Servlet Transport

Setting up your web.xml

To create services that use this transport you can either use the CXF APIs (for example, see JAX-WS) or create an XML file which registers services for you.

Publishing an endpoint from XML

CXF uses Spring to provide XML configuration of services. This means that first we'll want to load Spring via a Servlet listener and tell it where our XML configuration file is:

Next, you'll need to add CXFServlet to your web.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:com/acme/ws/services.xml
    </param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
       org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Alternatively, you can point to the configuration file using a CXFServlet init parameter :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>

  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
        org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <init-param>
      <param-name>config-location</param-name>
      <param-value>/WEB-INF/beans.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

</web-app>
```

The next step is to actually write the configuration file:

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

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>

  <jaxws:endpoint id="greeter"
      implementor="org.apache.hello_soap_http.GreeterImpl"
      address="/Greeter1"/>

  <jaxrs:server id="greeterRest"
      serviceClass="org.apache.hello_soap_http.GreeterImpl"
      address="/GreeterRest"/>

</beans>
```

Here we're creating a JAX-WS endpoint based on our implementation class, GreeterImpl.

NOTE: We're publishing endpoints "http://localhost/mycontext/services/Greeter1" and "http://localhost/mycontext/services/GreeterRest", but we set jaxws:endpoint/@address and jaxrs:server/@address to relative values such as "/Greeter1" "/GreeterRest".

Disabling the Services Page

By default, Apache CXF creates a /services page containing a listing of the available endpoints. To disable this listing, configure the Servlet as follows:

Disable Services Listing

```
<servlet>
    ....
    <init-param>
        <param-name>hide-service-list-page</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
```

Support for Asynchronous Requests

Enable an 'async-supported' servlet property if you work with Servlet3 API containers and need to support asynchronous requests:

```
<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
       org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>

    <!-- Enable asynchronous requests -->

    <async-supported>true</async-supported>

    <load-on-startup>1</load-on-startup>
</servlet>
```

Redirecting requests and serving the static content

Starting from CXF 2.2.5 it is possible to configure CXFServlet to redirect current requests to other servlets or serve the static resources.

"redirects-list" init parameter can be used to provide a space separated list of URI patterns; if a given request URI matches one of the patterns then CXFServlet will try to find a RequestDispatcher using the pathInfo of the current HTTP request and will redirect the request to it.

"redirect-servlet-path" can be used to affect a RequestDispatcher lookup, if specified then it will concatenated with the pathInfo of the current request.

"redirect-servlet-name" init parameter can be used to enable a named RequestDispatcher look-up, after one of the URI patterns in the "redirects-list" has matched the current request URI.

"static-resources-list" init parameter can be used to provide a space separated list of static resource such as html, css, or pdf files which CXFServlet will serve directly.

One can have requests redirected to other servlets or JSP pages.

CXFServlets serving both JAXWS and JAXRS based endpoints can avail of this feature.

For example, please see this web.xml.

The "http://localhost:9080/the/bookstore1/books/html/123" request URI will initially be matched by the CXFServlet given that it has a more specific URI pattern than the RedirectCXFServlet. After a current URI has reached a jaxrs:server endpoint, the response will be redirected by the JAXRS RequestDispatcherProvider to a "/book.html" address, see "dispatchProvider1" bean here.

Next, the request URI "/book.html" will be handled by RedirectCXFServlet. Note that a uri pattern can be a regular expression. This servlet redirects the request further to a RequestDispatcher capable of handling a "/static/book.html".

Finally, DefaultCXFServlet serves a requested book.html.

Serving welcome pages

Starting from CXF 2.5.5 and 2.6.2 it is possible to configure CXFServlet to serve welcome pages in a number of ways.

For example, lets assume we have a web application called "webapp" which has a root resource called "index.html". For CXFServlet to support both "/webapp" and "/webapp/index.html" requests returning "index.html", while letting all other requests to proceed to the actual endpoints, the following can be done.

Option1. Delegating to Default Servlet

```
<servlet>
   <servlet-name>CXFServlet</servlet-name>
   <display-name>CXF Servlet</display-name>
   <servlet-class>
      org.apache.cxf.transport.servlet.CXFServlet
   </servlet-class>
   <init-param>
      <param-name>redirects-list</param-name>
      <param-value>
         /
         /index.html
      </param-value>
   </init-param>
   <init-param>
       <param-name>redirect-attributes</param-name>
       <param-value>
         javax.servlet.include.request_uri
       </param-value>
   </init-param>
   <init-param>
       <param-name>redirect-servlet-name</param-name>
       <param-value>default</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>CXFServlet</servlet-name>
   <url-pattern>/*</url-pattern>
</servlet-mapping>

<welcome-file-list>
   <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

Note that the redirects-list parameter has two space separated values, "/" and "index.html". The request attribute 'javax.servlet.include.request_uri' might need to be set for the underlying container like Jetty to successfully read "index.html".

Option2. Using CXFServlet itself to read index.html

```
<servlet>
   <servlet-name>CXFServlet</servlet-name>
   <display-name>CXF Servlet</display-name>
   <servlet-class>
      org.apache.cxf.transport.servlet.CXFServlet
   </servlet-class>
   <init-param>
      <param-name>static-welcome-file</param-name>
      <param-value>/index.html</param-value>
   </init-param>
   <init-param>
      <param-name>static-resources-list</param-name>
      <param-value>/index.html</param-value>
   </init-param>
   <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
   <servlet-name>CXFServlet</servlet-name>
   <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Publishing an endpoint with the API

Once your Servlet is registered in your web.xml, you should set the default bus with CXFServlet's bus to make sure that CXF uses it as its HTTP Transport. Simply publish with the related path "Greeter" and your service should appear at the address you specify:

```
import javax.xml.ws.Endpoint;
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.servlet.CXFServlet;
.....
// cxf is the instance of the CXFServlet, you could also get
// this instance by extending the CXFServlet
Bus bus = cxf.getBus();
BusFactory.setDefaultBus(bus);
Endpoint.publish("/Greeter", new GreeterImpl());
```

The one thing you must ensure is that your CXFServlet is set up to listen on that path. Otherwise the CXFServlet will never receive the requests.

NOTE:

Endpoint.publish(...) is a JAX-WS API for publishing JAX-WS endpoints. Thus, it would require the JAX-WS module and APIs to be present. If you are not using JAX-WS or want more control over the published endpoint properties, you should replace that call with the proper calls to the appropriate ServerFactory.

Since CXFServlet know nothing about the web container listening port and the application context path, you need to specify the relative path instead of the full http address.

Using the servlet transport without Spring

A user who doesn't want to touch any Spring stuff could also publish the endpoint with CXF servlet transport. First you should extend the CXFNonSpringServlet and then override the method loadBus, e.g.:

```
import javax.xml.ws.Endpoint;
...

    @Override
    public void loadBus(ServletConfig servletConfig) throws ServletException {
        super.loadBus(servletConfig);

        // You could add the endpoint publish codes here
        Bus bus = cxf.getBus();
        BusFactory.setDefaultBus(bus);
        Endpoint.publish("/Greeter", new GreeterImpl());

        // You can als use the simple frontend API to do this
        ServerFactoryBean factroy = new ServerFactoryBean();
        factory.setBus(bus);
        factory.setServiceClass(GreeterImpl.class);
        factory.setAddress("/Greeter");
        factory.create();
    }
```

If you are using the Jetty as the embedded servlet engine, you could publish endpoint like this:

```
import javax.xml.ws.Endpoint;
...

        // Setup the system properties to use the CXFBusFactory not the SpringBusFactory
        String busFactory =
            System.getProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME);
        System.setProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME,
            "org.apache.cxf.bus.CXFBusFactory");
        try {
            // Start up the jetty embedded server
            httpServer = new Server(9000);
            ContextHandlerCollection contexts = new ContextHandlerCollection();
            httpServer.setHandler(contexts);

            Context root = new Context(contexts, "/", Context.SESSIONS);

            CXFNonSpringServlet cxf = new CXFNonSpringServlet();
            ServletHolder servlet = new ServletHolder(cxf);
            servlet.setName("soap");
            servlet.setForcedPath("soap");
            root.addServlet(servlet, "/soap/*");

            httpServer.start();

            Bus bus = cxf.getBus();
            setBus(bus);
            BusFactory.setDefaultBus(bus);
            GreeterImpl impl = new GreeterImpl();
            Endpoint.publish("/Greeter", impl);
        } catch (Exception e) {
            throw new RuntimeException(e);
        } finally {
            // clean up the system properties
            if (busFactory != null) {
                System.setProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME,
                   busFactory);
            } else {
                System.clearProperty(BusFactory.BUS_FACTORY_PROPERTY_NAME);
            }
        }
```

Accessing the MessageContext and/or HTTP Request and Response

Sometimes you'll want to access more specific message details in your service implementation. One example might be accessing the actual request or response object itself. This can be done using the WebServiceContext object.

First, declare a private field for the WebServiceContext in your service implementation, and annotate it as a resource:

```
@Resource
private WebServiceContext context;
```

Then, within your implementing methods, you can access the MessageContext, HttpServletRequest, and HttpServletResponse as follows:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.ws.handler.MessageContext;
import org.apache.cxf.transport.http.AbstractHTTPDestination;
...

MessageContext ctx = context.getMessageContext();
HttpServletRequest request = (HttpServletRequest)
    ctx.get(AbstractHTTPDestination.HTTP_REQUEST);
HttpServletResponse response = (HttpServletResponse)
    ctx.get(AbstractHTTPDestination.HTTP_RESPONSE);
```

Of course, it is always a good idea to program defensively if using transport-specific entities like the HttpServletRequest and HttpServletResponse. If the transport were changed (for instance to the JMS transport), then these values would likely be null.