pro!vision

.adaptTo() 2011

15.09.2011

Apache CXF and Sling

Consuming and Publishing SOAP Services

About Me

Stefan Seifert

Founder & CTO of pro!vision GmbH

pro!vision GmbH

Spezialized Solution Provider for CMS and Web Projects
International Day CQ Projects since 2003
Customers in Automotive and Telecommunication
Adobe Premium Partner

Located in Berlin & Frankfurt am Main

Overview

- Goal: JAX-WS Support in Apache Sling
- Apache CXF Overview
- CXF and OSGi
- ClassLoader Issues & Solutions
- Sample Application
- JAX-WS SOAP Client
- JAX-WS SOAP Server

JAX-WS Support in Apache Sling

- Goals
  - Consume SOAP Webservices from Sling Applications
  - Publish SOAP Webservices in Apache Sling
    - e.g. for delivering Content from JCR
  - Generate Proxy-Classes from WSDL
  - Use JAX-WS with JAXB Annotations
  - Support for latest WS-* Standards
    - e.g. WS-Addressing

JAX-WS Support in Apache Sling

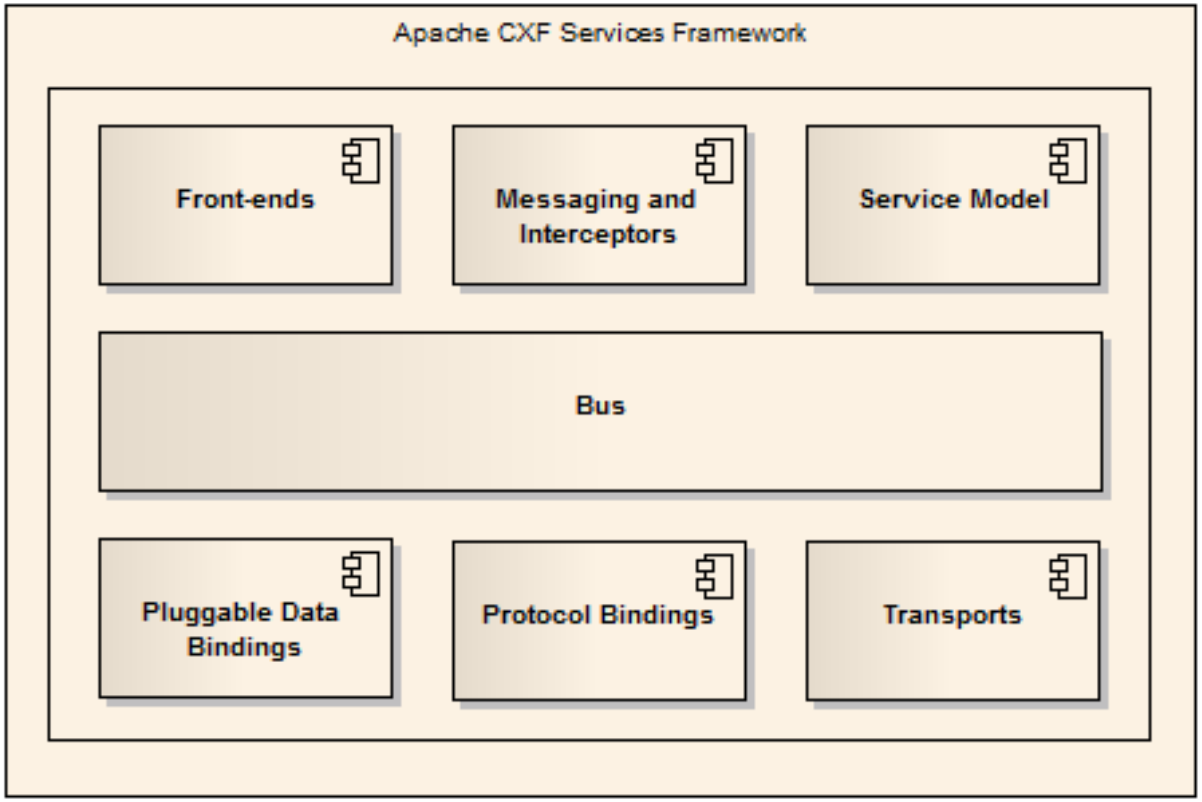
- JAX-WS vs. JAX-RS
  - In this talk we focus on JAX-WS (for SOAP Webservices), not JAX-RS (for RESTful Services)
  - Apache Sling has it's own RESTful concepts
  - See SLING-2192 for an Example of integrating JAX-RS Support in Apache Sling using Apache Wink
  - Basically the integration should work for JAX-RS as well

Apache CXF Overview

- **Apache CXF:**
  An Open-Source Services Framework
  - Implements APIs like JAX-WS and JAX-RS
  - Speaks a variety of protocols such as SOAP, XML/HTTP, RESTful HTTP vie HTTP, JMS etc.
  - Supports various WS-* web service specs
  - Flexible and highly configurable
  - Optimized for integration with Spring (but we will use it standalone)

Apache CXF Overview

- Overall CXF architecture



Apache CXF Overview

- **Depends internally on XML APIs and 3rdparty Implementations**

Available in JDK 1.6

- javax.jws.soap
- javax.xml.bind
- javax.xml.soap
- javax.xml.ws
- javax.activation
- ....

Others

- com.ctc.wstx
- org.codehaus.stax2
- com.ibm.wsdl
- org.apache.xml.resolver
- org.apache.ws.commons.schema
- ...

- **Goal: Supply CXF with all required dependencies internally, but export only org.apache.cxf.***

CXF and OSGi

- CXF provides prepacked OSGI bundle JARs:
  - **cxf-bundle-minimal**
    - Contains minimal set of CXF JARs in one single bundle
    - Does not contain 3rdparty Dependencies/XML APIs
  - **cxf-dosgi-ri-singlebundle-distribution**
    - Contains minimal set of CXF JARs in one single bundle
    - Contains all 3rdparty Dependencies and XML APIs
    - Contains a lot of additional JARs like Spring, Jetty, PAX, cxf-dosgi, slf4j, Commons Logging, log4j, jdom etc.
    - Reference Implementation of Distribution Provider component of the OSGi Remote Services Specification

CXF and OSGi

- Both approaches have drawbacks
  - **cxf-bundle-minimal**
    - Dozens of additional 3rdparty Bundles needed
  - **cxf-dosgi-ri-singlebundle-distribution**
    - Works, but is much more than we wanted
    - Contains dependencies that are exported already from system bundle
    - "Feels unstable" in Sling Environment e.g. when starting/stopping bundle

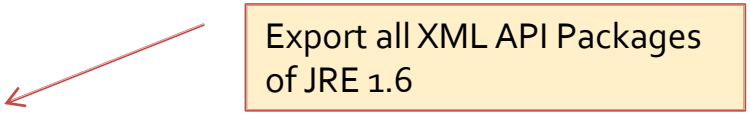
CXF and OSGi

- Solution
  - **System Bundle Extension for XML APIs included in JRE 1.6**
    - See also discussion in SLING-2103
    - *Sample project: framework-extension-xml-jre-1.6*
  - **Handcrafted CXF Bundle**
    - Repackages required CXF JARs including 3rdparty References needed for JAX-WS
    - References XML APIs from System Bundle
    - Keeps single JARs in Bundle to avoid problems with CXF Bus configuration (overlapping META-INF/cxf resources)
    - *Sample project: cxfbundle*
  - **With JRE 1.5**: Use cxf-dosgi-ri-singlebundle-distribution

- System Bundle Extension XML APIs JRE 1.6

framework-extension-xml-jre-1.6/pom.xml

```
<project>
    <artifactId>org.apache.sling.fragment.xml-jre-1.6</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>Apache Sling System Bundle Extension: XML APIs for JRE 1.6</name>
...
            <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <forceCreation>true</forceCreation>
                    <archive>
                        <manifestFile>
                            ${project.build.outputDirectory}/META-INF/MANIFEST.MF
                        </manifestFile>
                        <manifestEntries>
                            <Export-Package>
                                javax.xml,
                                javax.xml.bind,
                                javax.xml.bind.annotation,
                                javax.xml.bind.annotation.adapters,
                                javax.xml.bind.attachment,
                                javax.xml.bind.helpers,
                                javax.xml.bind.util,
                                ...
                            </Export-Package>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
...
</project>
```



see also http://blog.meschberger.ch/2008/10/osgi-framework-extension-as-maven.html

Handcrafted CXF Bundle

cxfbundle/pom.xml

```
<project>
  <artifactId>adaptto.slingcxf.cxfbundle</artifactId>
  <packaging>bundle</packaging>
  <version>1.0.0-SNAPSHOT</version>
  <name>.adaptTo() CXF Bundle</name>
...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>

            <!-- bundle supplied resource prefixes -->
            <Include-Resource>{maven-resources}</Include-Resource>

            <!-- Do not inline jars, include as jar files -->
            <Embed-Dependency>*;scope=compile|runtime;inline=false</Embed-Dependency>
            <Embed-Transitive>true</Embed-Transitive>

            <!-- use _exportcontents instead of Export-Package to avoid conflict with Embed-Dependency an inline=true -->
            <_exportcontents>
              org.apache.cxf.*;version=${project.version}
            </_exportcontents>

            <!-- declare optional dependencies -->
            <Import-Package>
              com.sun.msv.*;resolution:=optional,
              com.sun.xml.bind.marshaller.*;resolution:=optional,
              com.sun.xml.fastinfoset.stax.*;resolution:=optional,
              ...
              *
            </Import-Package>

          </instructions>
        </configuration>
      </plugin>
...
</project>
```

Include all dependencies (except those excluded by <exclusion> Elements in the <dependencies> Section) as Inline JARs

Export only Apache CXF API

Ignore some imports to external 3rdparty Libraries not required for our usecases

but see also http://blog.meschberger.ch/2011/04/to-embed-or-to-inline.html

ClassLoader Issues & Solutions

- We've got our OSGI Bundles, let's use CXF
- But every call fails with errors like:

```
Caused by: javax.xml.bind.JAXBException
        at javax.xml.bind.ContextFinder.newInstance(Unknown Source)
        at javax.xml.bind.ContextFinder.find(Unknown Source)
        at javax.xml.bind.JAXBContext.newInstance(Unknown Source)
        at org.apache.cxf.jaxb.JAXBContextCache.createContext(JAXBContextCache.java:258)
        at org.apache.cxf.jaxb.JAXBContextCache.getCachedContextAndSchemas(JAXBContextCache.java:167)
        at org.apache.cxf.jaxb.JAXBDataBinding.createJAXBContextAndSchemas(JAXBDataBinding.java:418)
        at org.apache.cxf.jaxb.JAXBDataBinding.initialize(JAXBDataBinding.java:290)
        ... 75 more
Caused by: java.lang.ClassNotFoundException: com.sun.xml.internal.bind.v2.ContextFactory
        at
      org.apache.sling.commons.classloader.impl.ClassLoaderFacade.loadClass(ClassLoaderFacade.java:127)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at javax.xml.bind.ContextFinder.safeLoadClass(Unknown Source)
        ... 82 more
```

- (Stacktraces and classes not found vary)

ClassLoader Issues & Solutions

- Reason: Illegal class loading in OSGI context
  - Referenced library calls currentThread.getContextClassLoader().loadClass
  - In this case it's not CXF, but the JAXB implementation
  - We cannot fix the 3rdparty Implementation

- Solution
  - See CQ5 KB Article: OsgiClassLoading3Party.html

```
ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
Thread.currentThread().setContextClassLoader(FrameworkXYZ.class.getClassLoader());
try {
  // execute 3rd party code
} finally {
  Thread.currentThread().setContextClassLoader(oldClassLoader);
}
```

Reference class from 3rdparty Bundle which is executed

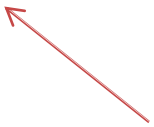
Sample Application

- Directory structure
  - framework-extension-xml-jre-1.6
    - System bundle extension for XML APIs in JRE 1.6
  - cxfbundle
    - Handcrafted OSGI bundle for CXF incl. dependencies
  - helloworld-proxy
    - Auto-generated Proxy classes for Webservice with JAX-WS/JAXB Annotations (from WSDL)
  - helloworld-application
    - Sample Sling application with SOAP Client and Server
  - See README.txt

Sample Application

- Generate Proxy classes from WSDL
  - Using Apache CXF CodeGen plugin

```
<plugin>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-codegen-plugin</artifactId>
  <version>${apache-cxf-version}</version>
  <executions>
    <execution>
      <id>generate-sources</id>
      <phase>generate-sources</phase>
      <configuration>
        <sourceRoot>${project.build.directory}/generated/cxf</sourceRoot>
        <wsdlRoot>${basedir}/src/main/wsdl</wsdlRoot>
        <wsdlOptions>
          <wsdlOption>
            <wsdl>${basedir}/src/main/wsdl/helloworld.wsdl</wsdl>
          </wsdlOption>
        </wsdlOptions>
      </configuration>
      <goals>
        <goal>wsdl2java</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



This is our „hello world“ Sample WSDL

Sample Application

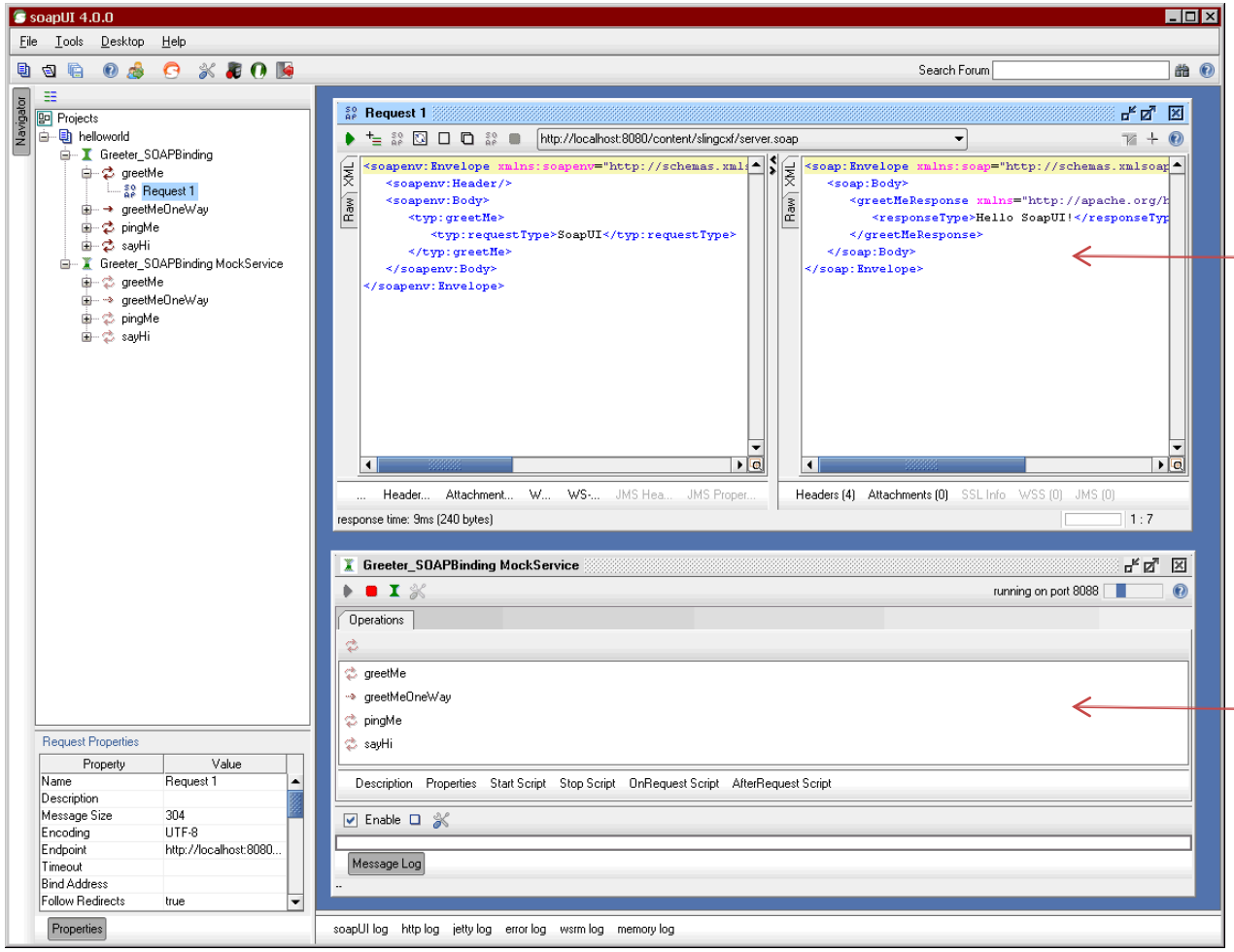
- Interface of Sample Webservice
  - with JAXB/JAX-WS Annotations

```
@WebService(targetNamespace = "http://apache.org/hello_world_soap_http", name = "Greeter")
@XmlSeeAlso({org.apache.hello_world_soap_http.types.ObjectFactory.class})
public interface Greeter {

    @WebResult(name = "responseType", targetNamespace =
       "http://apache.org/hello_world_soap_http/types")
    @RequestWrapper(localName = "greetMe", targetNamespace =
       "http://apache.org/hello_world_soap_http/types", className =
       "org.apache.hello_world_soap_http.types.GreetMe")
    @WebMethod
    @ResponseWrapper(localName = "greetMeResponse", targetNamespace =
       "http://apache.org/hello_world_soap_http/types", className =
       "org.apache.hello_world_soap_http.types.GreetMeResponse")
    public java.lang.String greetMe(
        @WebParam(name = "requestType", targetNamespace =
       "http://apache.org/hello_world_soap_http/types")
        java.lang.String requestType
    );

...

}
```

We will use this method for our sample usecases

Sample Application

- Testing against SoapUI



Download from:
http://www.soapui.org/

Test SOAP Call to SOAP Server in Apache Sling

SOAP Mock Service to Test SOAP Call from Apache Sling

JAX-WS SOAP Client

- Goals
  - Do not clutter application code with ClassLoader handling code
  - Create generic factory methods to get read-to-use proxy objects
  - Ensure correct ClassLoader handling on Proxy initialization and on invoke of a SOAP client method

JAX-WS SOAP Client

- Solution
  - Create Factory class for Client Initiliaztion
  - Create subclasses of JaxWsClientFactoryBean and ClientImpl to ensure correct ClassLoader handling on SOAP method invoke
  - Enforce usage in Factory class
- Execute SOAP client in sample project

JAX-WS SOAP Client

- ## JaxWS Client Factory

helloworld-application/src/main/java/adaptto/slingcxf/client/util/JaxWsClientFactory.java

```
/**
 * Create webservice port via JAXWS proxy factory.
 *
 * This method fixes numerous problems with 3rdparty libs used by CXF and
 * CXF itself and classloader issues with OSGI. Using this method the
 * initialization phase of JAXB mapping is wrapped in an OSGI-aware
 * classloader. Furthermore each client instances is wrapped in an
 * OSGI-aware subclass (see {@link OsgiAwareClientImpl}), which ensures that
 * each invoke call on a webservice method is itself executed within an
 * OSGI-aware classloader context.
 *
 * @param <T> Port class
 * @param pClass Port class
 * @param pPortUrl Port url (this is not the WSDL location)
 * @param pUsername Username for port authentication
 * @param pPassword Password for port authentication
 * @return Port object
 */
@SuppressWarnings("unchecked")
public static <T> T create(Class<T> pClass, String pPortUrl) {
    ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
    try {
        // set classloader to CXF bundle class loader to avoid OSGI classloader problems
        Thread.currentThread().setContextClassLoader(BusFactory.class.getClassLoader());

        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean(new OsgiAwareClientFactoryBean());
        factory.setServiceClass(pClass);
        factory.setAddress(pPortUrl);
        return (T) factory.create();
    }
    finally {
        Thread.currentThread().setContextClassLoader(oldClassLoader);
    }
}
```

Workaround for Context ClassLoader handling

Inject custom „OSGi-aware“ implementation of factory bean

JAX-WS SOAP Client

- OsgiAwareClientFactoryBean

helloworld-application/src/main/java/adaptto/slingcxf/client/util/OsgiAwareClientFactoryBean.java

```
/**
 * Factory bean that creates {@link OsgiAwareClientImpl} objects instead of
 * {@link org.apache.cxf.endpoint.ClientImpl} objects to ensure correct
 * classloader usage in OSGI context.
 */
class OsgiAwareClientFactoryBean extends org.apache.cxf.jaxws.JaxWsClientFactoryBean {

    @Override
    protected Client createClient(Endpoint ep) {
        // use osgi-aware client impl instead of {@link
        // org.apache.cxf.endpoint.ClientImpl}
        return new OsgiAwareClientImpl(getBus(), ep, getConduitSelector());
    }

}
```

Inject custom „OSGi-aware" implementation of client implementation

JAX-WS SOAP Client

- ## OsgiAwareClientImpl

helloworld-application/src/main/java/adaptto/slingcxf/client/util/OsgiAwareClientImpl.java

```
/**
 * Enhances {@link ClientImpl} by ensuring that each webservice method invoke is
 * called within the context of an OSGI aware classloader.
 */
class OsgiAwareClientImpl extends org.apache.cxf.endpoint.ClientImpl {

    public OsgiAwareClientImpl(Bus pB, Endpoint pE, Conduit pC) {
        super(pB, pE, pC);
    }

...

    @Override
    public Object[] invoke(BindingOperationInfo pOi, Object[] pParams, Map<String, Object> pContext, Exchange pExchange)
            throws Exception {
        ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
        try {
            // set classloader to CXF bundle class loader to avoid OSGI classloader problems
            Thread.currentThread().setContextClassLoader(BusFactory.class.getClassLoader());

            return super.invoke(pOi, pParams, pContext, pExchange);
        } finally {
            Thread.currentThread().setContextClassLoader(oldClassLoader);
        }
    }

}
```

Workaround for Context ClassLoader handling

JAX-WS SOAP Client

- GreeterClientService
  - Allow Configuration of Webservice URL via OSGi config

helloworld-application/src/main/java/adaptto/slingcxf/client/GreeterClientService.java

```
/**
 * Service for configuring SOAP proxy client instance
 */
@Component(metatype=true, label="slingcxf Greeter Client Service")
@Service(GreeterClientService.class)
public class GreeterClientService {

    @Property(label="Greeter Webservice URL", value=GreeterClientService.DEFAULT_PORTURL)
    private static final String PROPERTY_PORTURL = "portUrl";
    private static final String DEFAULT_PORTURL = "http://localhost:8088/mockGreeter_SOAPBinding";

    private Greeter greeterInstance;
    private final Logger log = LoggerFactory.getLogger(getClass());

    @Activate
    protected final void activate(final Map<String, String> config) {
        try {
            // get port url from OSGI config
            String portUrl = PropertiesUtil.toString(config.get(PROPERTY_PORTURL), DEFAULT_PORTURL);
            // instantiate greeter SOAP client proxy
            this.greeterInstance = JaxWsClientFactory.create(Greeter.class, portUrl);
        }
        catch (RuntimeException ex) {
            log.error("Unable to instanciate SOAP service proxy client.", ex);
        }
    }

    /**
     * @return Greeter SOAP client proxy instance
     */
    public Greeter getInstance() {
        return this.greeterInstance;
    }

}
```

Create JAX-WS SOAP Client using generated Proxy Interface and our Factory method

JAX-WS SOAP Client

- Client Sample JSP
  - Executes method of SOAP webservice

helloworld-application/src/main/webapp/app-root/components/client/html.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN""http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<%@page import="org.apache.hello_world_soap_http.Greeter"%>
<%@page import="adaptto.slingcxf.client.GreeterClientService"%>
<%@taglib prefix="sling" uri="http://sling.apache.org/taglibs/sling/1.0"%>
<sling:defineObjects/>
<html>
  <head>
    <title>.adaptTo() Sling CXF Integration - SOAP Client</title>
  </head>
  <body>

...

    <h2>Response from Webservice</h2>
<%
        String greetName = StringUtils.defaultString(request.getParameter("greetName"), "Apache Sling");
        GreeterClientService greeterClientService = sling.getService(GreeterClientService.class);
        Greeter greeter = greeterClientService.getInstance();
%>
        <p>Greet me: <strong><%=greeter.greetMe(greetName)%></strong></p>

  </body>

</html>
```

Execute SOAP Service method

JAX-WS SOAP Server

- Goals
  - Do not clutter application code with ClassLoader handling code
  - Transparent mapping of SOAP Action URLs in Sling Component URL name schema
  - Make use of Sling Security for Authentication
  - Provide Access to Sling Context Objects (Resource Resolver, JCR Session etc.)

JAX-WS SOAP Server

- Solution
  - Ensure SOAP Server can be initialized without Spring Context
  - Create subclass of CXFNonSpringServlet to ensure correct ClassLoader handling
  - Register as Sling Servlet and map it to a resource type
  - Map to virtual path using a RequestWrapper
  - Provide access to request context using ThreadLocal
- Execute SOAP server in sample project

JAX-WS SOAP Server

AbstractJaxWsServer part I

helloworld-application/src/main/java/adaptto/slingcxf/server/util/AbstractJaxWsServer.java

```
/**
 * Abstract servlet-based implementation for CXF-based SOAP services. Ensures
 * that correct class loader is used is during initialization and invoking
 * phases. Via getCurrentRequest() and getCurrentResponse() it is possible to
 * access these objects from SOAP method implementations.
 */
public abstract class AbstractJaxWsServer extends org.apache.cxf.transport.servlet.CXFNonSpringServlet {

    @Override
    public void init(ServletConfig pServletConfig) throws ServletException {
        ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
        try {
            // set classloader to CXF bundle class loader to avoid OSGI classloader problems
            Thread.currentThread().setContextClassLoader(BusFactory.class.getClassLoader());

            super.init(pServletConfig);

            // register SOAP service
            ServerFactoryBean factory = new JaxWsServerFactoryBean();
            factory.setBus(getBus());
            factory.setAddress(RequestWrapper.VIRTUAL_PATH);
            factory.setServiceClass(getServerInterfaceType());
            factory.setServiceBean(this);
            factory.create();

        } finally {
            Thread.currentThread().setContextClassLoader(oldClassLoader);
        }
    }

    /**
     * @return Interface of SOAP service
     */
    protected abstract Class getServerInterfaceType();

...
```

Workaround for Context ClassLoader handling

Map to virtual URI path (from RequestWrapper)

JAX-WS Interface class is specified by subclass

The subclass itself is the implementation of the JAX-WS SOAP Server

JAX-WS SOAP Server

- ## AbstractJaxWsServer part II

helloworld-application/src/main/java/adaptto/slingcxf/server/util/AbstractJaxWsServer.java

```
...
    @Override
    protected void invoke(HttpServletRequest pRequest, HttpServletResponse pResponse) throws ServletException {
        RequestContext.getThreadLocal().set(new RequestContext(pRequest, pResponse));
        ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
        try {
            // set classloader to CXF bundle class loader to avoid OSGI classloader problems
            Thread.currentThread().setContextClassLoader(BusFactory.class.getClassLoader());

            super.invoke(new RequestWrapper(pRequest), pResponse);
        } finally {
            Thread.currentThread().setContextClassLoader(oldClassLoader);
            RequestContext.getThreadLocal().remove();
        }
    }

    /**
     * @return Servlet request for current threads SOAP request
     */
    protected SlingHttpServletRequest getCurrentRequest() {
        RequestContext requestContext = RequestContext.getRequestContext();
        if (requestContext == null) {
            throw new IllegalStateException("No current soap request context available.");
        }
        return (SlingHttpServletRequest)requestContext.getRequest();
    }

    /**
     * @return Servlet response for current threads SOAP request
     */
    protected SlingHttpServletResponse getCurrentResponse() {
        RequestContext requestContext = RequestContext.getRequestContext();
        if (requestContext == null) {
            throw new IllegalStateException("No current soap request context available.");
        }
        return (SlingHttpServletResponse)requestContext.getResponse();
    }
}
```

Put Request/Response in ThreadLocal

Workaround for Context ClassLoader handling

Inject custom RequestWrapper to simulate virtual path to which the SOAP server is registered in CXF

JAX-WS SOAP Server

- ## RequestWrapper

helloworld-application/src/main/java/adaptto/slingcxf/server/util/RequestWrapper.java

```
/**
 * Request wrapper that maps all pathinfo to a virtual path, to whom the SOAP
 * services are registered to.
 */
class RequestWrapper extends javax.servlet.http.HttpServletRequestWrapper {

    public static final String VIRTUAL_PATH = "/soaprequest";

    public RequestWrapper(HttpServletRequest pRequest) {
        super(pRequest);
    }

    @Override
    public String getPathInfo() {
        return VIRTUAL_PATH;
    }

}
```

Define virtual path for request handling inside CXF

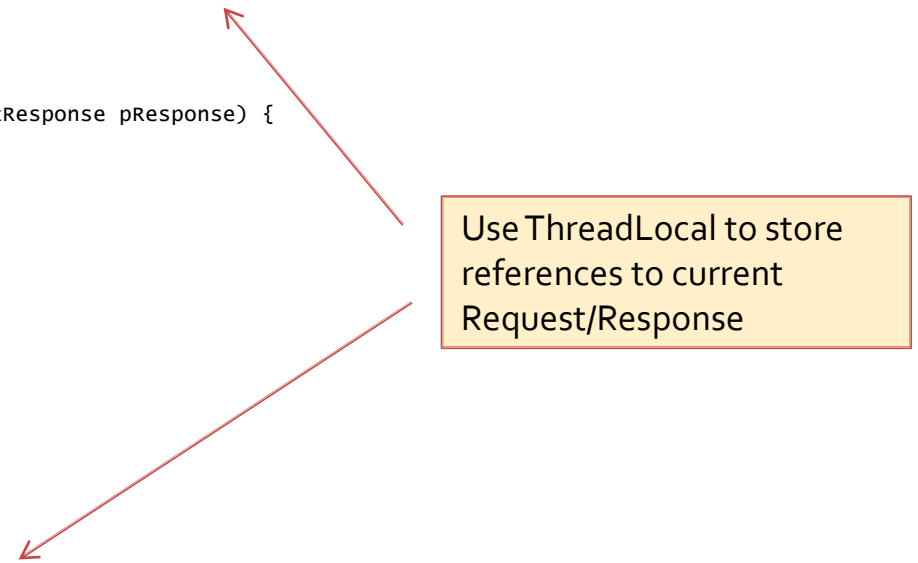
Override „getPathInfo()“ in RequestWrapper to simulate virtual path for CXF address mapping implementation

JAX-WS SOAP Server

- RequestContext

helloworld-application/src/main/java/adaptto/slingcxf/server/util/RequestContext.java

```
/**
 * Manages request/response context of incoming SOAP server HTTP requests using
 * a ThreadLocal.
 */
class RequestContext {

    private static ThreadLocal<RequestContext> mThreadLocal = new ThreadLocal<RequestContext>();

    private final HttpServletRequest mRequest;
    private final HttpServletResponse mResponse;

    RequestContext(HttpServletRequest pRequest, HttpServletResponse pResponse) {
        mRequest = pRequest;
        mResponse = pResponse;
    }

    public HttpServletRequest getRequest() {
        return mRequest;
    }

    public HttpServletResponse getResponse() {
        return mResponse;
    }

    /**
     * @return Context for current SOAP server request
     */
    public static RequestContext getRequestContext() {
        return mThreadLocal.get();
    }

    static ThreadLocal<RequestContext> getThreadLocal() {
        return mThreadLocal;
    }

}
```



JAX-WS SOAP Server

- ## Example SOAP Server implementation

helloworld-application/src/main/java/adaptto/slingcxf/server/GreeterServer.java

```
/**
 * Implementation for {@link Greeter} SOAP service.
 */
@SlingServlet(
    resourceTypes = "/apps/slingcxf/components/server",
    extensions = AbstractJaxWsServer.SOAP_EXTENSION,
    methods = { "GET", "POST" }
)
public class GreeterServer extends AbstractJaxWsServer implements Greeter {
    private static final long serialVersionUID = 1L;

    protected Class getServerInterfaceType() {
        return Greeter.class;
    }

    public void pingMe() throws PingMeFault {
        // nothing to do
    }

    public String sayHi() {
        return "Hi!";
    }

    public void greetMeOneWay(String requestType) {
        // nothing to do
    }

    public String greetMe(String name) {
        // get pattern for greeting response from JCR
        Resource resource = getCurrentRequest().getResource();
        ValueMap properties = resource.adaptTo(ValueMap.class);
        String greetResponsePattern = properties.get("greetResponsePattern", "Hello ${name}!");

        // replace placeholder with the client's name
        return StringUtils.replace(greetResponsePattern, "${name}", name);
    }

}
```

Register JAX-WS SOAP Server implementation as Sling Servlet to a resource type

Define which JAX-WS interface is implemented by this server

Sample implementation of „greetMe“ method

Q & A

Contact Information

pro!vision GmbH

Karim Khan
kkhan@pro-vision.de
Tel. +49 (69) 8700328-10

Office Berlin
pro!vision GmbH
Wilmersdorfer Str. 50/51
10627 Berlin
Germany
Tel. +49 (30) 818828-50

Office Frankfurt
pro!vision GmbH
Löwengasse 27 A
60385 Frankfurt am Main
Germany
Tel. +49 (69) 8700328-0