Tutorial-AXIS-Camel

Tutorial using Axis 1.4 with Apache Camel

Removed from distribution

This example has been removed from **Camel 2.9** onwards. Apache Axis 1.4 is a very old and unsupported framework. We encourage users to use CXF instead of Axis.

- Tutorial using Axis 1.4 with Apache Camel
  - Prerequisites
  - Distribution
  - Introduction
  - Setting up the project to run Axis
    - Maven 2
    - wsdl
    - Configuring Axis
    - Running the Example
  - Integrating Spring
    - Using Spring
  - Integrating Camel
    - CamelContext
    - Store a file backup
  - Running the example
  - Unit Testing
    - Smarter Unit Testing with Spring
  - Unit Test calling WebService
  - Annotations
  - The End
  - See Also

Prerequisites

This tutorial uses Maven 2 to setup the Camel project and for dependencies for artifacts.

Distribution

This sample is distributed with the Camel 1.5 distribution as `examples/camel-example-axis`.

Introduction

Apache Axis is/was widely used as a webservice framework. So in line with some of the other tutorials to demonstrate how Camel is not an invasive framework but is flexible and integrates well with existing solution.

We have an existing solution that exposes a webservice using Axis 1.4 deployed as web applications. This is a common solution. We use contract first so we have Axis generated source code from an existing wsdl file. Then we show how we introduce Spring and Camel to integrate with Axis.

This tutorial uses the following frameworks:

- Maven 2.0.9
- Apache Camel 1.5.0
- Apache Axis 1.4
- Spring 2.5.5

Setting up the project to run Axis

This first part is about getting the project up to speed with Axis. We are not touching Camel or Spring at this time.

Maven 2

Axis dependencies is available for maven 2 so we configure our pom.xml as:

```
<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis-jaxrpc</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>org.apache.axis</groupId>
    <artifactId>axis-saaj</artifactId>
    <version>1.4</version>
</dependency>

<dependency>
    <groupId>axis</groupId>
    <artifactId>axis-wsdl4j</artifactId>
    <version>1.5.1</version>
</dependency>

<dependency>
    <groupId>commons-discovery</groupId>
    <artifactId>commons-discovery</artifactId>
    <version>0.4</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
</dependency>
```

Then we need to configure maven to use Java 1.5 and the Axis maven plugin that generates the source code based on the wsdl file:

```
<!-- to compile with 1.5 -->
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                        <source>1.5</source>
                        <target>1.5</target>
                </configuration>
        </plugin>

            <plugin>
               <groupId>org.codehaus.mojo</groupId>
               <artifactId>axistools-maven-plugin</artifactId>
               <configuration>
                  <sourceDirectory>src/main/resources/</sourceDirectory>
                  <packageSpace>com.mycompany.myschema</packageSpace>
                  <testCases>false</testCases>
                  <serverSide>true</serverSide>
                  <subPackageByFileName>false</subPackageByFileName>
               </configuration>
               <executions>
                 <execution>
                   <goals>
                     <goal>wsdl2java</goal>
                   </goals>
                 </execution>
               </executions>
            </plugin>
```

wsdl

We use the same .wsdl file as the Tutorial-Example-ReportIncident and copy it to `src/main/webapp/WEB-INF/wsdl`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns:tns="http://reportincident.example.camel.apache.org"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
        xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
        targetNamespace="http://reportincident.example.camel.apache.org">

        <!-- Type definitions for input- and output parameters for webservice -->
        <wsdl:types>
        <xs:schema targetNamespace="http://reportincident.example.camel.apache.org">
                        <xs:element name="inputReportIncident">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element type="xs:string"  name="incidentId"/>
                                                <xs:element type="xs:string"  name="incidentDate"/>
                                                <xs:element type="xs:string"  name="givenName"/>
                                                <xs:element type="xs:string"  name="familyName"/>
                                                <xs:element type="xs:string"  name="summary"/>
                                                <xs:element type="xs:string"  name="details"/>
                                                <xs:element type="xs:string"  name="email"/>
                                                <xs:element type="xs:string"  name="phone"/>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                        <xs:element name="outputReportIncident">
                                <xs:complexType>
                                        <xs:sequence>
                                                <xs:element type="xs:string" name="code"/>
                                        </xs:sequence>
                                </xs:complexType>
                        </xs:element>
                </xs:schema>
```

```
        </wsdl:types>

        <!-- Message definitions for input and output -->
        <wsdl:message name="inputReportIncident">
                <wsdl:part name="parameters" element="tns:inputReportIncident"/>
        </wsdl:message>
        <wsdl:message name="outputReportIncident">
                <wsdl:part name="parameters" element="tns:outputReportIncident"/>
        </wsdl:message>

        <!-- Port (interface) definitions -->
        <wsdl:portType name="ReportIncidentEndpoint">
                <wsdl:operation name="ReportIncident">
                        <wsdl:input message="tns:inputReportIncident"/>
                        <wsdl:output message="tns:outputReportIncident"/>
                </wsdl:operation>
        </wsdl:portType>

        <!-- Port bindings to transports and encoding - HTTP, document literal encoding is used -->
        <wsdl:binding name="ReportIncidentBinding" type="tns:ReportIncidentEndpoint">
                <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
                <wsdl:operation name="ReportIncident">
                        <soap:operation
                                soapAction="http://reportincident.example.camel.apache.org/ReportIncident"
                                style="document"/>
                        <wsdl:input>
                                <soap:body parts="parameters" use="literal"/>
                        </wsdl:input>
                        <wsdl:output>
                                <soap:body parts="parameters" use="literal"/>
                        </wsdl:output>
                </wsdl:operation>
        </wsdl:binding>

        <!-- Service definition -->
        <wsdl:service name="ReportIncidentService">
                <wsdl:port name="ReportIncidentPort" binding="tns:ReportIncidentBinding">
                        <soap:address location="http://reportincident.example.camel.apache.org"/>
                </wsdl:port>
        </wsdl:service>

</wsdl:definitions>
```

Configuring Axis

Okay we are now setup for the contract first development and can generate the source file. For now we are still only using standard Axis and not Spring nor Camel. We still need to setup Axis as a web application so we configure the web.xml in `src/main/webapp/WEB-INF/web.xml` as:

```
    <servlet>
        <servlet-name>axis</servlet-name>
        <servlet-class>org.apache.axis.transport.http.AxisServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>axis</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
```

The web.xml just registers Axis servlet that is handling the incoming web requests to its servlet mapping. We still need to configure Axis itself and this is done using its special configuration file `server-config.wsdd`. We nearly get this file for free if we let Axis generate the source code so we run the maven goal:

```
mvn axistools:wsdl2java
```

The tool will generate the source code based on the wsdl and save the files to the following folder:

```
.\target\generated-sources\axistools\wsdl2java\org\apache\camel\example\reportincident
deploy.wsdd
InputReportIncident.java
OutputReportIncident.java
ReportIncidentBindingImpl.java
ReportIncidentBindingStub.java
ReportIncidentService_PortType.java
ReportIncidentService_Service.java
ReportIncidentService_ServiceLocator.java
undeploy.wsdd
```

This is standard Axis and so far no Camel or Spring has been touched. To implement our webservice we will add our code, so we create a new class `AxisReportIncidentService` that implements the port type interface where we can implement our code logic what happens when the webservice is invoked.

```
package org.apache.camel.example.axis;

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;
import org.apache.camel.example.reportincident.ReportIncidentService_PortType;

import java.rmi.RemoteException;

/**
 * Axis webservice
 */
public class AxisReportIncidentService implements ReportIncidentService_PortType {

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        System.out.println("Hello AxisReportIncidentService is called from " + parameters.getGivenName());

        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }

}
```

Now we need to configure Axis itself and this is done using its `server-config.wsdd` file. We nearly get this for for free from the auto generated code, we copy the stuff from `deploy.wsdd` and made a few modifications:

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/" xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
>
  <!-- global configuration -->
        <globalConfiguration>
                <parameter name="sendXsiTypes" value="true"/>
                <parameter name="sendMultiRefs" value="true"/>
                <parameter name="sendXMLDeclaration" value="true"/>
                <parameter name="axis.sendMinimizedElements" value="true"/>
        </globalConfiguration>
        <handler name="URLMapper" type="java:org.apache.axis.handlers.http.URLMapper"/>

  <!-- this service is from deploy.wsdd -->
  <service name="ReportIncidentPort" provider="java:RPC" style="document" use="literal">
      <parameter name="wsdlTargetNamespace" value="http://reportincident.example.camel.apache.org"/>
      <parameter name="wsdlServiceElement" value="ReportIncidentService"/>
      <parameter name="schemaUnqualified" value="http://reportincident.example.camel.apache.org"/>
      <parameter name="wsdlServicePort" value="ReportIncidentPort"/>
      <parameter name="className" value="org.apache.camel.example.reportincident.ReportIncidentBindingImpl"/>
      <parameter name="wsdlPortType" value="ReportIncidentService"/>
      <parameter name="typeMappingVersion" value="1.2"/>
      <operation name="reportIncident" qname="ReportIncident" returnQName="retNS:outputReportIncident" xmlns:
retNS="http://reportincident.example.camel.apache.org"
                returnType="rtns:>outputReportIncident" xmlns:rtns="http://reportincident.example.camel.apache.
org"
                soapAction="http://reportincident.example.camel.apache.org/ReportIncident" >
        <parameter qname="pns:inputReportIncident" xmlns:pns="http://reportincident.example.camel.apache.org"
                type="tns:>inputReportIncident" xmlns:tns="http://reportincident.example.camel.apache.org"/>
      </operation>
      <parameter name="allowedMethods" value="reportIncident"/>

      <typeMapping
        xmlns:ns="http://reportincident.example.camel.apache.org"
        qname="ns:>outputReportIncident"
        type="java:org.apache.camel.example.reportincident.OutputReportIncident"
        serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
        deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
        encodingStyle=""
      />
      <typeMapping
        xmlns:ns="http://reportincident.example.camel.apache.org"
        qname="ns:>inputReportIncident"
        type="java:org.apache.camel.example.reportincident.InputReportIncident"
        serializer="org.apache.axis.encoding.ser.BeanSerializerFactory"
        deserializer="org.apache.axis.encoding.ser.BeanDeserializerFactory"
        encodingStyle=""
      />
  </service>

  <!-- part of Axis configuration -->
        <transport name="http">
                <requestFlow>
                        <handler type="URLMapper"/>
                        <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
                </requestFlow>
        </transport>
</deployment>
```

The **globalConfiguration** and **transport** is not in the deploy.wsdd file so you gotta write that yourself. The **service** is a 100% copy from deploy.wsdd. Axis has more configuration to it than shown here, but then you should check the Axis documentation.

What we need to do now is important, as we need to modify the above configuration to use our webservice class than the default one, so we change the classname parameter to our class **AxisReportIncidentService**:

```
<parameter name="className" value="org.apache.camel.example.axis.AxisReportIncidentService"/>
```

Running the Example

Now we are ready to run our example for the first time, so we use Jetty as the quick web container using its maven command:

```
mvn jetty:run
```

Then we can hit the web browser and enter this URL: `http://localhost:8080/camel-example-axis/services` and you should see the famous Axis start page with the text **And now... Some Services**.

Clicking on the .wsdl link shows the wsdl file, but what. It's an auto generated one and not our original .wsdl file. So we need to fix this ASAP and this is done by configuring Axis in the server-config.wsdd file:

```
<service name="ReportIncidentPort" provider="java:RPC" style="document" use="literal">
  <wsdlFile>/WEB-INF/wsdl/report_incident.wsdl</wsdlFile>
  ...
```

We do this by adding the wsdlFile tag in the service element where we can point to the real .wsdl file.

Integrating Spring

First we need to add its dependencies to the **pom.xml**.

```
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-web</artifactId>
          <version>2.5.5</version>
      </dependency>
```

Spring is integrated just as it would like to, we add its listener to the web.xml and a context parameter to be able to configure precisely what spring xml files to use:

```
  <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>
          classpath:axis-example-context.xml
      </param-value>
  </context-param>

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

Next is to add a plain spring XML file named **axis-example-context.xml** in the src/main/resources folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<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-
2.5.xsd">

</beans>
```

The spring XML file is currently empty. We hit jetty again with `mvn jetty:run` just to make sure Spring was setup correctly.

Using Spring

We would like to be able to get hold of the Spring ApplicationContext from our webservice so we can get access to the glory spring, but how do we do this? And our webservice class AxisReportIncidentService is created and managed by Axis we want to let Spring do this. So we have two problems.

We solve these problems by creating a delegate class that Axis creates, and this delegate class gets hold on Spring and then gets our real webservice as a spring bean and invoke the service.

First we create a new class that is 100% independent from Axis and just a plain POJO. This is our real service.

```
package org.apache.camel.example.axis;

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;

/**
 * Our real service that is not tied to Axis
 */
public class ReportIncidentService  {

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        System.out.println("Hello ReportIncidentService is called from " + parameters.getGivenName());

        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }

}
```

So now we need to get from AxisReportIncidentService to this one ReportIncidentService using Spring. Well first of all we add our real service to spring XML configuration file so Spring can handle its lifecycle:

```
<?xml version="1.0" encoding="UTF-8"?>
<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-
2.5.xsd">

    <bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

</beans>
```

And then we need to modify AxisReportIncidentService to use Spring to lookup the spring bean **id="incidentservice"** and delegate the call. We do this by extending the spring class `org.springframework.remoting.jaxrpc.ServletEndpointSupport` so the refactored code is:

```
package org.apache.camel.example.axis;

import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;
import org.apache.camel.example.reportincident.ReportIncidentService_PortType;
import org.springframework.remoting.jaxrpc.ServletEndpointSupport;

import java.rmi.RemoteException;

/**
 * Axis webservice
 */
public class AxisReportIncidentService extends ServletEndpointSupport implements ReportIncidentService_PortType
{

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        // get hold of the spring bean from the application context
        ReportIncidentService service = (ReportIncidentService) getApplicationContext().getBean
("incidentservice");

        // delegate to the real service
        return service.reportIncident(parameters);
    }

}
```

To see if everything is okay we run `mvn jetty:run`.

In the code above we get hold of our service at each request by looking up in the application context. However Spring also supports an **init** method where we can do this once. So we change the code to:

```
public class AxisReportIncidentService extends ServletEndpointSupport implements ReportIncidentService_PortType
{

    private ReportIncidentService service;

    @Override
    protected void onInit() throws ServiceException {
        // get hold of the spring bean from the application context
        service = (ReportIncidentService) getApplicationContext().getBean("incidentservice");
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) throws RemoteException {
        // delegate to the real service
        return service.reportIncident(parameters);
    }

}
```

So now we have integrated Axis with Spring and we are ready for Camel.

Integrating Camel

Again the first step is to add the dependencies to the maven **pom.xml** file:

```
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-core</artifactId>
            <version>1.5.0</version>
        </dependency>

        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-spring</artifactId>
            <version>1.5.0</version>
        </dependency>
```

Now that we have integrated with Spring then we easily integrate with Camel as Camel works well with Spring.

Camel does not require Spring

Camel does not require Spring, we could easily have used Camel without Spring, but most users prefer to use Spring also.

We choose to integrate Camel in the Spring XML file so we add the camel namespace and the schema location:

```
xmlns:camel="http://activemq.apache.org/camel/schema/spring"
http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/camel/schema/spring/camel-spring.xsd"
```

CamelContext

CamelContext is the heart of Camel its where all the routes, endpoints, components, etc. is registered. So we setup a CamelContext and the spring XML files looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://activemq.apache.org/camel/schema/spring"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-
2.5.xsd
         http://activemq.apache.org/camel/schema/spring http://activemq.apache.org/camel/schema/spring/camel-
spring.xsd">

    <bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

    <camel:camelContext id="camel">
        <!-- TODO: Here we can add Camel stuff -->
    </camel:camelContext>

</beans>
```

Store a file backup

We want to store the web service request as a file before we return a response. To do this we want to send the file content as a message to an endpoint that produces the file. So we need to do two steps:

- configure the file backup endpoint
- send the message to the endpoint

The endpoint is configured in spring XML so we just add it as:

```
    <camel:camelContext id="camelContext">
        <!-- endpoint named backup that is configued as a file component -->
        <camel:endpoint id="backup" uri="file://target?append=false"/>
    </camel:camelContext>
```

In the CamelContext we have defined our endpoint with the id `backup` and configured it use the URL notation that we know from the internet. Its a `file` scheme that accepts a context and some options. The contest is `target` and its the folder to store the file. The option is just as the internet with ? and & for subsequent options. We configure it to not append, meaning than any existing file will be overwritten. See the File component for options and how to use the camel file endpoint.

Next up is to be able to send a message to this endpoint. The easiest way is to use a ProducerTemplate. A ProducerTemplate is inspired by Spring template pattern with for instance JmsTemplate or JdbcTemplate in mind. The template that all the grunt work and exposes a simple interface to the end-user where he/she can set the payload to send. Then the template will do proper resource handling and all related issues in that regard. But how do we get hold of such a template? Well the CamelContext is able to provide one. This is done by configuring the template on the camel context in the spring XML as:

```
    <camel:camelContext id="camelContext">
        <!-- producer template exposed with this id -->
        <camel:template id="camelTemplate"/>

        <!-- endpoint named backup that is configued as a file component -->
        <camel:endpoint id="backup" uri="file://target?append=false"/>
    </camel:camelContext>
```

Then we can expose a ProducerTemplate property on our service with a setter in the Java code as:

```
public class ReportIncidentService {

    private ProducerTemplate template;

    public void setTemplate(ProducerTemplate template) {
        this.template = template;
    }
```

And then let Spring handle the dependency inject as below:

```
    <bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService">
        <!-- set the producer template to use from the camel context below -->
        <property name="template" ref="camelTemplate"/>
    </bean>
```

Now we are ready to use the producer template in our service to send the payload to the endpoint. The template has many **sendXXX** methods for this purpose. But before we send the payload to the file endpoint we must also specify what filename to store the file as. This is done by sending meta data with the payload. In Camel metadata is sent as headers. Headers is just a plain `Map<String, Object>`. So if we needed to send several metadata then we could construct an ordinary HashMap and put the values in there. But as we just need to send one header with the filename Camel has a convenient send method `sendBodyAndHeader` so we choose this one.

```
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        System.out.println("Hello ReportIncidentService is called from " + parameters.getGivenName());

        String data = parameters.getDetails();

        // store the data as a file
        String filename = parameters.getIncidentId() + ".txt";
        // send the data to the endpoint and the header contains what filename it should be stored as
        template.sendBodyAndHeader("backup", data, "org.apache.camel.file.name", filename);

        OutputReportIncident out = new OutputReportIncident();
        out.setCode("OK");
        return out;
    }
```

The template in the code above uses 4 parameters:

- the endpoint name, in this case the id referring to the endpoint defined in Spring XML in the camelContext element.
- the payload, can be any kind of object
- the key for the header, in this case a Camel keyword to set the filename
- and the value for the header

Running the example

We start our integration with maven using `mvn jetty:run`. Then we open a browser and hit `http://localhost:8080`. Jetty is so smart that it display a frontpage with links to the deployed application so just hit the link and you get our application. Now we hit append /services to the URL to access the Axis frontpage. The URL should be `http://localhost:8080/camel-example-axis/services`.

You can then test it using a web service test tools such as SoapUI.
Hitting the service will output to the console

```
2008-09-06 15:01:41.718::INFO:  Started SelectChannelConnector @ 0.0.0.0:8080
[INFO] Started Jetty Server
Hello ReportIncidentService is called from Ibsen
```

And there should be a file in the target subfolder.

```
dir target /b
123.txt
```

Unit Testing

We would like to be able to unit test our **ReportIncidentService** class. So we add junit to the maven dependency:

```
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.2</version>
            <scope>test</scope>
        </dependency>
```

And then we create a plain junit testcase for our service class.

```
package org.apache.camel.example.axis;

import junit.framework.TestCase;
import org.apache.camel.example.reportincident.InputReportIncident;
import org.apache.camel.example.reportincident.OutputReportIncident;

/**
 * Unit test of service
 */
public class ReportIncidentServiceTest extends TestCase {

    public void testIncident() {
        ReportIncidentService service = new ReportIncidentService();

        InputReportIncident input = createDummyIncident();
        OutputReportIncident output = service.reportIncident(input);
        assertEquals("OK", output.getCode());
    }

   protected InputReportIncident createDummyIncident() {
        InputReportIncident input = new InputReportIncident();
        input.setEmail("davsclaus@apache.org");
        input.setIncidentId("12345678");
        input.setIncidentDate("2008-07-13");
        input.setPhone("+45 2962 7576");
        input.setSummary("Failed operation");
        input.setDetails("The wrong foot was operated.");
        input.setFamilyName("Ibsen");
        input.setGivenName("Claus");
        return input;
    }

}
```

Then we can run the test with maven using: `mvn test`. But we will get a failure:

```
Running org.apache.camel.example.axis.ReportIncidentServiceTest
Hello ReportIncidentService is called from Claus
Tests run: 1, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: 0.235 sec <<< FAILURE!

Results :

Tests in error:
  testIncident(org.apache.camel.example.axis.ReportIncidentServiceTest)

Tests run: 1, Failures: 0, Errors: 1, Skipped: 0
```

What is the problem? Well our service uses a CamelProducer (the template) to send a message to the file endpoint so the message will be stored in a file. What we need is to get hold of such a producer and inject it on our service, by calling the setter.

Since Camel is very light weight and embedable we are able to create a CamelContext and add the endpoint in our unit test code directly. We do this to show how this is possible:

```
    private CamelContext context;

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        // CamelContext is just created like this
        context = new DefaultCamelContext();

        // then we can create our endpoint and set the options
        FileEndpoint endpoint = new FileEndpoint();
        // the endpoint must have the camel context set also
        endpoint.setCamelContext(context);
        // our output folder
        endpoint.setFile(new File("target"));
        // and the option not to append
        endpoint.setAppend(false);

        // then we add the endpoint just in java code just as the spring XML, we register it with the "backup"
id.
        context.addSingletonEndpoint("backup", endpoint);

        // finally we need to start the context so Camel is ready to rock
        context.start();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
        // and we are nice boys so we stop it to allow resources to clean up
        context.stop();
    }
```

So now we are ready to set the ProducerTemplate on our service, and we get a hold of that baby from the CamelContext as:

```
    public void testIncident() {
        ReportIncidentService service = new ReportIncidentService();

        // get a producer template from the camel context
        ProducerTemplate template = context.createProducerTemplate();
        // inject it on our service using the setter
        service.setTemplate(template);

        InputReportIncident input = createDummyIncident();
        OutputReportIncident output = service.reportIncident(input);
        assertEquals("OK", output.getCode());
    }
```

And this time when we run the unit test its a success:

```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

We would like to test that the file exists so we add these two lines to our test method:

```
        // should generate a file also
        File file = new File("target/" + input.getIncidentId() + ".txt");
        assertTrue("File should exists", file.exists());
```

Smarter Unit Testing with Spring

The unit test above requires us to assemble the Camel pieces manually in java code. What if we would like our unit test to use our spring configuration file **axis-example-context.xml** where we already have setup the endpoint. And of course we would like to test using this configuration file as this is the real file we will use. Well hey presto the xml file is a spring ApplicationContext file and spring is able to load it, so we go the spring path for unit testing. First we add the spring-test jar to our maven dependency:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <scope>test</scope>
</dependency>
```

And then we refactor our unit test to be a standard spring unit class. What we need to do is to extend `AbstractJUnit38SpringContextTests` instead of `TestCase` in our unit test. Since Spring 2.5 embraces annotations we will use one as well to instruct what our xml configuration file is located:

```
@ContextConfiguration(locations = "classpath:axis-example-context.xml")
public class ReportIncidentServiceTest extends AbstractJUnit38SpringContextTests {
```

What we must remember to add is the **classpath:** prefix as our xml file is located in `src/main/resources`. If we omit the prefix then Spring will by default try to locate the xml file in the current package and that is org.apache.camel.example.axis. If the xml file is located outside the classpath you can use file: prefix instead. So with these two modifications we can get rid of all the setup and teardown code we had before and now we will test our real configuration.

The last change is to get hold of the producer template and now we can just refer to the bean id it has in the spring xml file:

```
<!-- producer template exposed with this id -->
<camel:template id="camelTemplate"/>
```

So we get hold of it by just getting it from the spring ApplicationContext as all spring users is used to do:

```
// get a producer template from the the spring context
ProducerTemplate template = (ProducerTemplate) applicationContext.getBean("camelTemplate");
// inject it on our service using the setter
service.setTemplate(template);
```

Now our unit test is much better, and a real power of Camel is that is fits nicely with Spring and you can use standard Spring'ish unit test to test your Camel applications as well.

Unit Test calling WebService

What if you would like to execute a unit test where you send a webservice request to the **AxisReportIncidentService** how do we unit test this one? Well first of all the code is merely just a delegate to our real service that we have just tested, but nevertheless its a good question and we would like to know how. Well the answer is that we can exploit that fact that Jetty is also a slim web container that can be embedded anywhere just as Camel can. So we add this to our pom.xml:

```
<dependency>
    <groupId>org.mortbay.jetty</groupId>
    <artifactId>jetty</artifactId>
    <version>${jetty-version}</version>
    <scope>test</scope>
</dependency>
```

Then we can create a new class **AxisReportIncidentServiceTest** to unit test with Jetty. The code to setup Jetty is shown below with code comments:

```
public class AxisReportIncidentServiceTest extends TestCase {

    private Server server;

    private void startJetty() throws Exception {
        // create an embedded Jetty server
        server = new Server();

        // add a listener on port 8080 on localhost (127.0.0.1)
        Connector connector = new SelectChannelConnector();
        connector.setPort(8080);
        connector.setHost("127.0.0.1");
        server.addConnector(connector);

        // add our web context path
        WebAppContext wac = new WebAppContext();
        wac.setContextPath("/unittest");
        // set the location of the exploded webapp where WEB-INF is located
        // this is a nice feature of Jetty where we can point to src/main/webapp
        wac.setWar("./src/main/webapp");
        server.setHandler(wac);

        // then start Jetty
        server.setStopAtShutdown(true);
        server.start();
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        startJetty();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
        server.stop();
    }

}
```

Now we just need to send the incident as a webservice request using Axis. So we add the following code:

```
public void testReportIncidentWithAxis() throws Exception {
    // the url to the axis webservice exposed by jetty
    URL url = new URL("http://localhost:8080/unittest/services/ReportIncidentPort");

    // Axis stuff to get the port where we can send the webservice request
    ReportIncidentService_ServiceLocator locator = new ReportIncidentService_ServiceLocator();
    ReportIncidentService_PortType port = locator.getReportIncidentPort(url);

    // create input to send
    InputReportIncident input = createDummyIncident();
    // send the webservice and get the response
    OutputReportIncident output = port.reportIncident(input);
    assertEquals("OK", output.getCode());

    // should generate a file also
    File file = new File("target/" + input.getIncidentId() + ".txt");
    assertTrue("File should exists", file.exists());
}

protected InputReportIncident createDummyIncident() {
    InputReportIncident input = new InputReportIncident();
    input.setEmail("davsclaus@apache.org");
    input.setIncidentId("12345678");
    input.setIncidentDate("2008-07-13");
    input.setPhone("+45 2962 7576");
    input.setSummary("Failed operation");
    input.setDetails("The wrong foot was operated.");
    input.setFamilyName("Ibsen");
    input.setGivenName("Claus");
    return input;
}
```

And now we have an unittest that sends a webservice request using good old Axis.

Annotations

Both Camel and Spring has annotations that can be used to configure and wire trivial settings more elegantly. Camel has the endpoint annotation `@EndpointInjected` that is just what we need. With this annotation we can inject the endpoint into our service. The annotation takes either a name or uri parameter. The name is the bean id in the Registry. The uri is the URI configuration for the endpoint. Using this you can actually inject an endpoint that you have not defined in the camel context. As we have defined our endpoint with the id **backup** we use the name parameter.

```
@EndpointInject(name = "backup")
private ProducerTemplate template;
```

Camel is smart as `@EndpointInjected` supports different kinds of object types. We like the ProducerTemplate so we just keep it as it is.
Since we use annotations on the field directly we do not need to set the property in the spring xml file so we change our service bean:

```
<bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>
```

Running the unit test with `mvn test` reveals that it works nicely.

And since we use the `@EndpointInjected` that refers to the endpoint with the id backup directly we can loose the template tag in the xml, so its shorter:

```
<bean id="incidentservice" class="org.apache.camel.example.axis.ReportIncidentService"/>

<camel:camelContext id="camelContext">
    <!-- producer template exposed with this id -->
    <camel:template id="camelTemplate"/>

    <!-- endpoint named backup that is configued as a file component -->
    <camel:endpoint id="backup" uri="file://target?append=false"/>

</camel:camelContext>
```

And the final touch we can do is that since the endpoint is injected with concrete endpoint to use we can remove the `"backup"` name parameter when we send the message. So we change from:

```
    // send the data to the endpoint and the header contains what filename it should be stored as
    template.sendBodyAndHeader("backup", data, "org.apache.camel.file.name", filename);
```

To without the name:

```
    // send the data to the endpoint and the header contains what filename it should be stored as
    template.sendBodyAndHeader(data, "org.apache.camel.file.name", filename);
```

Then we avoid to duplicate the name and if we rename the endpoint name then we don't forget to change it in the code also.

The End

This tutorial hasn't really touched the one of the key concept of Camel as a powerful routing and mediation framework. But we wanted to demonstrate its flexibility and that it integrates well with even older frameworks such as Apache Axis 1.4.

Check out the other tutorials on Camel and the other examples.

Note that the code shown here also applies to Camel 1.4 so actually you can get started right away with the released version of Camel. As this time of writing Camel 1.5 is work in progress.

See Also

- Tutorials
- Examples