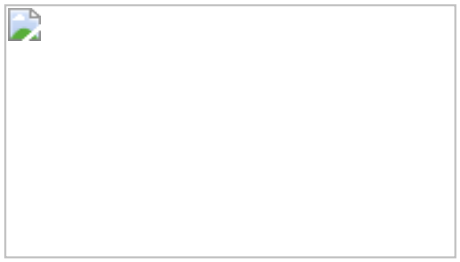
The World's Leading Java Resource

Using Apache Tuscany SDO and JSF To Build Dynamic Web Forms

This was the challenge: Build a generic system that lets users compare data suppliers in different categories. The data to be compared is defined by XML Schemas, where new schemas will be frequently uploaded and existing schemas may be changed. Moreover, the schemas aren't specifically designed for this system, so system specific metadata can't be added as attributes.



Based on the schemas the data suppliers must be presented with standard HTML forms to enter in their own specific data. Alternatively the suppliers must be able to use a Web Service interface.

Overall the sum of the requirements demanded extreme flexibility and robustness of the system.

So basically we wanted something that could go from XSD/XML to HTML forms and back. We stumbled upon a few COTS products, but these weren't suitable. I know many of you will probably think XForms. XForms was what turned up most when searching for known ways to do such a thing. Not knowing XForms beforehand I just superficially skimmed the technology and concluded that it wasn't right for us for several different reasons - one of them being the flexibility the requirements demanded. This may be because we didn't examine it thoroughly enough.

Anyway, looking further we came upon EMF (Eclipse Modelling Framework) and started building a small prototype. EMF seemed rather complex and during a discussion in a newsgroup somebody recommended SDO and the Apache Tuscany SDO implementation, which is based on EMF.

It immediately caught our attention since it seemed to supply just what we needed through a much simpler API. Any issues we had were quickly resolved by the extremely friendly people behind Apache Tuscany via the user mailing list. And our proposals for new functionality were quickly implemented.

Now, let's get into some technical details. This article doesn't introduce SDO as such - for that I recommend the articles by Kelvin Goodson and Geoffrey Winn in earlier issues of *JDJ* (Volume 11, Issue 12).

SDO

With Apache Tuscany SDO you operate within certain helper contexts. You do your SDO work in a context and you can choose to use this one context for all your work and let it live throughout your application lifetime, or you can use as many contexts as you like. This can be useful when XML Schemas aren't static - even in the same namespace. Once an XML Schema has been defined the types are cached for easy access in your context. If an XML Schema changes (and still keeps the same namespace) you can just scratch the original context and create a new one. Of course it's never recommended to work with XML Schemas that way since it opens up millions of issues regarding schema instances (XML) and the version of the schema. It's recommended that you do your versioning using the namespace. Still it shows some of the flexibility of the Apache Tuscany SDO implementation.

Let's look at some examples of how to define an XML Schema as SDOs and how to load and save XML from them (see Listing 1). For simplicity's sake the examples are without enumerations or many-valued properties. All use Apache Tuscany SDO 1.0 Incubating (beta 1, but the final has just been released).

Okay, so following listing 1, we have got a hold of the SDO data object representing the type of the schema we're interested in. Before we continue let's just see how you can get to this point if you already have existing XML that you want loaded into the data object.

```
String xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
   + "<test:TestElementX xmlns:test=\"http://mytestnamespace\">"
   + " <test:SubElement1>1.0</test:SubElement1>"
   + " <test:SubElement2>Hello</test:SubElement2>"
   + "</test:TestElementX>";
XMLDocument xmlDoc = context.getXMLHelper().load(xml);
rootDataObject = xmlDoc.getRootObject();
```

At this point you can choose to extract XML from the data object. This will be more relevant once we've altered the data object, but this is how it can be done:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
xmlDoc.setEncoding("UTF-8");
try {
   context.getXMLHelper().save(xmlDoc, baos, null);
} catch (IOException e) {
   e.printStackTrace();
   System.exit(-1);
}
System.out.println(baos.toString());
```

If you haven't already got hold of the XMLDocument, it can be created like this:

```
XMLDocument xmlDoc = context.getXMLHelper().createDocument(dataObject, namespace,
"TestElementX");
```

Before we continue to work with the root data object, let's take a brief look at how we can make any of this useful in a browser.

JSF

JSF was chosen as the Web framework since it has an extensive component model, where each HTML element needed is represented by a Java object. It comes with built-in conversion and validation and suits the project well.

Per XML element we want displayed we need a UIOutput object for the label and an UIInput object for the data.

Data Structure

We needed a data structure that would wrap the SDO objects along with the needed JSF objects and found the Composite design pattern useful (http://en.wikipedia.org/wiki/Composite_pattern). Using this pattern we could store our own objects in a structure that matches the SDO data graph. SDO containment properties (complex types) went into the composites and SDO non-containment properties (simple types) went into the leafs as illustrated in Figure 1.

Filling the Data Structure from XSD and XML

From the root data object obtained earlier we're going to construct the JSF components and fill the data structure.

Each property is extracted from the data object and handled recursively. The JSF components are generated, as we recursively traverse the structure, ending in the leafs (see Listing 2).

Now the data structure is filled with the SDO and JSF components - ready to be plugged into a GUI such as added to the child list of a panel:

```
private void setupXmlElement(XmlElement xmlElement) {

  CompositeXmlElement comp = xmlElement.getCompositeXmlElement();
  if (comp != null) {
    for (XmlElement child : comp.getChildren()) {
      setupXmlElement(child);
    }
  } else {

    LeafXmlElement leaf = (LeafXmlElement) xmlElement;
    if (leaf.getUiInput() != null && leaf.getUiInput().isRendered()) {

      this.getPanel().getChildren().add(leaf.getLabel());
      this.getPanel().getChildren().add(leaf.getUiInput());
    }
  }
}
```

This very simple example renders to this page (see Figure 2).

Because of the flexibility of JSF validation messages, such things as styles can be easily be added to each relevant component.

Submitting Data Back into the SDO Data Model
Submitting the form is easy. You just call submit on the XmlElement and the composite pattern handles the rest by setting each value from the UIInput in the leaf's submit method:

```
Object inputValue = this.uiInput.getValue();
String stringValue = inputValue.toString();
Object value = SDOUtil.createFromString(this.property.getType(), stringValue);
this.dataObject.set(property, value);
```

Conclusion
These were some of the basic operations that made it possible for us to assemble a very generic system. Apache Tuscany SDO turned out to be very flexible. Through its simple APIs it was easy to work with schemas and data and easy to plug this into a presentation framework. The final application contained a lot more than exemplified here, since most schema constructs were supported, but that's beyond the scope of this article. Supplying the application with a Web Service interface was also easy since the SDOs could be used to both import and generate XML directly as shown above.

Apache Tuscany also covers SCA and DAS but that's a whole different article.