Defining Contract first webservices with wsdl generation from java

In this post I will describe how to do contract first in a very simple and efficient way. All people involved in webservice design have the problem what tool to use to describe the webservice interface. Of course there are wsdl editors but you can make many errors in using them and they are normally not very intuitive.
The original post is from Christian Schneider´s blog and can be found here

What we would like to have

Ideally I would imagine to have a domain specific language that makes designing the webservice very easy and fail safe. Then there should be an IDE that supports me in writing the DSL. Either with content assist or with a graphical editor. In any case the IDE should check the DSL while it is entered and help me to do the right thing. Some people favor using Model Driven Design tools to do this job but it is quite hard finding good tools and configuring them for the job.

Which tools did I choose and why

I wanted to go some way in the direction of a DSL for webservices but without using a big MDD tool. So I thought about how to have most advantages of a DSL with just Java and the normal Eclipse IDE. So I needed Java code that looks almost like a DSL and a tool to generate the webservice out of it. For the WSDL generation I used Apache CXF with JAXWS and JAXB annotations to describe the webservice. While this setup is quite standard I focused on keeping the syntax as simple as possible.

So how does the DSL look like

To describe a data object I use a java class with just attributes. The Namespace will come from the package name. To make JAXB understand this syntax only one annotation is necessary. Of course some more annotations are necessary if you want to use special features.

Customer datatype

```
package com.example.customerservice;

@XmlAccessorType( XmlAccessType.FIELD )
public class Customer {
    String name;
    String[] address;
    int numOrders;
    double revenue;
    BigDecimal test;
    Date birthDate;
    CustomerType type;
}
```

The sample class Customer gives a nice overview which primitive datatypes can be used. Additionally you can create arrays of primitive or class datatypes in this way. The complete example also contains an enumeration definition to show this is possible. I think this code is quite near the DSL I would imagine to describe my services.

Enumeration CustomerType

Shows how enumerations are handled:

```
package com.example.customerservice;

public enum CustomerType {
    PRIVATE, BUSINESS
}
```

NoSuchCustomerException

Defining Exceptions is a little tricky as the default behaviour is to create Exception_Exception classes in the later generated Java code. So we have to use the @WebFault annotation to give the Bean for the data a name that is separate from the Exception name.

```
package com.example.customerservice;

@WebFault(name="NoSuchCustomer")
@XmlAccessorType( XmlAccessType.FIELD )
public class NoSuchCustomerException extends RuntimeException {
    /**
      * We only define the fault details here. Additionally each fault has a message
      * that should not be defined separately
      */
    String customerName;
}
```

Service definition

```
package com.example.customerservice;

@WebService
public interface CustomerService {
    public Customer[] getCustomersByName(@WebParam(name="name") String name) throws NoSuchCustomerException;
}
```

As you can see only two annotations are necessary here. @WebService marks the interface as a service and @Webparam is necessary as Java will else lose the name of the parameter and the wsdl will contain arg0 instead of the desired name. Using the @WebService annotation you can also customize the port name and service name.

How is the wsdl generated

To generate the wsdl the maven plugin cxf-java2ws-plugin is used. See the pom.xml in the complete example for details.

Let´s take a look at the resulting WSDL

You can download the wsdl this example creates here.

```
<xs:complexType name="customer">
    <xs:sequence>
        <xs:element minOccurs="0" name="name" type="xs:string"/>
        <xs:element maxOccurs="unbounded" minOccurs="0" name="address" nillable="true" type="xs:string"/>
        <xs:element name="numOrders" type="xs:int"/>
        <xs:element name="revenue" type="xs:double"/>
        <xs:element minOccurs="0" name="test" type="xs:decimal"/>
        <xs:element minOccurs="0" name="birthDate" type="xs:dateTime"/>
        <xs:element minOccurs="0" name="type" type="tns:customerType"/>
    </xs:sequence>
</xs:complexType>
```

The customer Class is a complex type in xsd. As you can see the Java types have been described as their respective XSD primitive types.

Each element has minOccurs="0" which marks it as optional. This is a good thing as you can add new optional elements and keep compatible. If you do not want this optionality you can use @XmlElement(required=true).

The array of Strings for address is described as maxOccurs="unbounded" so the element may be repeated in the later xml to form the array.

Enumeration customerType

The enumeration customerType is described as a simple type with a restriction:

```
<xs:simpleType name="customerType">
    <xs:restriction base="xs:string">
        <xs:enumeration value="PRIVATE"/>
        <xs:enumeration value="BUSINESS"/>
    </xs:restriction>
</xs:simpleType>
```

Fault NoSuchCustomerException

The Exception we defined is generated as a complexType with the defined details and a message for the fault. It is important here that the Element and the Message have different names. We ensure this by using the @WebFault Annotation above. Else the later Java Code generation will produce an ugly Exception name NoSuchCustomerException_Exception.

```
<xs:element name="NoSuchCustomer" type="tns:NoSuchCustomer"/>
<xs:complexType name="NoSuchCustomer">
    <xs:sequence>
        <xs:element name="customerName" nillable="true" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
<wsdl:message name="NoSuchCustomerException">
    <wsdl:part name="NoSuchCustomerException" element="tns:NoSuchCustomer">
    </wsdl:part>
</wsdl:message>
```

The wsdl defines a SOAP/HTTP binding by default but can also be used to build services based on JMS as I will show in my next post.

Summary

As you can see the generated WSDL looks quite clean and correctly expresses the service interface we wanted to describe. In most cases where you are not satisfied with what the conversion does you can correct the WSDL using JAX-WS or JAXB annotations. But I recommend to use them sparsely to keep the DSL easy to read.

In the next post I will show ho to build service consumers and providers using the WSDL we just built.

You can download the complete example here.

References

- Apache CXF
- JAX-WS specification
- JAXB Architecture
- JAXB user guide
- JAXB Tutorial