1. Bindings

1.1. Bindings Overview

Bindings are one of the most important pieces of XFire. They control how the incoming and outoing XML is mapped to objects.

The default binding in XFire is "Aegis". Aegis means "shield", and the point of it is to create a simple layer which makes going between objects and XML easy. Aegis also provides the layer which allows other bindings to integrate with XFire - like JAXB, XMLBeans and Castor.

Lets take a look at how you would use a different binding with XFire. The BindingProvider class is the one for actually taking the xml streams and reading/writing to them. When we create a service we need to actually supply a BindingProvider to the service. In the simplest case of creating a service you're probably writing code like this:

```
ServiceFactory factory = new ObjectServiceFactory();
```

But, what this is actually doing in the constructor is creating an AegisBindingProvider:

```
ServiceFactory factory = new ObjectServiceFactory(new AegisBindingProvider());
```

1.2. Using another binding

This AegisBindingProvider has the concept of a TypeRegistry and TypeCreators (don't worry you don't need to know much about this), which create Type classes for you. Types serialize and deserialize java classes. When you use a different binding it actually provides a different TypeRegistry and TypeCreator for its types.

So if you were to create a JAXB 2.0 binding you would do:

```
import org.codehaus.xfire.aegis.AegisBindingProvider;
import org.codehaus.xfire.jaxb2.JaxbTypeRegistry;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;
...

ServiceFactory factory = new ObjectServiceFactory(new AegisBindingProvider(new
```

```
JaxbTypeRegistry()));
```

Each binding has a different type registry:

Binding	Type Registry
Default POJOs	org.codehaus.xfire.aegis.DefaultTypeMappingRegistry
Castor	org.codehaus.xfire.castor.CastorTypeMappingRegistry
JAXB 1.1	org.codehaus.xfire.jaxb.JaxbTypeRegistry
JAXB 2.0	org.codehaus.xfire.jaxb2.JaxbTypeRegistry
XMLBeans	org.codehaus.xfire.xmlbeans.XmlBeansTypeRegistry

But don't stop here, you can use these BindingProviders with different ServiceFactorys. Here is an example using JAXB 2.0 and the annotation service factory.

```
import org.codehaus.xfire.aegis.AegisBindingProvider;
import org.codehaus.xfire.jaxb2.JaxbTypeRegistry;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;
...

ServiceFactory factory = new AnnotationServiceFactory(new Jsr181WebAnnotations(),
  new AegisBindingProvider(new JaxbTypeRegistry()));
```

1.3. Convenience ServiceFactorys

If you just want to use the simple ObjectServiceFactory we have several convenience ServiceFactorys around:

Binding	Service Factory

JAXB 1.1	org.codehaus.xfire.jaxb.JaxbServiceFactory
JAXB 2.0	org.codehaus.xfire.jaxb2.JaxbServiceFactory
XMLBeans	org.codehaus.xfire.xmlbeans.XmlBeansServiceFactory

You can use these just like you would an ObjectServiceFactory:

```
import org.codehaus.xfire.jaxb2.JaxbServiceFactory;
import org.codehaus.xfire.service.ServiceFactory;
...
ServiceFactory factory = new JaxbServiceFactory();
```

1.4. The MessageBinding

There is also one other type of binding, the MessageBinding. The MessageBinding has special semantics to allow you to work with XML streams and fragments real easily. Read more on the Message Binding page.

1.5. Aegis Binding

Aegis is the default XFire binding which maps XML to POJOs. It supports code first development only at this point - i.e. you write your service in POJOs and it will generate the XML schema/wsdl for you.

1.6. XML and Annotation Mapping Overview

Aegis has a flexible mapping system so you can control how your beans are controlled. By default your POJOs are serialized based on their name and namespaces. If you have a class in the "org.codehaus.xfire" package named "Employee" it would be serialized in namespace "http://xfire.codehaus.org" with the local name "YourBean."

Fore example, the java class:

```
public class Employee
{
  private String name;
  private String title;

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }
```

```
}
```

In XML this translates to:

```
<Employee xmlns="http://xfire.codehaus.org">
  <name>Santa Claus</name>
  <title>Chief Present Officer (CPO)</title>
</Employee>
```

In XML Schema this would become a complex type:

```
<xsd:complexType name="Employee">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs="1/>
    <xsd:element name="title" type="xsd:string" minOccurs="0" maxOccurs="1/>
  </xsd:sequence>
</xsd:complexType>
```

Validate your mapping! You can find an XML Schema for Aegis mapping files here .

1.6.1. Supported Types

- Basic types: int, double, float, long, byte[], short, String, BigDecimal
- Arrays
- Collections
- Dates: java.util.Date, java.util.Calendar, java.sql.Timestamp, java.sql.Date, java.sql.Time
- XML: org.w3c.dom.Docmument, org.jdom.Element, XMLStreamReader, Source
- Complex types which are aggregations of the above

If you have constructors defined in your Java beans, make sure a default constructor (i.e. no arguments) is also declared. (Aegis needs a no-argument contstructor to instantiate client Java classes.) Controlling Mappings with XML

Its easy to control how your service and its beans are mapped to xml. If you are using Java 5.0 skip straight down to that section otherwise read on to learn how to configure serialization via mapping files.

Mapping files must exist in the same package as your bean or service class on the class path. In the above example the mapping file would be named "/org/codehaus/xfire/YourBean.aegis.xml", with the following format:

```
<mappings>
```

```
  <mapping uri="" name="">
    <method name="methodName">
      <return-type mappedName="" componentType=""/>
      <parameter index="" mappedName=""/>
    </method>
    <property name="" mappedName="" style="attribute|element" componentType=""/>
  </mapping>
</mappings>
```

Note that <method>is used to configure methods on your service and property is used to configure properties on your javabeans.

The above example highlights many of the possible elements, most are optional and the format encourages minimally specified mappings.

1.7. Controlling Naming

Lets pretend that in the above example you would like the elements names to be capatilized and in the namespace "urn:north-pole:operations". You could achieve this through a mapping file like so:

```
<mappings xmlns:np="urn:north-pole:operations">
  <mapping name="np:Employee">
    <property name="name" mappedName="Name"/>
    <property name="title" mappedName="Title"/>
  </mapping>
</mappings>
```

Notice that the namespace was declared on the mappings element and then the prefix was used to specify the element QNames for the name/title properties.

This will result in a mapping like so:

```
<np:Employee xmlns:np="urn:north-pole:operations">
  <np:Name>Santa Claus</np:Name>
  <np:Title>Chief Present Officer (CPO)</np:Title>
</np:Employee>
```

1.7.1. Ignoring properties

If you don't want to serialize a certain property it is easy to ignore it:

```
<mappings>
  <mapping>
    <property name="propertyName" ignore="true"/>
  </mapping>
</mappings>
```

1.7.2. Handling Collections

You undoubtedly use Collections in your code. Pre Java 5 it is impossible to determine the "component type" of a Collection by introspection. So you need to give Aegis some hints. For a service which returned a Collection of employees like so:

```
public class EmployeeService
{
    Collection getEmployees(String id) { ... }
}
```

You would need to supply metadata which gave the component type in a mapping file like this one:

```
<mappings>
  <mapping>
    <method name="getEmployees">
      <return-type componentType="org.codehaus.xfire.Employee" />
    </method>
  </mapping>
</mappings>
```

1.7.3. Handling Maps

Java Maps don't map well to XML Schema (no pun intended) because there is no Map concept in XML Schema so your clients. Maps are transformed to a collection of key, value tuples instead. In addition to providing the type of the value, you must also provide Aegis with the type of the key:

```
public class GiftService
{
    Map getGiftList() { /* returns a map of NiceChild => Present */ }
}
```

The mapping file should look like this:

```
<mappings>
  <mapping>
```

```
    <method name="getGiftList">
      <return-type keyType="org.codehaus.xfire.NiceChild"
componentType="org.codehaus.xfire.Present">
    </method>
  </mapping>
 </mappings>
```

This will generate the following type:

```
<xsd:complexType name="NiceChild2PresentMap">
  <xsd:sequence>
    <xsd:element name="entry" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="key" type="ns1:NiceChild" minOccurs="0" maxOccurs="1"/>
          <xsd:element name="value" type="ns1:Present" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

1.8. Interfaces and Aegis

The Aegis binding will automatically create proxies for your interfaces when reading XML. So if you have an interface like this:

```
public interface User {
  public String getUsername();
  public String getPasswrod();
}
```

It will then create its own implementation of User and provide a username and password from the XML. You can specify your implementation class (in 1.1+) by setting a property on your service:

```
Service service = ...;
service.setProperty("com.acme.User.implementation", "com.acme.UserImpl");
```

1.9. Castor

Castor is a flexible XML binding tool that provides run-time marshalling and unmarshalling of XML and Java objects. One strength of Castor when compared to most (not all) other Java XML binding frameworks is that re-compilation of the Java code is not required if the mapping definition changes. Therefore, systems where the web service layer is being developed independently from the business layer can benefit from using Castor. XFire support for Castor is currently available in the latest XFire release. Two approaches to developing a Web service using Castor with XFire are

presented below: top-down (schema first) and bottom-up (code first). Before proceeding, check the Dependency Guide for required castor module dependencies.

1.9.1. Assumptions about Reader

- Competence with Java and XML
- Basic knowledge of Castor XML binding framework
- Experience configuring Java webapp and deploying
- Nominal familiarity with Spring framework

1.10. Top-down Approach (starting with XML schema):

Firstly, the XML schema that defines the structure of your web service messages must be developed. For the purposes of this guide, we'll borrow a schema from http://www.webservicex.net/WeatherForecast.asmx?WSDL which defines a pre-existing weather forecast service.

The borrowed schema below should be saved under META-INF/schema/ in the classpath:

```
 <s:schema elementFormDefault="qualified" targetNamespace="http://www.webservicex.net"
          xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://www.webservicex.net">
      <s:element name="GetWeatherByZipCode">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="ZipCode" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetWeatherByZipCodeResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="GetWeatherByZipCodeResult"
type="tns:WeatherForecasts"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="WeatherForecasts">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="Latitude" type="s:float"/>
          <s:element minOccurs="1" maxOccurs="1" name="Longitude" type="s:float"/>
```

```
        <s:element minOccurs="1" maxOccurs="1" name="AllocationFactor" type="s:float"/>
        <s:element minOccurs="0" maxOccurs="1" name="FipsCode" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="PlaceName" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="StateCode" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="Status" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="Details"
type="tns:ArrayOfWeatherData"/>
      </s:sequence>
    </s:complexType>
    <s:complexType name="ArrayOfWeatherData">
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="unbounded" name="WeatherData"
type="tns:WeatherData"/>
      </s:sequence>
    </s:complexType>
    <s:complexType name="WeatherData">
      <s:sequence>
        <s:element minOccurs="0" maxOccurs="1" name="Day" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="WeatherImage" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="MaxTemperatureF"
type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="MinTemperatureF" type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="MaxTemperatureC"
type="s:string"/>
        <s:element minOccurs="0" maxOccurs="1" name="MinTemperatureC" type="s:string"/>
      </s:sequence>
    </s:complexType>
  </s:schema>
```

Next, we'll use castor to generate POJO classes from the service schema. First, define the source generator task and a goal for generation. The example below is for a maven 1.x configuration. Go to Using the Source Code Generator for a full reference on generating java classes from XML schema.

```
<ant:path id="castor.class.path">
        <ant:path refid="maven.dependency.classpath"/>
        <ant:pathelement path="${maven.build.dest}"/>
        <ant:pathelement path="${maven.test.dest}"/>
</ant:path>

<ant:taskdef name="castor-srcgen"
                classname="org.exolab.castor.tools.ant.taskdefs.CastorSourceGenTask"
                classpathref="castor.class.path"/>

<goal name="generate-source">
        <ant:delete dir="${maven.build.dir}/generated-src"/>
        <ant:mkdir dir="${maven.build.dir}/generated-src"/>
        <castor-srcgen file="src/main/META-INF/schema/WeatherForecast.xsd"
                        package="net.webservicex"
                        todir="${maven.build.dir}/generated-src"
                        types="j2"
                        warnings="false" />
</goal>
```

After running the generate-source goal, the supporting classes will be in the generated-src directory under the net.webservicex package. Along with each class is an accompanying Descriptor class file (e.g. WeatherDataDescriptor.java) that contains XML binding information.

Next, write a Web service with the support of the generated classes. Note, the example below is a trivial implementation.

```
package foo.bar;

import net.webservicex.*;

public class WeatherService
{

    public GetWeatherByZipCodeResponse GetWeatherByZipCode(GetWeatherByZipCode
body)
    {
        GetWeatherByZipCodeResponse res = new GetWeatherByZipCodeResponse();
```

```
        String zipCode = body.getZipCode();
        if (!zipCode.equals("1050"))
            throw new RuntimeException("Parameter isnt passed correctly. expected: 1050, got "
                    + zipCode);
        GetWeatherByZipCodeResult weather = new GetWeatherByZipCodeResult();

        weather.setLatitude(1);
        weather.setLongitude(1);
        weather.setPlaceName("Vienna, AT");
        weather.setAllocationFactor(1);

        res.setGetWeatherByZipCodeResult(weather);

        return res;
    }

}
```

After this, configure the service in Xfire. The example below takes the XML configuration approach (which uses Spring integration). In addition, XFire supports integration of configuration into containers such as Plexus and PicoContainer. For more control over the Web service definition, JSR 181 Annotations should be used in the service code.

The configuration goes in the services.xml descriptor. This file goes in META-INF/xfire/ on the classpath:

```
 <beans xmlns="http://xfire.codehaus.org/config/1.0">

   <service>
     <serviceClass>foo.bar.WeatherService</serviceBean>
     <schemas>
       <schema>META-INF/schema/WeatherForecast.xsd</schema>
     </schemas>
     <style>document</style>
     <serviceFactory>#castorServiceFactory</serviceFactory>
   </service>

   <bean id="castorTypeRegistry"
class="org.codehaus.xfire.castor.CastorTypeMappingRegistry" />

   <bean id="bindingProvider" class="org.codehaus.xfire.aegis.AegisBindingProvider">
         <constructor-arg ref="castorTypeRegistry" />
   </bean

   <bean id="castorServiceFactory"
class="org.codehaus.xfire.service.binding.ObjectServiceFactory">
     <constructor-arg index="0" ref="xfire.transportManager" />
     <constructor-arg index="1" ref="bindingProvider" />
    </bean>

 </beans>
```

The service definition specifies the web service implementation class with the serviceClass property. A list of <schemas>can be provided to be included in WSDL generation. In this case, the borrowed schema definition in the classpath at META-INF/schema/WeatherForecast.xsd has been included.

The request and response messages were defined in our schema and were included as a single parameter and return type, respectively, in our service method. This lends itself to a bare (or unwrapped) parameter-style. The service definition specifies a bare parameter-style by setting the style property as 'document'. If a wrapped parameter-style is preferable, the request and response schema definitions and generated classes would not be needed, as the operation name would wrap any request parameters and return type. The style property in the service definition would

not be needed as service factories create services as wrapped style by default. The service method signature for a wrapped style service would be:

```
public WeatherForecasts GetWeatherByZipCode(String zipCode)
```

Service factories are responsible for creating the service inside of XFire. In this case, we want to use an ObjectServiceFactory with Castor binding, so the beans 'castorTypeRegistry', 'bindingProvider' and 'castorServiceFactory' are defined. Notice that these beans are using spring-style bean definitions (e.g. bean id=...) since XML Configuration uses Spring to build its services. We reference this service factory in the serviceFactory property of the service definition as #castorServiceFactory. The '#' denotes a reference to another defined bean. Read the Bindings section to learn more about setting up service factories with different xml binding mechanisms.

After configuring the service within XFire, the service needs to be exposed so it can be reached by client-proxies. This can be done over HTTP by defining a servlet in the web.xml. Since we have taken the XML Configuration approach, it is best to set up the XFireConfigurableServlet in our web.xml. XFireConfigurableServlet will load the services.xml along with the included org.codehaus.xfire.spring.xfire.xml files upon initialization. Reference the XML Configuration for how to write the web.xml file.

Finally, deploy the web application to your favorite servlet container, and the service should be visible at /services/WeatherService?wsdl under the deployed webapp context path.

1.11. Bottom-up Approach (starting with Java classes):

Let's say you want to leverage your pre-existing business code by exposing certain methods with a Web Service. That's not too hard to do with Castor binding and XFire, but it's a different approach that involves mapping Java to XML rather than generating Java classes from schema. Service-oriented architecture discourages simply exposing the domain model, but in this example we're going to be rebellious and not insulate it. This doesn't mean that the bottom-up approach can't follow SOA best practices, altogether.

Below is a Book class that's part of the domain:

```
package foo.bar;

public class Book
{
    private String title;

    private String isbn;
```

```
    private String author;

    public String getIsbn()
    {
        return isbn;
    }

    public void setIsbn(String isbn)
    {
        this.isbn = isbn;
    }

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }

    public String getAuthor()
    {
        return author;
    }

    public void setAuthor(String author)
    {
        this.author = author;
    }
}
```

Next, is a service class that has a couple methods we want to expose as a web service. Note: This service class could have existed already as a business service, or could have been written specifically to be a web service. As you might have noticed, the implementation is quite trivial (ie. stoopid).

```
package foo.bar;

public class BookService
{
    private Book onlyBook;

    public BookService()
    {
        onlyBook = new Book();
        onlyBook.setAuthor("Steve Ballmer");
        onlyBook.setTitle("How to Yell Real Loud and Look Like You Might Have a Heart
Attack");
        onlyBook.setIsbn("012924828");
    }

    public String addBook(Book book)
    {
        return onlyBook.getIsbn();
    }

    public Book findBook(String isbn)
    {
        return onlyBook;
    }
}
```

Let's create an XSD that matches the Book class, so that we know what the XML element for the Book class should look like. The Book.xsd schema below should be saved under META-INF/schema/ in the classpath:

```
<xsd:schema elementFormDefault="qualified" targetNamespace="http://xfire.codehaus.org"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://xfire.codehaus.org">
    <xsd:complexType name="Book">
        <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="isbn" type="xsd:string"/>
            <xsd:element name="author" type="xsd:string"/>
```

```
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

The schema outlines a complex type that has three string elements. It is important to note that the namespace prefix for XML schema namespace is 'xsd' and the namespace prefix for the target namespace is 'tns'. These are the prefixes used by the XFire WSDL builder when creating the schema section, and this schema will actually be inserted inline into the schema section of the WSDL.

Now we need to specify how to transform our Book class to an XML element that conforms to the XSD, and vice versa. This is done with a castor mapping file. More information on writing castor mappings is available at http://www.castor.org/xml-mapping.html

The castor.xml mapping file below should be saved under the foo.bar package:

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
  <class name="org.codehaus.xfire.castor.Book">
    <map-to xml="Book" ns-uri="http://xfire.codehaus.org" ns-prefix="tns" element-
definition="false"/>
    <field name="title" type="string"/>
    <field name="isbn" type="string"/>
    <field name="author" type="string"/>
  </class>
</mapping>
```

Note again, the target namespace and tns prefix are the same as in the XSD. The element-defintion attribute denotes whether the XSD definition is a concrete element or an abstract complex type. In this case, it is a complex type so element-definition is false (which is actually is default value).

All the distinct parts are there, time to configure the service in Xfire. The example below takes the XML configuration approach (which uses Spring integration).

The configuration goes in the services.xml descriptor. This file goes in META-INF/xfire/ on the classpath:

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">

  <service>
    <serviceClass>foo.bar.BookService</serviceBean>
    <namespace>http://xfire.codehaus.org</namespace>
    <schemas>
      <schema>META-INF/schema/Book.xsd</schema>
    </schemas>
    <serviceFactory>#castorServiceFactory</serviceFactory>
```

```
  </service>

  <bean id="castorTypeRegistry"
class="org.codehaus.xfire.castor.CastorTypeMappingRegistry">
      <property name="mappingFile" value="foo/bar/castor.xml" />
  </bean>

  <bean id="bindingProvider" class="org.codehaus.xfire.aegis.AegisBindingProvider">
       <constructor-arg ref="castorTypeRegistry" />
  </bean

  <bean id="castorServiceFactory"
class="org.codehaus.xfire.service.binding.ObjectServiceFactory">
    <constructor-arg index="0" ref="xfire.transportManager" />
    <constructor-arg index="1" ref="bindingProvider" />
   </bean>

 </beans>
```

The service definition specifies the web service implementation class with the serviceClass property. A list of schemas can be provided to be included in WSDL generation. In this case, the schema definition in the classpath at META-INF/schema/Book.xsd has been included. Our service is of parameter style 'wrapped', as the method names in the service will wrap our request and response messages (e.g. findBook). The style property in the service definition is not be needed as service factories create services as wrapped style by default.

Service factories are responsible for creating the service inside of XFire. In this case, we want to use an ObjectServiceFactory with Castor binding, so the beans 'castorTypeRegistry', 'bindingProvider' and 'castorServiceFactory' are defined. Notice that these beans are using spring-style bean definitions (e.g. bean id=...) since XML Configuration uses Spring to build its services. We reference this service factory in the serviceFactory property of the service definition as #castorServiceFactory. The '#' denotes a reference to another defined bean. The 'castorTypeRegistry' definition sets the 'mappingFile' property to point to the castor.xml mapping file in the classpath.

After configuring the service within XFire, the service needs to be exposed so it can be reached by client-proxies. This can be done over HTTP by defining a servlet in the web.xml. Reference the XML Configuration for how to write the web.xml file.

After configuring the service and web application, deploy it to your favorite servlet container, and the service should be visible at /services/BookService?wsdl under the deployed webapp context path.

2. Transports

2.1. Transports and Channels

Channels and Transports provide the basic unit of communication in XFire. A Channel simply sends messages (via send()) and listens for messages (via receive()). If you want to send a message to a particular URL you do:

```
TransportManager tm = ...;
Transport t = tm.getTransportForUri("http://foo");

OutMessage msg = ...; // create an outmessage yourself
Channel c = t.createChannel(); // create an anonymous endpoint
MessageContext context = new MessageContext();
c.send(msg, context);
```

Each transport is responsible for creating its own protocol specific listener, for example a servlet in the case of HTTP. This listener then passes whatever messages it receives to the channel via Channel.receive(MessageContext, InMessage). Channels simply delegate their receive() to a ChannelEndpoint which application specific handling of what to do with the message. The default endpoint is aptly named DefaultEndpoint and will be covered in the next section.

2.1.1. InMessage and OutMessage

2.1.2. MessageSerializer

Each OutMessage has a MessageSerializer. A message serializer takes the message body (message.getBody()) and writes it to an XMLStreamWriter that the Channel provides. The semantics of MessageSerializers should be such that they can be invoked multiple times.

2.1.3. Channels/Transports without Services

Its important to note that Channels and transports are completely independent of XFire's Services. So I can use a channel to send a receive messages and never even create a service. I simply need to provide my own ChannelEndpoint.

2.1.4. DefaultEndpoint and the processing flow

DefaultEndpoint takes a message, creates a default message exchange called InMessageExchange and creates a message pipeline. The message pipeline at first consists of the global in handlers from XFire.getInHandlers() and the transport handlers from Transport.getInHandlers. Later on when the service is resolved, the service's handlers get added into the pipeline. Once the operation is resolved, if there is an out message to be set an Out pipeline is created and added to the MessageContext.

2.1.5. Phases and Handlers

See Processing Pipeline for now.

2.1.6. SOAP Processing

SOAPTransport.createTransport() adds SOAP support to a particular transport. It does so by adding three additional handlers:

1. ReadHeaderHandler - this handler reads in the soap headers. It stops reading the XML stream the moment the whitespace stops after the Body tag. If it encounters a Fault in the Body, an XFireFault is thrown. [[1]] ValidateHeadersHandler - Ensure that all the necessary headers are understood by the receiving handlers. [[1]] SoapSerializerHandler - Services provide a MessageSerializer of their own which is responsible for serializing the soap body. But what about the message headers? This is written out by the SoapSerializer. What the SoapSerializerHandler does is replace the outMessage.MessageSerializer with new SoapSerializer(outMsg.getMessageSerializer()).

2.1.7. MessageExchanges

2.2. HTTP Transport

2.2.1. XFireServlet

The core of the HTTP Transport takes place in the XFireServletController. Your own servlets can

delegate appropriate requests to this class or you can use one of XFire's internal servlet classes. The XFireServlet is just a thin wrapper for the controller. The XFireServletController provides an xml configuration layer on top of this.

XFire also provides the XFireConfigurableServlet which reads the services.xml format automatically for you and the XFireSpringServlet which provides Sprign integration.

2.2.2. HttpServletRequest/HttpServletResponse

The HttpServletRequest/HttpServletResponse can be accessed via the XFireServletController.

```
HttpServletRequest request = XFireServletController.getRequest();
HttpServletResponse response = XFireServletController.getResponse();
```

This method will work all the XFire servlets (XFireServlet, XFireConfigurableServlet, XFireSpringServlet).

2.2.3. Client authentication

The Apache Jakarta HttpClient is used under the covers to provide HTTP client support. There are two ways which you can override the HttpClient settings:

1. You can set the USERNAME/PASSWORD

   ```
   // Create your client
   Client client = ....;

   // Or get it from your proxy
   Client client = ((XFireProxy) Proxy.getInvocationHandler(myClientProxy)).getClient();

   client.setProperty(Channel.USERNAME, "username");
   client.setProperty(Channel.PASSWORD, "pass");
   ```

2. You can supply your own HttpClientParms

   ```
   client.setProperty(CommonsHttpMessageSender.HTTP_CLIENT_PARAMS, myParams);
   ```

   The HTTPClient javadocs provide information on how to configure the HttpClientParams.

2.2.4. Proxy Support

Proxy support looks very similar to the username/password scenario:

```
// Create your client
Client client = ....;

// Or get it from your proxy
Client client = ((XFireProxy) Proxy.getInvocationHandler(myClientProxy)).getClient();
client.setProperty(CommonsHttpMessageSender.HTTP_PROXY_HOST, "host");
client.setProperty(CommonsHttpMessageSender.HTTP_PROXY_PORT, "8080");
```

2.2.5. HTTP Chunking

You'll need to enable HTTP chunking on the client if you are sending large files which can't be cached in memory:

```
import org.codehaus.xfire.transport.http.HttpTransport;

Client client = ....;
client.setProperty(HttpTransport.CHUNKING_ENABLED, "true");
```

2.3. JMS Transport

This guide gives you a quick rundown of how to configure XFire to use JMS as a transport. JMS is one of the easiest means to create a reliable SOAP connection. Additionally it is much faster then things such as WS-Reliability.

This example assumes that you already know how to:

- Configure services via XFire's services.xml format
- Build and deploy simple XFire applications
- Use your JMS provider
- A working knowledge of Spring

We're just going to show a simple synchronous Echo example running over JMS. The first thing you need to do is create your services.xml file:

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
```

```
  <!--
    Register the JMS transport. Note: this needs to happen *before* we
    create our service.
    -->
  <xfire>
    <transports>
      <bean id="jmsTransport"
            class="org.codehaus.xfire.transport.jms.JMSTransport"
            xmlns="http://xbean.org/schemas/spring/1.0">
        <constructor-arg ref="xfire"/>
        <constructor-arg ref="connectionFactory"/>
      </bean>
    </transports>
  </xfire>

  <service xmlns:e="urn:Echo">
    <name>Echo</name>
    <serviceClass>org.codehaus.xfire.test.Echo</serviceClass>
    <implementationClass>org.codehaus.xfire.test.EchoImpl</implementationClass>
    <bindings>
      <soap11Binding name="e:EchoJMSBinding" transport="urn:xfire:transport:jms">
        <endpoints>
          <endpoint name="e:EchoJMSEndpoint" url="jms://Echo" />
        </endpoints>
      </soap11Binding>
    </bindings>
  </service>

  <bean id="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory"
        singleton="true"
        xmlns="http://xbean.org/schemas/spring/1.0/">
    <constructor-arg value="vm://localhost?broker.persistent=false" type="java.lang.String"/>
  </bean>

</beans>
```

There is a lot in here, so lets recap this a little bit.

The xfire section contains a transports element. In transports we are creating our JMSTransport via the Spring bean syntax. XFire will then automatically register this transport for us into the TransportManager.

The service element contains our service definition. This is pretty standard, except you'll notice we're creating a new binding for JMS. soap11Binding transport="urn:xfire:transport:jms"tells XFire that we want to add a SOAP 1.1 binding for JMS. In the endpoints section we tell XFire exactly what that endpoint will be. The JMS urls take the form of jms:// QueueName .

In the sections below we configure our JMS QueueConnectionFactory using ActiveMQ.

Once all of this is properly configured we will of course want to write a client:

```
import java.lang.reflect.Proxy;

import org.codehaus.xfire.client.XFireProxy;
import org.codehaus.xfire.client.XFireProxyFactory;
import org.codehaus.xfire.service.Service;
import org.codehaus.xfire.service.binding.ObjectServiceFactory;
import org.codehaus.xfire.spring.AbstractXFireSpringTest;
import org.codehaus.xfire.test.Echo;
import org.codehaus.xfire.transport.jms.JMSTransport;
import org.springframework.context.ApplicationContext;
import org.apache.xbean.spring.context.ClassPathXmlApplicationContext;

public class JMSExampleTest
    extends AbstractXFireSpringTest
{
    protected ApplicationContext createContext()
    {
        return new ClassPathXmlApplicationContext(new String[] {
                "/org/codehaus/xfire/transport/jms/example/jms.xml",
                "/org/codehaus/xfire/spring/xfire.xml" });
    }

    public void testClient()
            throws Exception
    {
        // Create a ServiceFactory to create the ServiceModel.
        // We need to add the JMSTransport to the list of bindings to create.
        ObjectServiceFactory sf = new ObjectServiceFactory(getTransportManager());
        sf.addSoap11Transport(JMSTransport.BINDING_ID);
```

```
        // Create the service model
        Service serviceModel = sf.create(Echo.class);

        // Create a proxy for the service
        XFireProxyFactory factory = new XFireProxyFactory(getXFire());
        Echo echo = (Echo) factory.create(serviceModel, "jms://Echo");

        // Since JMS doesn't really have a concept of anonymous endpoints, we need
        // need to let xfire know what JMS endpoint we should use
        ((XFireProxy)
Proxy.getInvocationHandler(echo)).getClient().setEndpointUri("jms://Peer1");

        // run the client!
        String resString = echo.echo("hello");
        assertEquals("hello", resString);
    }
}
```

2.4. Local Transport

XFire includes an in-JVM transport called the LocalTransport. If you are using the XFire client or the Channel API, you can address a local service like so:

```
xfire.local://FooService
```

For example, when creating a service:

```
Service service = getServiceRegistry().getService("FooService");

XFireProxyFactory factory = new XFireProxyFactory(getXFire());
FooService foo = (FooService) factory.create(service, "xfire.local://FooService");
```