Data Binding Architecture

Introduction

In CXF, data binding components are responsible for mapping between XML 'on the wire' and Java objects. Each data binding implements a particular discipline for mapping, such as JAXB or XML Beans.

There are three parts to a data binding:

- Mapping the live data as it comes into and out of services.
- Providing XML schema based on Java objects for dynamic ?wsdl URLs and java2ws.
- Generating Java code from WSDL for wsdl2java (and, theoretically, dynamic clients).

All data bindings provide the live data mapping. The other two facilities are optional.

Data Mapping

Data bindings provide data mapping by implementing the org.apache.cxf.databinding.DataBinding interface. The AbstractDataBinding class in the same package provides some common functionality.

To add a data binding, you will need to create a class that implements DataBinding, perhaps by extending AbstractDataBinding.

Formats

Each data binding supports one or more formats for the data in transit. A 'format' is a Java representation of XML. CXF works, primarily, with STaX. Thus, all data bindings must support XMLStreamReader and XMLStreamWriter as formats. Some interceptors expect to be able to read or write DOM Nodes. Thus, new data bindings should also support this format. Data bindings advertise their supported data formats via their implementation of two functions from DataBinding, e.g.:

```
public Class<?>[] getSupportedReaderFormats() {
     return new Class[] {XMLStreamReader.class, Node.class};
 }

 public Class<?>[] getSupportedWriterFormats() {
     return new Class[] {XMLStreamWriter.class, Node.class};
 }
```

Readers and Writers

All the work of mapping is done by objects that implement DataReader<Format> and DataWriter<Format>, where 'Format' is a representation class as defined above. CXF code obtains readers and writers from data binding objects via

```
<T> DataReader<T> createReader(Class<T> cls);
<T> DataWriter<T> createWriter(Class<T> cls);
```

Such a call might look like:

```
DataReader<XMLStreamReader> reader = binding.createReader(XMLStreamReader.class);
```

Please read the javadoc for these interfaces, as it has a reasonably complete explanation of their contracts.

Getting Set Up

If you've read the reader and writer interfaces, you've seen that these objects are required to take XML and produce Java object, and vica versa. In general, data bindings need to obtain information about the service in order to implement their mapping. Data bindings obtain this information from the CXF service model in their *initialize* methods. The service model describes the operations, messages, and parts of the service.

The initialize method is often the most complex part of a data binding, as it must work with the WSDL model of a service, perhaps in conjunction with Java reflection on the service implementation class, to determine the mappings from parts to objects. The 'front end' sets up the service model based on WSDL contents or annotations or other information, but it is left to the data binding to work out the gory details. It is very likely that some refactoring could move code from the existing data bindings to common code; anyone willing to do the necessary heavy thinking is welcome to join in.

At the end of the initialization process, the data binding object should know what Java objects it is prepared to operate on and what XML types or elements it is prepared to work with.

XML Schema

Internally, CXF uses XML Schema as it's XML type vocabulary. The front ends annotate the service model with XML schema information when it is available to them. Data bindings provide XML schema.

There are no APIs in the DataBinding interface for schema. Instead, data bindings add schema information to the service model. As of CXF 2.1, they add it twice: once as objects in the XmlSchema object model, and once as a DOM document. This is an artifact of workarounds to problems with XmlSchema, and we hope to remove it some day.

org.apache.cxf.xmlbeans.XmlBeansDataBinding and its related classes provide a relatively clean and simple example of this process.

Tooling and Code Generation

<To Be Written>