Configuration

Configuration

Subpages

- Bus Configuration
- FeaturesList
- JMX Management
- WSAConfiguration
- WSPConfiguration
- WSRMConfiguration

Supplying a Configuration file to CXF

CXF can discover XML configuration files which you have written. For both web service clients and servers, the default location that CXF will look for a configuration for is "/cxf.xml" on the class path. For example, when running your application in a servlet container, this file is expected to be located in a /WEB-INF/classes folder of your web application.

If you wish to override this location, you can specify a command line property: -Dcxf.config.file=some_other_config.xml. This custom configuration file is also expected to be on the class path.

To use a url as the configuration location, specify as follows: -Dcxf.config.file.url=config_file_url.

A CXF configuration file is really a Spring configuration file, so all configuration files will start with the following:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<!-- Configuration goes here! -->

</beans>
```

If you are new to Spring or do not desire to learn more about it, don't worry, you won't have to. The only piece of Spring that you will see is the <beans> element outlined above. Simply create this file, place it on your classpath, and add the configuration for a component you wish to configure (see below). Note starting with CXF 2.6.0, Maven users will need to add the following dependency for the cxf.xml file to be read:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>3.0.6.RELEASE</version>  (or most recent supported)
</dependency>
```

Types of Configuration files

Client configuration file

Placing a cxf.xml file (or other-named file as configured above) in the classpath of the Web Service Client can be used to configure client-specific functionality. For example, the following client cxf.xml file turns off chunked transfer encoding for a specific service in requests and responses:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
       xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration
           http://cxf.apache.org/schemas/configuration/http-conf.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http-conf:conduit
           name="{urn:ebay:apis:eBLBaseComponents}Shopping.http-conduit">
      <http-conf:client AllowChunking="false"/>
  </http-conf:conduit>
</beans>
```

Server configuration files

Typically, the cxf.xml file in the classpath of the web service is intended primarily for configuration of the CXF bus, the object used for the creation of all services and endpoints. Endpoint configuration is primarily done either via a WEB-INF/cxf-servlet.xml file or a Spring application context file designated by the web application deployment descriptor (web.xml file). The cxf-servlet.xml file is somewhat slower because it loads all possible CXF modules for an endpoint; the Spring application context method is faster because it allows you to specify which CXF modules are needed.

For an example configuration via a cxf-servlet.xml file, our system tests have a cxf-servlet.xml file which is explicitly referenced in the init-param element in the web.xml deployment descriptor. (Note it is not necessary to use the **init-param** element if you use the file name "cxf-servlet.xml"; this is the default name that CXF uses to look for such a file.)

For an example of using a Spring application context file for endpoint configuration, refer to our Java First Spring Support sample. You can see how the web.xml deployment descriptor explicitly references the beans.xml application context file via a **context-param** element (and ContextLoaderListener object); also that the application context file manually imports the three cxf modules that it needs.

What can I configure and how do I do it?

If you want to change CXF's default behaviour, enable specific functionality or fine tune a component's behaviour, you can in most cases do so without writing a single line of code, simply by supplying a Spring configuration file.
In some cases it also possible to achieve the same end by extending your wsdl contract: you can add CXF specific extension elements to the wsdl:port element and in that way fine tune the behaviour of the specified transport. Or you can use WS-Policy to express the fact that your application uses WS-Addressing, for example.

Using Spring configuration files however is the most versatile way to achieve a specific goal: you can use it to

1. Enable functionality via simple constructs called features.
2. Set properties of runtime components by referring to these runtime components using either plain Spring bean elements, or, more conveniently, using CXF custom beans elements.
3. Modify the actual composition of the runtime (change the way the runtime is wired up).

The following examples show the what the Spring configuration would look like if you wanted to enable the logging of inbound and outbound messages and faults.

Enabling message logging using plain Spring bean elements

Using this format is STRONGLY discouraged as it ties your configuration with internal CXF class names (like SpringBus). It is much better to use the cxf:bus element described below

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="logInbound" class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
    <bean id="logOutbound" class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

    <bean id="cxf" class="org.apache.cxf.bus.spring.SpringBus">
        <property name="inInterceptors">
            <list>
                <ref bean="logInbound"/>
            </list>
        </property>
        <property name="outInterceptors">
            <list>
                <ref bean="logOutbound"/>
            </list>
        </property>
        <property name="outFaultInterceptors">
            <list>
                <ref bean="logOutbound"/>
            </list>
        </property>
        <property name="inFaultInterceptors">
            <list>
                <ref bean="logInbound"/>
            </list>
        </property>
    </bean>
</beans>
```

In this example, you specify that the CXF bus is implemented by class org.apache.cxf.bus.spring.SpringBus, and that its id is "cxf". This is the default, but you have to re-iterate the fact if you want the bus to contribute the logging interceptors to the outbound and inbound interceptor chain for all client and server endpoints. You can avoid this duplication by using the next form of configuration:

Enabling message logging using custom CXF bean elements

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xsi:schemaLocation="
http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="logInbound" class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
    <bean id="logOutbound" class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>

    <cxf:bus>
        <cxf:inInterceptors>
            <ref bean="logInbound"/>
        </cxf:inInterceptors>
        <cxf:outInterceptors>
            <ref bean="logOutbound"/>
        </cxf:outInterceptors>
        <cxf:outFaultInterceptors>
            <ref bean="logOutbound"/>
        </cxf:outFaultInterceptors>
        <cxf:inFaultInterceptors>
            <ref bean="logInbound"/>
        </cxf:inFaultInterceptors>
    </cxf:bus>
</beans>
```

Here, there is no need to specify the implementation class of the bus - nor the fact that the inInterceptors, outInterceptors, outFaultInterceptors, and inFaultInterceptors child elements are of type list. All of this information is embedded in the underlying schema and the bean definition parser that's invoked for <cxf:bus> elemens. Note that you have to specify the location of this schema in the schemaLocation attribute of the <beans> element so that Spring can validate the configuration file. But it gets even simpler in the next example:

Enabling message logging using the Logging feature

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xsi:schemaLocation="
http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>
</beans>
```

The following sections explain which of the above mechanisms (feature, custom CXF bean or plain Spring bean; or indeed others such as wsdl extensors or policy assertions) you can use to enable and/or configure the various components in CXF, and what your configuration options are:

- Bus
- JAX-WS Configuration
- HTTP Client and Server
- JMS Transport
- Soap Binding
- XML Binding
- WS-Addressing
- WS-Reliable Messaging
- WS-Policy Framework

For a list of available features, see here.

Advanced Configuration

If you are writing your own component for CXF, please see Configuration for Developers page.