in" side of the problem.

However, this is rather ugly.

First of all, SAX does not keep track of context. There are separate methods called for the start of an element and the characters the element contains. Thus, it's up to your implementing class to keep track of which databean is currently "under construction" so that the right state goes into the right bean.

For example, in the rather simple XML above we'll get a call to startElement() when the opening tag of <customer> is reached. Naturally, we'll want to instantiate a Customer object. This same call will hand us the ID attribute, so we'll have what we need to call setID on the new Customer instance. However, the next thing that will happen is another call to startElement(), this time responding to the opening tag for the <firstName> element. The fact that we've got a partially-built <customer> to add this new object to has to be dealt with in the method, and we've not even discussed how we're going to respond to the characters() method call when the first name "Fred" is processed.

This results in fairly convoluted logic that will be difficult to maintain when the structure of your data is changed. It creates a lot of coupling between your processing classes and the data itself, which we know we want to avoid.

Also, SAX is meant to be a generic API, comprehensively processing all the possible content a XML document might hold. Therefore there are methods, such as those that deal with name spaces and DTD references, that are necessary parts of the interface but which we likely won't care about. Nevertheless, all these methods must be present in our code, making it needlessly verbose.

Finally, SAX is a reader, not a writer. If it's possible that we'll need to re-serialize our object model back into XML, we'll either have to write our own code to do that, or use another technology entirely, like DOM or JDOM.

Okay then, what about DOM (or JDOM, or DOM4J, or…)

DOM (the Document Object Model), and the other dom-like parsers, definitely offer an advantage to simple SAX parsing. DOM represents another layer of functionality over SAX, which should not surprise us since most DOM parsers use a SAX parser internally, as the reader (Apache's Xerces-DOM uses Xerces-SAX).

DOM is already, as its name implies, an object model, so it's a more convenient form for us to deal with. A DOM parser reads the complete XML document, constructs a hierarchical tree, and hands the client class a reference to the rootmost, or "document node" object. Calling methods on this object gets us references to its attributes, data, and "child nodes", and then calling methods on those children gets us references to their attributes, data, and child nodes, and so forth. Recursive coding techniques work well here, and the tree itself models the context of each bit of data.

Add to this the fact that the DOM tree can be modified, and then re-serialized back into XML, and this would seem to be an ideal solution. Indeed, DOM is a powerful technology; every web browser in the world uses it to "make use of" HTML for display purposes.

However, while DOM is "an" object model, it's not *our* object model. The objects in DOM are named for the type of XML thingy they represent – Document, Element, Attribute, and so forth. So what we're going to have to do is to walk the DOM tree and create a second structure of objects, our Customer, FirstName, MI, and the other classes we've defined in our model.

This represents an unfortunate performance hit, since we're basically building one object model in order to build another. It would be better to move the XML directly from the serial form (the form we think of when we think of XML documents) into a representative object model, and then back again when we're done.

That's what data binding does for us.

Meet Castor

Castor (www.castor.org) is a free, open source tool that binds object models to, among other things[1], XML. Once you import the Castor classes, it takes remarkably few lines of code to move your data from an XML source into an object model and back again, freeing you to concentrate on business logic.

Two terms we need here are *marshalling*, and *un-marshalling*.

Marshalling refers to the process that takes an object or a tree of objects, and creates the XML representation that will record its state. It's a kind of serialization.

Un-marshalling is just the opposite, it takes the XML representation and builds an object model from it.

Castor is a pretty smart set of tools. Using its "default behaviour", one can let Castor create an XML structure to represent an existing set of classes – or one can take an existing XML structure (assuming you have a Schema for it), and hand it to the automatic class generation tool that comes with Castor, which will build the object model for you.

If you already have an XML structure in mind, and you already have a set of objects you need to populate with it, then you can use the more sophisticated "mapping" functionality that Castor offers, which gives you very complete control over what goes where.

We'll try it both ways.

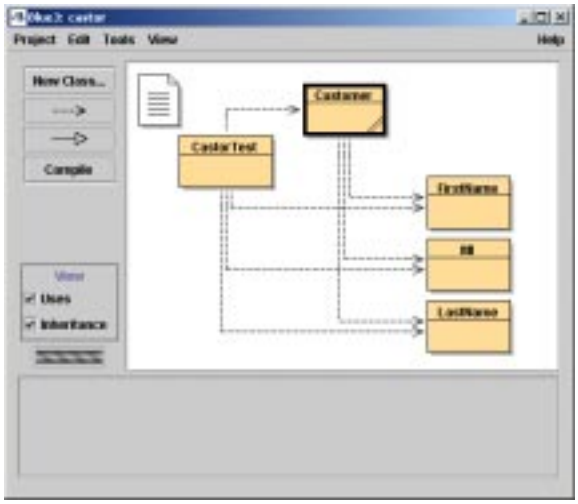
[1] Don't miss the significance of this statement. The "other things" include LDAP and RDBMS's, so this one tool can be used to move your object model into other worlds besides XML.

First, let's create an object model, and then let Castor create an XML representation for us. We'll use the classes defined in the discussion on SAX, and then write a simple test routine.

```
import org.exolab.castor.xml.*;
import java.io.*;
public class CastorTest{
      public static void main(String[] args){
            // Lets make our object model. First the pieces…
            Customer myCust = new Customer();
            FirstName myFN = new FirstName();
            myFN.setText("Fred");
            MI myMI = new MI();
            myMI.setText("B");
            LastName myLN = new LastName();
            myLN.setText("Scerbo");

            // Now, structure them
            myCust.setID("fbs0001");
            myCust.setFirstName(myFN);
            myCust.setMI(myMI);
            myCust.setLastName(myLN);

            // Now, "marshal" them into an XML structure of
            // Castor's choosing

            try {
                  // A stream to write the XML into
                  FileWriter out = new FileWriter("customer.xml");
                  // One method call!  That's it!
                  Marshaller.marshal(myCust, out);
                  out.close();
            } catch(Exception e) {
                  System.out.println("Exception: "+e.getMessage());
            }
      }
}
```

Here we're letting Castor create whatever XML structure it wants, using the static marshal() method of the Marshaller class. Here's what it will create:

```
<?xml version="1.0"?>
<customer>
    <firstName>
        <text>Fred</text>
    </firstName>
    <ID>fbs0001</ID>
    <lastName>
        <text>Scerbo</text>
    </lastName>
    <MI>
        <text>B</text>
    </MI>
</customer>
```



A couple of side notes here. The tool I used to create all my classes and test them is a free Java IDE called BlueJ. It's available from www.bluej.org, which is maintained by Michael Kölling at Monash University in Australia. To the left is what our project looks like in BlueJ.

(Note that it is UML-based, not focused on the GUI widgets. I like BlueJ a bunch.)

Also, note that Castor does not indent your XML unless you ask it to. Indenting large XML documents can drastically increase their size. It's obviously not a problem here, and so I've changed the org.exolab.castor.indent property in the castor.properties file inside the distribution .JAR to "true". Similarly, I've changed the org.exolab.castor.xml.naming property to "mixed", since I prefer the standard java case conventions when naming fields. Otherwise my elements would be named <first-name> and so forth, instead of the "camel case" we typically use in Java: firstName. Might as well be consistent.

Castor uses Java reflection to determine the public method signature, and creates the element and attribute names accordingly. Where Castor finds primitives (int, long, float), it will make attributes, and where it finds objects it will make elements. In this case, since the String in Java is an object, it made ID an element, which was not what we had in mind.

In fact, this is not really the XML grammar we were shooting for at all. Using Castor's default behaviour, we got its best guess at a decent XML structure for our data model. If we didn't really have a model in mind, this one would probably be fine – and that's definitely one way to use data binding. Create the most convenient object model for your data, let Castor create an XML structure to represent it, and use that structure from then on.

Let's complete this process by demonstrating how we'd un-Marshal this XML file back into our object model:

```
import org.exolab.castor.xml.*;
import java.io.*;

public class CastorTest2 {
  public static void main(String[] args) {
    Customer myCust = new Customer();
    try {
      FileReader in = new FileReader("customer.xml");
      myCust = (Customer)Unmarshaller.unmarshal(Customer.class, in);
    } catch (Exception e) {
      System.out.println("Exception: "+e.getMessage());
    }

    System.out.println("    Customer ID: " + myCust.getID());
    System.out.println("     First Name: " +
      myCust.getFirstName().getText());
    System.out.println(" Middle Initial: " + myCust.getMI().getText());
    System.out.println("      Last Name: " +
      myCust.getLastName().getText());
      }
}
```

Here again, the Castor functionality is accomplished by a single call to a static method, passing in the class of the senior-most object in the object model (in this case, Customer), and a source for the XML. Castor does the rest, though the return type is Object, so you must explicitly cast the reference back into Customer before you can call the class-specific methods. Here's the output:

```
    Customer ID: fbs0001
     First Name: Fred
 Middle Initial: B
      Last Name: Scerbo
```

Pretty much what you'd expect from this simple example.

The alternate way of using Castor's default mode is to start with an XML grammar and allow Castor to create the object model. We won't go into the details here, but the trick is that you must create an XML-Schema for the structure you want, and then hand it to the static main() method of org.exolab.castor.builder.SourceGenerator, which will create a set of objects to bind to any XML instance document that conforms to the Schema involved. The process is really pretty simple, and the details are readily available at http://www.castor.org/sourcegen.html. You need to know XML-Schema to use this tool, which is outside the scope of this article.

So, as you can see, if you are in a position to allow Castor to control either the object model, or the XML structure, then often this simple "default mode" is all you need to accomplish the marshaling and un-marshaling of your data.

However, there are times when both the object model and the XML grammar are already set, and we need to create a discreet mapping from one to the other. That's where Castor's mapping capability comes in.

Mapping with Castor

Castor comes with a set of tools to allow you to specifically map certain XML elements and attributes to a known object model. In short, what you do is create another XML document, one that conforms to the Castor mapping DTD (for details see (http://www.castor.org/xml-mapping.html), and then use this to control the marshaling and un-marshaling process.

Remember that Castor created the following XML from our object model when we used its default mode:

```
<customer>
    <firstName>
        <text>Fred</text>
    </firstName>
    <ID>fbs0001</ID>
    <lastName>
        <text>Scerbo</text>
    </lastName>
    <MI>
        <text>B</text>
    </MI>
</customer>
```

While we were looking for this structure:

```
<customer ID = ”fbs0001”>
      <firstName>Fred</firstName>
      <MI>B.</MI>
      <lastName>Scerbo</lastName>
</customer>
```

There are lots of differences. Castor made ID a sub-element, rather than an attribute, because a String in an object in Java, and Castor only makes primitives into attributes in its default mode. Also, it created <text> elements, rather than simply making <firstName> and the like hold text nodes per se.

By creating a mapping document, we can force Castor to marshal and un-marshal from the XML structure we want. Here’s an example of such a map:

```
<mapping>
   <class name = "Customer">
      <field name = "ID" type = "string">
         <bind-xml name="ID" node="attribute"/>
      </field>
      <field name = "FirstName" type = "FirstName"/>
      <field name = "MI" type = "MI"/>
      <field name = "LastName" type = "LastName"/>
   </class>
   <class name = "FirstName">
      <field name = "text">
         <bind-xml name = "text" node = "text"/>
```

```
      </field>
   </class>
   <class name = "MI">
      <field name = "text">
         <bind-xml name = "text" node = "text"/>
      </field>
   </class>
   <class name = "LastName">
      <field name = "text">
         <bind-xml name = "text" node = "text"/>
      </field>
   </class>
</mapping>
```

Each of our classes is defined here with a <class> element. The name attribute tells castor what .class file to look for when instantiating the object in question, and also what the name of the element will be in the XML structure. Each class contains some number of <field> elements, which allow us to specify the type of each field, what it contains, and (optionally) the getter and setter (accessor) method names. If we don't specify, Castor will assume we've named them using the Javabean specification and will look for the methods accordingly.

The code that makes use of this mapping document is a little more involved than we wrote before, using Castor's default mode, but not by much.

First, we can no longer use the static methods of the marshaller and un-marshaller classes, but must create an instance that's specifically aware of the senior-most class of our object model:

```
Unmarshaller myUnmarshaller = new Unmarshaller(Customer.class)
```

Next, we create a Mapping object, and point it at the mapping document we've created above:

```
Mapping myMapping = new Mapping();
myMapping.loadMapping("customerMapping.xml");
```

Finally, we assign the mapping to the marshaller or un-marshaller:

```
myUnmarshaller.setMapping(myMapping);
```

Now we open the file and pass the stream in to the unmarshal() method of our instance of the Unmarshaller, only this time we don't have to reference the class object of Customer. Our Unmarshaller instance already has this information from its construction.

```
try {
  FileReader in = new FileReader("customer.xml");
  myCust = (Customer)myUnmarshaller.unmarshal(in);
} catch (Exception e) {
  System.out.println("Exception: "+e.getMessage());
}
```

The rest of our code is unchanged. Here's the complete class (note that we need to import the org.exolab.castor.mapping package to use the Mapping class):

```
import org.exolab.castor.xml.*;
import org.exolab.castor.mapping.*;
import java.io.*;

public class CastorTest3 {
  public static void main(String[] args) {
    Customer myCust = new Customer();
    try {
      // We're making an instance specifically for Customer.class
      Unmarshaller myUnmarshaller = new Unmarshaller(Customer.class);
      FileReader in = new FileReader("customer.xml");
      Mapping myMapping = new Mapping();
      // The myMapping object will parse and load the mapping here
      myMapping.loadMapping("customerMapping.xml");
      // Now we tell the Unmarshaller to use this mapping
      myUnmarshaller.setMapping(myMapping);
      // As before, we have to cast the result of the unMarshalling
      myCust = (Customer)myUnmarshaller.unmarshal(in);
      } catch (Exception e) {
         System.out.println("Exception: "+e.getMessage());
      }

   System.out.println("    Customer ID: " + myCust.getID());
   System.out.println("     First Name: " +
      myCust.getFirstName().getText());
   System.out.println(" Middle Initial: " + myCust.getMI().getText());
   System.out.println("      Last Name: " +
      myCust.getLastName().getText());
   }
}
```

Castor allows us to create flexible binding between XML structures and java object models, in a way that is easy to maintain and quite straightforward. Once this binding is in place, we can concentrate on the business logic, user interfaces, and all the other issues in our project domain without having to concern ourselves with the details of the XML we're interfacing with.

Castor is an open source tool. You can download the entire package from http://www.castor.org, including extensive documentation, and/or your can join the CVS tree there and participate in its development. There are also developer and announcement mailing lists at the site, which I recommend you subscribe to, at least while you're getting comfortable with Castor.

Get Castor! It's free, its fast, and it works. For most of my applications, I have abandoned traditional parsing techniques in favor of data binding, and my code is fundamentally cleaner as a result.