Tutorial-Example-ReportIncident

Tutorial - camel-example-reportincident

Introduction

Creating this tutorial was inspired by a real life use-case I discussed over the phone with a colleague. He was working at a client whom uses a heavy-weight integration platform from a very large vendor. He was in talks with developer shops to implement a new integration on this platform. His trouble was the shop tripled the price when they realized the platform of choice. So I was wondering how we could do this integration with Camel. Can it be done, without tripling the cost .

This tutorial is written during the development of the integration. I have decided to start off with a sample that isn't Camel's but standard Java and then plugin Camel as we goes. Just as when people needed to learn Spring you could consume it piece by piece, the same goes with Camel.

The target reader is person whom hasn't experience or just started using Camel.

Motivation for this tutorial

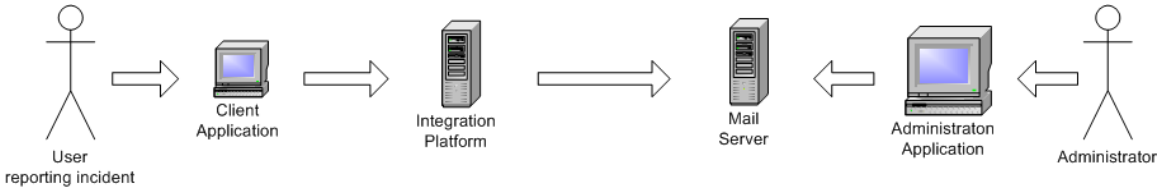
I wrote this tutorial motivated as Camel lacked an example application that was based on the web application deployment model. The entire world hasn't moved to pure OSGi deployments yet.

> The full source code for this tutorial as complete is part of the Apache Camel distribution in the `examples/camel-example-reportincident` directory

The use-case

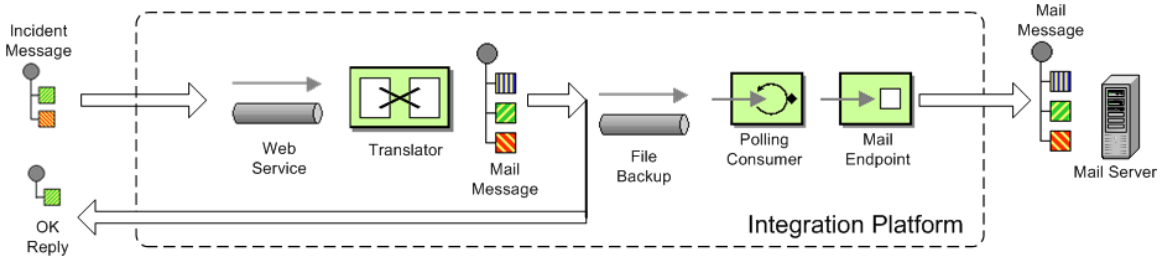
The goal is to allow staff to report incidents into a central administration. For that they use client software where they report the incident and submit it to the central administration. As this is an integration in a transition phase the administration should get these incidents by email whereas they are manually added to the database. The client software should gather the incident and submit the information to the integration platform that in term will transform the report into an email and send it to the central administrator for manual processing.

The figure below illustrates this process. The end users reports the incidents using the client applications. The incident is sent to the central integration platform as webservice. The integration platform will process the incident and send an OK acknowledgment back to the client. Then the integration will transform the message to an email and send it to the administration mail server. The users in the administration will receive the emails and take it from there.



In EIP patterns

We distill the use case as EIP patterns:



Parts

This tutorial is divided into sections and parts:

Section A: Existing Solution, how to slowly use Camel

Part 1 - This first part explain how to setup the project and get a webservice exposed using Apache CXF. In fact we don't touch Camel yet.

Part 2 - Now we are ready to introduce Camel piece by piece (without using Spring or any XML configuration file) and create the full feature integration. This part will introduce different Camel's concepts and How we can build our solution using them like :

- CamelContext
- Endpoint, Exchange & Producer
- Components : Log, File

Part 3 - Continued from part 2 where we implement that last part of the solution with the event driven consumer and how to send the email through the Mail component.

Section B: The Camel Solution

Part 4 - We now turn into the path of Camel where it excels - the routing.
Part 5 - Is about how embed Camel with Spring and using CXF endpoints directly in Camel
Part 6 - Showing a alternative solution primarily using XML instead of Java code

Using Axis 2
See this blog entry by Sagara demonstrating how to use Apache Axis 2 instead of Apache CXF as the web service framework.

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part1

Part 1

Prerequisites

This tutorial uses the following frameworks:

- Maven 3.0.4
- Apache Camel 2.10.0
- Apache CXF 2.6.1
- Spring 3.0.7

Note: The sample project can be downloaded, see the resources section.

Initial Project Setup

We want the integration to be a standard .war application that can be deployed in any web container such as Tomcat, Jetty or even heavy weight application servers such as WebLogic or WebSphere. There fore we start off with the standard Maven webapp project that is created with the following long archetype command:

```
mvn archetype:create -DgroupId=org.apache.camel
-DartifactId=camel-example-reportincident -DarchetypeArtifactId=maven-archetype-webapp
```

Notice that the groupId etc. doens't have to be org.apache.camel it can be com.mycompany.whatever. But I have used these package names as the example is an official part of the Camel distribution.

Then we have the basic maven folder layout. We start out with the webservice part where we want to use Apache CXF for the webservice stuff. So we add this to the pom.xml

```
    <properties>
        <cxf-version>2.6.1</cxf-version>
    </properties>

    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-core</artifactId>
        <version>${cxf-version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>${cxf-version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transports-http</artifactId>
        <version>${cxf-version}</version>
    </dependency>
```

Developing the WebService

As we want to develop webservice with the contract first approach we create our .wsdl file. As this is a example we have simplified the model of the incident to only include 8 fields. In real life the model would be a bit more complex, but not to much.

We put the wsdl file in the folder `src/main/webapp/WEB-INF/wsdl` and name the file `report_incident.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>
```

CXF wsdl2java

Then we integration the CXF wsdl2java generator in the pom.xml so we have CXF generate the needed POJO classes for our webservice contract.
However at first we must configure maven to live in the modern world of Java 1.6 so we must add this to the pom.xml

```
<!-- to compile with 1.6 -->
<plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <configuration>
  <source>1.6</source>
  <target>1.6</target>
 </configuration>
</plugin>
```

And then we can add the CXF wsdl2java code generator that will hook into the compile goal so its automatic run all the time:

```
<!-- CXF wsdl2java generator, will plugin to the compile goal -->
<plugin>
 <groupId>org.apache.cxf</groupId>
 <artifactId>cxf-codegen-plugin</artifactId>
 <version>${cxf-version}</version>
 <executions>
  <execution>
   <id>generate-sources</id>
   <phase>generate-sources</phase>
   <configuration>
    <sourceRoot>${basedir}/target/generated/src/main/java</sourceRoot>
    <wsdlOptions>
     <wsdlOption>
      <wsdl>${basedir}/src/main/webapp/WEB-INF/wsdl/report_incident.wsdl</wsdl>
     </wsdlOption>
    </wsdlOptions>
   </configuration>
   <goals>
    <goal>wsdl2java</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

You are now setup and should be able to compile the project. So running the `mvn compile` should run the CXF wsdl2java and generate the source code in the folder `&{basedir}/target/generated/src/main/java` that we specified in the pom.xml above. Since its in the `target/generated/src/main/java` maven will pick it up and include it in the build process.

Configuration of the web.xml

Next up is to configure the web.xml to be ready to use CXF so we can expose the webservice.
As Spring is the center of the universe, or at least is a very important framework in today's Java land we start with the listener that kick-starts Spring. This is the usual piece of code:

```
 <!-- the listener that kick-starts Spring -->
 <listener>

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

And then we have the CXF part where we define the CXF servlet and its URI mappings to which we have chosen that all our webservices should be in the path `/webservices/`

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

 <!-- all our webservices are mapped under this URI pattern -->
 <servlet-mapping>
  <servlet-name>CXFServlet</servlet-name>
  <url-pattern>/webservices/*</url-pattern>
 </servlet-mapping>
```

Then the last piece of the puzzle is to configure CXF, this is done in a spring XML that we link to fron the web.xml by the standard Spring `contextConfigLocation` property in the web.xml

```
 <!-- location of spring xml files -->
 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:cxf-config.xml</param-value>
 </context-param>
```

We have named our CXF configuration file `cxf-config.xml` and its located in the root of the classpath. In Maven land that is we can have the `cxf-config.xml` file in the `src/main/resources` folder. We could also have the file located in the WEB-INF folder for instance `<param-value>/WEB-INF/cxf-config.xml</param-value>`.

Getting rid of the old jsp world

The maven archetype that created the basic folder structure also created a sample .jsp file index.jsp. This file `src/main/webapp/index.jsp` should be deleted.

Configuration of CXF

The cxf-config.xml is as follows:

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

    <!-- implementation of the webservice -->
    <bean id="reportIncidentEndpoint"
class="org.apache.camel.example.reportincident.ReportIncidentEndpointImpl"/>

    <!-- export the webservice using jaxws -->
    <jaxws:endpoint id="reportIncident"
                    implementor="#reportIncidentEndpoint"
                    address="/incident"
                    wsdlLocation="/WEB-INF/wsdl/report_incident.wsdl"
                    endpointName="s:ReportIncidentPort"
                    serviceName="s:ReportIncidentService"
                    xmlns:s="http://reportincident.example.camel.apache.org"/>

</beans>
```

The configuration is standard CXF and is documented at the Apache CXF website.

The 3 import elements is needed by CXF and they must be in the file.

Noticed that we have a spring bean **reportIncidentEndpoint** that is the implementation of the webservice endpoint we let CXF expose.
Its linked from the jaxws element with the implementator attribute as we use the # mark to identify its a reference to a spring bean. We could have stated the classname directly as `implementor="org.apache.camel.example.reportincident.ReportIncidentEndpoint"` but then we lose the ability to let the ReportIncidentEndpoint be configured by spring.
The **address** attribute defines the relative part of the URL of the exposed webservice. **wsdlLocation** is an optional parameter but for persons like me that likes contract-first we want to expose our own .wsdl contracts and not the auto generated by the frameworks, so with this attribute we can link to the real .wsdl file. The last stuff is needed by CXF as you could have several services so it needs to know which this one is. Configuring these is quite easy as all the information is in the wsdl already.

Implementing the ReportIncidentEndpoint

Phew after all these meta files its time for some java code so we should code the implementor of the webservice. So we fire up `mvn compile` to let CXF generate the POJO classes for our webservice and we are ready to fire up a Java editor.

You can use `mvn idea:idea` or `mvn eclipse:eclipse` to create project files for these editors so you can load the project. However IDEA has been smarter lately and can load a pom.xml directly.

As we want to quickly see our webservice we implement just a quick and dirty as it can get. At first beware that since its jaxws and Java 1.5 we get annotations for the money, but they reside on the interface so we can remove them from our implementations so its a nice plain POJO again:

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

/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

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

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

}
```

We just output the person that invokes this webservice and returns a OK response. This class should be in the maven source root folder `src/main/java` under the package name `org.apache.camel.example.reportincident`. Beware that the maven archetype tool didn't create the `src/main/java folder`, so you should create it manually.

To test if we are home free we run `mvn clean compile`.

Running our webservice

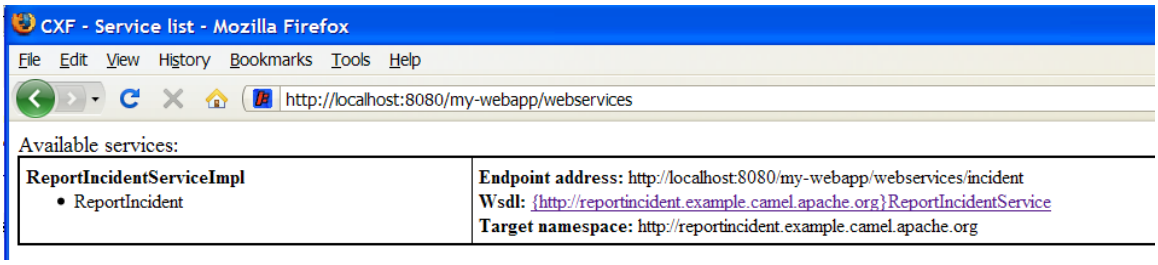
Now that the code compiles we would like to run it inside a web container, for this purpose we make use of Jetty which we will bootstrap using it's plugin `org.mortbay.jetty:maven-jetty-plugin`:

```
      <build>
          <plugins>
              ...
              <!-- so we can run mvn jetty:run -->
              <plugin>
                  <groupId>org.mortbay.jetty</groupId>
                  <artifactId>maven-jetty-plugin</artifactId>
                  <version>${jetty-version}</version>
              </plugin>
```

Notice: We make use of the Jetty version being defined inside the Camel's Parent POM.

So to see if everything is in order we fire up jetty with `mvn jetty:run` and if everything is okay you should be able to access `http://localhost:8080`.
Jetty is smart that it will list the correct URI on the page to our web application, so just click on the link. This is smart as you don't have to remember the exact web context URI for your application - just fire up the default page and Jetty will help you.

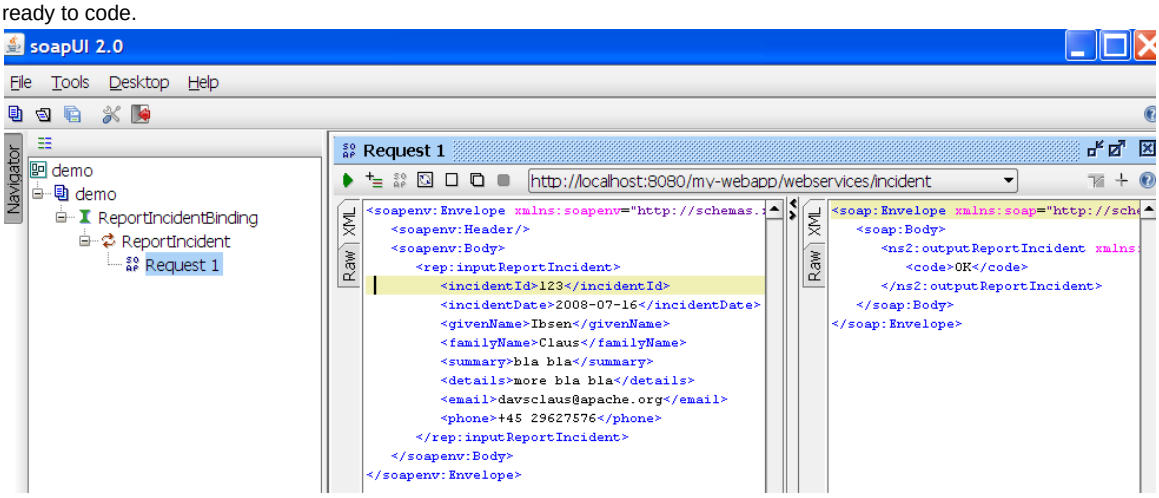
So where is the damn webservice then? Well as we did configure the web.xml to instruct the CXF servlet to accept the pattern `/webservices/*` we should hit this URL to get the attention of CXF: `http://localhost:8080/camel-example-reportincident/webservices`.



Hitting the webservice

Now we have the webservice running in a standard .war application in a standard web container such as Jetty we would like to invoke the webservice and see if we get our code executed. Unfortunately this isn't the easiest task in the world - its not so easy as a REST URL, so we need tools for this. So we fire up our trusty webservice tool SoapUI and let it be the one to fire the webservice request and see the response.

Using SoapUI we sent a request to our webservice and we got the expected OK response and the console outputs the System.out so we are ready to code.




```
[INFO] Started Jetty Server
Hello ReportIncidentWebService is called from Ibsen
```

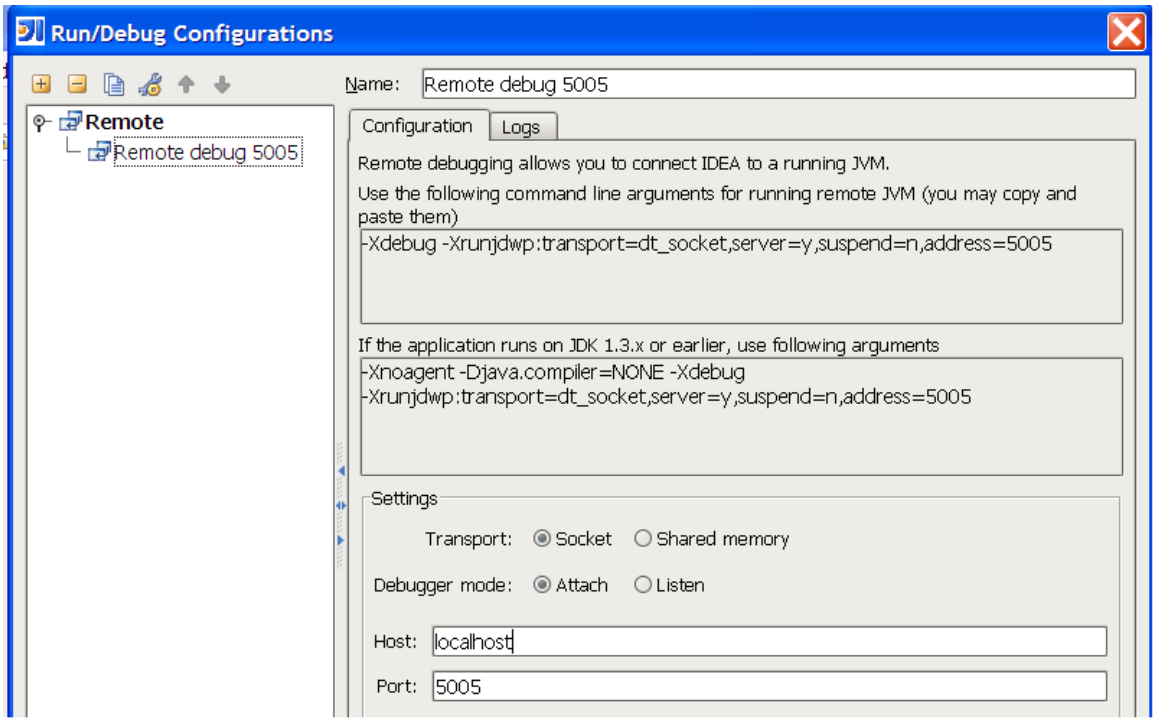
Remote Debugging

Okay a little sidestep but wouldn't it be cool to be able to debug your code when its fired up under Jetty? As Jetty is started from maven, we need to instruct maven to use debug mode.
Se we set the `MAVEN_OPTS` environment to start in debug mode and listen on port 5005.

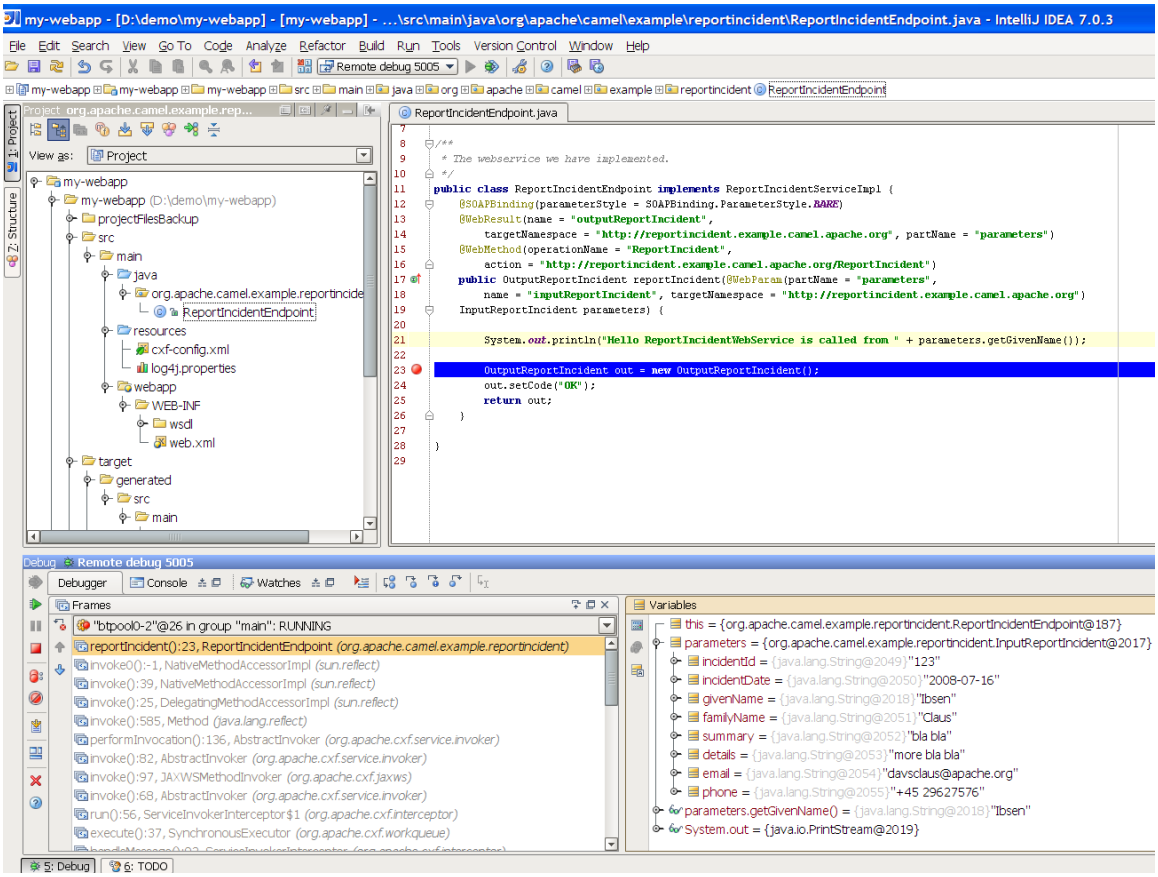
```
MAVEN_OPTS=-Xmx512m -XX:MaxPermSize=128m -Xdebug
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=5005
```

Then you need to restart Jetty so its stopped with **ctrl + c**. Remember to start a new shell to pickup the new environment settings. And start jetty again.

Then we can from our IDE attach a remote debugger and debug as we want.
First we configure IDEA to attach to a remote debugger on port 5005:



Then we set a breakpoint in our code `ReportIncidentEndpoint` and hit the SoapUI once again and we are breaked at the breakpoint where we can inspect the parameters:



Adding a unit test

Oh so much hard work just to hit a webservice, why can't we just use an unit test to invoke our webservice? Yes of course we can do this, and that's the next step.
First we create the folder structure `src/test/java` and `src/test/resources`. We then create the unit test in the `src/test/java` folder.

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

import junit.framework.TestCase;

/**
 * Plain JUnit test of our webservice.
 */
public class ReportIncidentEndpointTest extends TestCase {

}
```

Here we have a plain old JUnit class. As we want to test webservices we need to start and expose our webservice in the unit test before we can test it. And JAXWS has pretty decent methods to help us here, the code is simple as:

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

    private static String ADDRESS = "http://localhost:9090/unittest";

    protected void startServer() throws Exception {
        // We need to start a server that exposes or webservice during the unit
testing
        // We use jaxws to do this pretty simple
        ReportIncidentEndpointImpl server = new ReportIncidentEndpointImpl();
        Endpoint.publish(ADDRESS, server);
    }
```

The Endpoint class is the `javax.xml.ws.Endpoint` that under the covers looks for a provider and in our case its CXF - so its CXF that does the heavy lifting of exposing out webservice on the given URL address. Since our class ReportIncidentEndpointImpl implements the interface **ReportIncidentEndpoint** that is decorated with all the jaxws annotations it got all the information it need to expose the webservice. Below is the CXF wsdl2java generated interface:

```
/*
 *
 */

package org.apache.camel.example.reportincident;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.soap.SOAPBinding.ParameterStyle;
import javax.xml.bind.annotation.XmlSeeAlso;

/**
 * This class was generated by Apache CXF 2.1.1
 * Wed Jul 16 12:40:31 CEST 2008
 * Generated source version: 2.1.1
 *
 */

 /*
  *
  */


@WebService(targetNamespace = "http://reportincident.example.camel.apache.org", name =
"ReportIncidentEndpoint")
@XmlSeeAlso({ObjectFactory.class})
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)

public interface ReportIncidentEndpoint {

/*
 *
 */

    @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.BARE)
    @WebResult(name = "outputReportIncident", targetNamespace =
"http://reportincident.example.camel.apache.org", partName = "parameters")
    @WebMethod(operationName = "ReportIncident", action =
"http://reportincident.example.camel.apache.org/ReportIncident")
    public OutputReportIncident reportIncident(
        @WebParam(partName = "parameters", name = "inputReportIncident",
targetNamespace = "http://reportincident.example.camel.apache.org")
        InputReportIncident parameters
    );
}
```

Next up is to create a webservice client so we can invoke our webservice. For this we actually use the CXF framework directly as its a bit more easier to create a client using this framework than using the JAXWS style. We could have done the same for the server part, and you should do this if you need more power and access more advanced features.

```
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
...

protected ReportIncidentEndpoint createCXFClient() {
    // we use CXF to create a client for us as its easier than JAXWS and works
    JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
    factory.setServiceClass(ReportIncidentEndpoint.class);
    factory.setAddress(ADDRESS);
    return (ReportIncidentEndpoint) factory.create();
}
```

So now we are ready for creating a unit test. We have the server and the client. So we just create a plain simple unit test method as the usual junit style:

```
public void testRendportIncident() throws Exception {
    startServer();

    ReportIncidentEndpoint client = createCXFClient();

    InputReportIncident input = new InputReportIncident();
    input.setIncidentId("123");
    input.setIncidentDate("2008-07-16");
    input.setGivenName("Claus");
    input.setFamilyName("Ibsen");
    input.setSummary("bla bla");
    input.setDetails("more bla bla");
    input.setEmail("davsclaus@apache.org");
    input.setPhone("+45 2962 7576");

    OutputReportIncident out = client.reportIncident(input);
    assertEquals("Response code is wrong", "OK", out.getCode());
}
```

Now we are nearly there. But if you run the unit test with `mvn test` then it will fail. Why!!! Well its because that CXF needs is missing some dependencies during unit testing. In fact it needs the web container, so we need to add this to our **pom.xml**.

```
<!-- cxf web container for unit testing -->
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
    <scope>test</scope>
</dependency>
```

Well what is that, CXF also uses Jetty for unit test - well its just shows how agile, embedable and popular Jetty is.

So lets run our junit test with, and it reports:

```
mvn test
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO] BUILD SUCCESSFUL
```

Yep thats it for now. We have a basic project setup.

End of part 1

Thanks for being patient and reading all this more or less standard Maven, Spring, JAXWS and Apache CXF stuff. Its stuff that is well covered on the net, but I wanted a full fledged tutorial on a maven project setup that is web service ready with Apache CXF. We will use this as a base for the next part where we demonstrate how Camel can be digested slowly and piece by piece just as it was back in the times when was introduced and was learning the Spring framework that we take for granted today.

#Resources

- Apache CXF user guide

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part2

Part 2

Adding Camel

In this part we will introduce Camel so we start by adding Camel to our pom.xml:

```
      <properties>
           ...
           <camel-version>1.4.0</camel-version>
       </properties>

       <!-- camel -->
       <dependency>
           <groupId>org.apache.camel</groupId>
           <artifactId>camel-core</artifactId>
           <version>${camel-version}</version>
       </dependency>
```

That's it, only **one** dependency for now.

Synchronize IDE
If you continue from part 1, remember to update your editor project settings since we have introduce new .jar files. For instance IDEA has a feature to synchronize with Maven projects.

Now we turn towards our webservice endpoint implementation where we want to let Camel have a go at the input we receive. As Camel is very non invasive its basically a .jar file then we can just grap Camel but creating a new instance of `DefaultCamelContext` that is the hearth of Camel its context.

```
CamelContext camel = new DefaultCamelContext();
```

In fact we create a constructor in our webservice and add this code:

```
   private CamelContext camel;

   public ReportIncidentEndpointImpl() throws Exception {
       // create the camel context that is the "heart" of Camel
       camel = new DefaultCamelContext();

       // add the log component
       camel.addComponent("log", new LogComponent());

       // start Camel
       camel.start();
   }
```

Logging the "Hello World"

Here at first we want Camel to log the **givenName** and **familyName** parameters we receive, so we add the `LogComponent` with the key **log**. And we must **start** Camel before its ready to act.

> **Component Documentation**
> The Log and File components is documented as well, just click on the links. Just return to this documentation later when you must use these components for real.

Then we change the code in the method that is invoked by Apache CXF when a webservice request arrives. We get the name and let Camel have a go at it in the new method we create **sendToCamel**:

```
public OutputReportIncident reportIncident(InputReportIncident parameters) {
    String name = parameters.getGivenName() + " " + parameters.getFamilyName();

    // let Camel do something with the name
    sendToCamelLog(name);

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

Next is the Camel code. At first it looks like there are many code lines to do a simple task of logging the name - yes it is. But later you will in fact realize this is one of Camels true power. Its concise API. Hint: The same code can be used for **any** component in Camel.

```
    private void sendToCamelLog(String name) {
        try {
            // get the log component
            Component component = camel.getComponent("log");

            // create an endpoint and configure it.
            // Notice the URI parameters this is a common pratice in Camel to
configure
            // endpoints based on URI.
            // com.mycompany.part2 = the log category used. Will log at INFO level as
default
            Endpoint endpoint = component.createEndpoint("log:com.mycompany.part2");

            // create an Exchange that we want to send to the endpoint
            Exchange exchange = endpoint.createExchange();
            // set the in message payload (=body) with the name parameter
            exchange.getIn().setBody(name);

            // now we want to send the exchange to this endpoint and we then need a
producer
            // for this, so we create and start the producer.
            Producer producer = endpoint.createProducer();
            producer.start();
            // process the exchange will send the exchange to the log component, that
will process
            // the exchange and yes log the payload
            producer.process(exchange);

            // stop the producer, we want to be nice and cleanup
            producer.stop();



        } catch (Exception e) {
            // we ignore any exceptions and just rethrow as runtime
            throw new RuntimeException(e);

        }
    }
```

Okay there are code comments in the code block above that should explain what is happening. We run the code by invoking our unit test with maven `mvn test`, and we should get this log line:

```
INFO: Exchange[BodyType:String, Body:Claus Ibsen]
```

Write to file - easy with the same code style

Okay that isn't to impressive, Camel can log 😊 Well I promised that the above code style can be used for **any** component, so let's store the payload in a file. We do this by adding the file component to the Camel context

```
        // add the file component
        camel.addComponent("file", new FileComponent());
```

And then we let camel write the payload to the file after we have logged, by creating a new method **sendToCamelFile**. We want to store the payload in filename with the incident id so we need this parameter also:

```
        // let Camel do something with the name
        sendToCamelLog(name);
        sendToCamelFile(parameters.getIncidentId(), name);
```

And then the code that is 99% identical. We have change the URI configuration when we create the endpoint as we pass in configuration parameters to the file component.
And then we need to set the output filename and this is done by adding a special header to the exchange. That's the only difference:

```
    private void sendToCamelFile(String incidentId, String name) {
        try {
            // get the file component
            Component component = camel.getComponent("file");

            // create an endpoint and configure it.
            // Notice the URI parameters this is a common pratice in Camel to
configure
            // endpoints based on URI.
            // file://target instructs the base folder to output the files. We put in
the target folder
            // then its actumatically cleaned by mvn clean
            Endpoint endpoint = component.createEndpoint("file://target");

            // create an Exchange that we want to send to the endpoint
            Exchange exchange = endpoint.createExchange();
            // set the in message payload (=body) with the name parameter
            exchange.getIn().setBody(name);

            // now a special header is set to instruct the file component what the
output filename
            // should be
            exchange.getIn().setHeader(FileComponent.HEADER_FILE_NAME, "incident-" +
incidentId + ".txt");

            // now we want to send the exchange to this endpoint and we then need a
producer
            // for this, so we create and start the producer.
            Producer producer = endpoint.createProducer();
            producer.start();
            // process the exchange will send the exchange to the file component, that
will process
            // the exchange and yes write the payload to the given filename
            producer.process(exchange);

            // stop the producer, we want to be nice and cleanup
            producer.stop();
        } catch (Exception e) {
            // we ignore any exceptions and just rethrow as runtime
            throw new RuntimeException(e);
        }
    }
```

After running our unit test again with `mvn test` we have a output file in the target folder:

```
D:\demo\part-two>type target\incident-123.txt
Claus Ibsen
```

Fully java based configuration of endpoints

In the file example above the configuration was URI based. What if you want 100% java setter based style, well this is of course also possible. We just need to cast to the component specific endpoint and then we have all the setters available:

```
            // create the file endpoint, we cast to FileEndpoint because then we can
do
            // 100% java settter based configuration instead of the URI sting based
            // must pass in an empty string, or part of the URI configuration if
wanted
            FileEndpoint endpoint = (FileEndpoint)component.createEndpoint("");
            endpoint.setFile(new File("target/subfolder"));
            endpoint.setAutoCreate(true);
```

That's it. Now we have used the setters to configure the `FileEndpoint` that it should store the file in the folder target/subfolder. Of course Camel now stores the file in the subfolder.

```
D:\demo\part-two>type target\subfolder\incident-123.txt
Claus Ibsen
```

Lessons learned

Okay I wanted to demonstrate how you can be in 100% control of the configuration and usage of Camel based on plain Java code with no hidden magic or special **XML** or other configuration files. Just add the camel-core.jar and you are ready to go.

You must have noticed that the code for sending a message to a given endpoint is the same for both the **log** and **file**, in fact **any** Camel endpoint. You as the client shouldn't bother with component specific code such as file stuff for file components, jms stuff for JMS messaging etc. This is what the Message Endpoint EIP pattern is all about and Camel solves this very very nice - a key pattern in Camel.

Reducing code lines

Now that you have been introduced to Camel and one of its masterpiece patterns solved elegantly with the Message Endpoint its time to give productive and show a solution in fewer code lines, in fact we can get it down to 5, 4, 3, 2 .. yes only **1 line of code**.

The key is the **ProducerTemplate** that is a Spring'ish xxxTemplate based producer. Meaning that it has methods to send messages to any Camel endpoints. First of all we need to get hold of such a template and this is done from the CamelContext

```
    private ProducerTemplate template;

    public ReportIncidentEndpointImpl() throws Exception {
        ...

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer
for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // start Camel
        camel.start();
    }
```

Now we can use **template** for sending payloads to any endpoint in Camel. So all the logging gabble can be reduced to:

```
    template.sendBody("log:com.mycompany.part2.easy", name);
```

And the same goes for the file, but we must also send the header to instruct what the output filename should be:

```
    String filename = "easy-incident-" + incidentId + ".txt";
    template.sendBodyAndHeader("file://target/subfolder", name,
FileComponent.HEADER_FILE_NAME, filename);
```

Reducing even more code lines

Well we got the Camel code down to 1-2 lines for sending the message to the component that does all the heavy work of wring the message to a file etc. But we still got 5 lines to initialize Camel.

```
    camel = new DefaultCamelContext();
    camel.addComponent("log", new LogComponent());
    camel.addComponent("file", new FileComponent());
    template = camel.createProducerTemplate();
    camel.start();
```

This can also be reduced. All the standard components in Camel is auto discovered on-the-fly so we can remove these code lines and we are down to 3 lines.

Component auto discovery
When an endpoint is requested with a scheme that Camel hasn't seen before it will try to look for it in the classpath. It will do so by looking for special Camel component marker files that reside in the folder `META-INF/services/org/apache/camel/component`. If there are files in this folder it will read them as the filename is the **scheme** part of the URL. For instance the **log** component is defined in this file `META-INF/services/org/apache/component/log` and its content is:

```
class=org.apache.camel.component.log.LogComponent
```

The class property defines the component implementation.

Tip: End-users can create their 3rd party components using the same technique and have them been auto discovered on-the-fly.

Okay back to the 3 code lines:

```
    camel = new DefaultCamelContext();
    template = camel.createProducerTemplate();
    camel.start();
```

Later will we see how we can reduce this to ... in fact 0 java code lines. But the 3 lines will do for now.

Message Translation

Okay lets head back to the over goal of the integration. Looking at the EIP diagrams at the introduction page we need to be able to translate the incoming webservice to an email. Doing so we need to create the email body. When doing the message translation we could put up our sleeves and do it manually in pure java with a StringBuilder such as:

```
private String createMailBody(InputReportIncident parameters) {
    StringBuilder sb = new StringBuilder();
    sb.append("Incident ").append(parameters.getIncidentId());
    sb.append(" has been reported on the ").append(parameters.getIncidentDate());
    sb.append(" by ").append(parameters.getGivenName());
    sb.append(" ").append(parameters.getFamilyName());

    // and the rest of the mail body with more appends to the string builder

    return sb.toString();
}
```

But as always it is a hardcoded template for the mail body and the code gets kinda ugly if the mail message has to be a bit more advanced. But of course it just works out-of-the-box with just classes already in the JDK.

Lets use a template language instead such as Apache Velocity. As Camel have a component for Velocity integration we will use this component. Looking at the Component List overview we can see that camel-velocity component uses the artifactId **camel-velocity** so therefore we need to add this to the **pom.xml**

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-velocity</artifactId>
    <version>${camel-version}</version>
</dependency>
```

And now we have a Spring conflict as Apache CXF is dependent on Spring 2.0.8 and camel-velocity is dependent on Spring 2.5.5. To remedy this we could wrestle with the **pom.xml** with excludes settings in the dependencies or just bring in another dependency **camel-spring**:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
    <version>${camel-version}</version>
</dependency>
```

In fact camel-spring is such a vital part of Camel that you will end up using it in nearly all situations - we will look into how well Camel is seamless integration with Spring in part 3. For now its just another dependency.

We create the mail body with the Velocity template and create the file `src/main/resources/MailBody.vm`. The content in the **MailBody.vm** f ile is:

```
Incident $body.incidentId has been reported on the $body.incidentDate by
$body.givenName $body.familyName.

The person can be contact by:
- email: $body.email
- phone: $body.phone

Summary: $body.summary

Details:
$body.details

This is an auto generated email. You can not reply.
```

Letting Camel creating the mail body and storing it as a file is as easy as the following 3 code lines:

```
    private void generateEmailBodyAndStoreAsFile(InputReportIncident parameters) {
        // generate the mail body using velocity template
        // notice that we just pass in our POJO (= InputReportIncident) that we
        // got from Apache CXF to Velocity.
        Object response = template.sendBody("velocity:MailBody.vm", parameters);
        // Note: the response is a String and can be cast to String if needed

        // store the mail in a file
        String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
        template.sendBodyAndHeader("file://target/subfolder", response,
FileComponent.HEADER_FILE_NAME, filename);
    }
```

What is impressive is that we can just pass in our POJO object we got from Apache CXF to Velocity and it will be able to generate the mail body with this object in its context. Thus we don't need to prepare **anything** before we let Velocity loose and generate our mail body. Notice that the **te mplate** method returns a object with out response. This object contains the mail body as a String object. We can cast to String if needed.

If we run our unit test with `mvn test` we can in fact see that Camel has produced the file and we can type its content:

```
D:\demo\part-two>type target\subfolder\mail-incident-123.txt
Incident 123 has been reported on the 2008-07-16 by Claus Ibsen.

The person can be contact by:
- email: davsclaus@apache.org
- phone: +45 2962 7576

Summary: bla bla

Details:
more bla bla

This is an auto generated email. You can not reply.
```

First part of the solution

What we have seen here is actually what it takes to build the first part of the integration flow. Receiving a request from a webservice, transform it to a mail body and store it to a file, and return an OK response to the webservice. All possible within 10 lines of code. So lets wrap it up here is what it takes:

```
/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext camel;
    private ProducerTemplate template;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the camel context that is the "heart" of Camel
        camel = new DefaultCamelContext();

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer
for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // start Camel
        camel.start();
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // transform the request into a mail body
        Object mailBody = template.sendBody("velocity:MailBody.vm", parameters);

        // store the mail body in a file
        String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
        template.sendBodyAndHeader("file://target/subfolder", mailBody,
FileComponent.HEADER_FILE_NAME, filename);

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

}
```

Okay I missed by one, its in fact only **9 lines of java code and 2 fields**.

End of part 2

I know this is a bit different introduction to Camel to how you can start using it in your projects just as a plain java .jar framework that isn't invasive at all. I took you through the coding parts that requires 6 - 10 lines to send a message to an endpoint, buts it's important to show the Message Endpoint EIP pattern in action and how its implemented in Camel. Yes of course Camel also has to one liners that you can use, and will use in your projects for sending messages to endpoints. This part has been about good old plain java, nothing fancy with Spring, XML files, auto discovery, OGSi or other new technologies. I wanted to demonstrate the basic building blocks in Camel and how its setup in pure god old fashioned Java. There are plenty of eye catcher examples with one liners that does more than you can imagine - we will come there in the later parts.

Okay part 3 is about building the last pieces of the solution and now it gets interesting since we have to wrestle with the event driven consumer. Brew a cup of coffee, tug the kids and kiss the wife, for now we will have us some fun with the Camel. See you in part 3.

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part3

Part 3

Recap

Lets just recap on the solution we have now:

```
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext camel;
    private ProducerTemplate template;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the camel context that is the "heart" of Camel
        camel = new DefaultCamelContext();

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer
for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // start Camel
        camel.start();
    }

    /**
     * This is the last solution displayed that is the most simple
     */
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // transform the request into a mail body
        Object mailBody = template.sendBody("velocity:MailBody.vm", parameters);

        // store the mail body in a file
        String filename = "mail-incident-" + parameters.getIncidentId() + ".txt";
        template.sendBodyAndHeader("file://target/subfolder", mailBody,
FileComponent.HEADER_FILE_NAME, filename);

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

}
```

This completes the first part of the solution: receiving the message using webservice, transform it to a mail body and store it as a text file.
What is missing is the last part that polls the text files and send them as emails. Here is where some fun starts, as this requires usage of the Even t Driven Consumer EIP pattern to react when new files arrives. So lets see how we can do this in Camel. There is a saying: Many roads lead to Rome, and that is also true for Camel - there are many ways to do it in Camel.

Adding the Event Driven Consumer

We want to add the consumer to our integration that listen for new files, we do this by creating a private method where the consumer code lives.

We must register our consumer in Camel before its started so we need to add, and there fore we call the method **addMailSenderConsumer** in the constructor below:

```
    public ReportIncidentEndpointImpl() throws Exception {
        // create the camel context that is the "heart" of Camel
        camel = new DefaultCamelContext();

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer
for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // add the event driven consumer that will listen for mail files and process
them
        addMailSendConsumer();

        // start Camel
        camel.start();
    }
```

The consumer needs to be consuming from an endpoint so we grab the endpoint from Camel we want to consume. It's `file://target/subfolder`. Don't be fooled this endpoint doesn't have to 100% identical to the producer, i.e. the endpoint we used in the previous part to create and store the files. We could change the URL to include some options, and to make it more clear that it's possible we setup a delay value to 10 seconds, and the first poll starts after 2 seconds. This is done by adding `?consumer.delay=10000&consumer.initialDelay=2000` to the URL.

URL Configuration
The URL configuration in Camel endpoints is just like regular URL we know from the Internet. You use ? and & to set the options.

When we have the endpoint we can create the consumer (just as in part 1 where we created a producer}. Creating the consumer requires a Processor where we implement the java code what should happen when a message arrives. To get the mail body as a String object we can use the **get Body** method where we can provide the type we want in return.

Camel Type Converter
Why don't we just cast it as we always do in Java? Well the biggest advantage when you provide the type as a parameter you tell Camel what type you want and Camel can automatically convert it for you, using its flexible Type Converter mechanism. This is a great advantage, and you should try to use this instead of regular type casting.

Sending the email is still left to be implemented, we will do this later. And finally we must remember to start the consumer otherwise its not active and won't listen for new files.

```
    private void addMailSendConsumer() throws Exception {
        // Grab the endpoint where we should consume. Option - the first poll starts
after 2 seconds
        Endpoint endpint =
camel.getEndpoint("file://target/subfolder?consumer.initialDelay=2000");

        // create the event driven consumer
        // the Processor is the code what should happen when there is an event
        // (think it as the onMessage method)
        Consumer consumer = endpint.createConsumer(new Processor() {
            public void process(Exchange exchange) throws Exception {
                // get the mail body as a String
                String mailBody = exchange.getIn().getBody(String.class);

                // okay now we are read to send it as an email
                System.out.println("Sending email..." + mailBody);
            }
        });

        // star the consumer, it will listen for files
        consumer.start();
    }
```

Before we test it we need to be aware that our unit test is only catering for the first part of the solution, receiving the message with webservice, transforming it using Velocity and then storing it as a file - it doesn't test the Event Driven Consumer we just added. As we are eager to see it in action, we just do a common trick adding some sleep in our unit test, that gives our Event Driven Consumer time to react and print to System.out. We will later refine the test:

```
    public void testRendportIncident() throws Exception {
       ...

        OutputReportIncident out = client.reportIncident(input);
        assertEquals("Response code is wrong", "OK", out.getCode());

        // give the event driven consumer time to react
        Thread.sleep(10 * 1000);
    }
```

We run the test with `mvn clean test` and have eyes fixed on the console output.
During all the output in the console, we see that our consumer has been triggered, as we want.

```
2008-07-19 12:09:24,140 [mponent@1f12c4e] DEBUG FileProcessStrategySupport - Locking
the file: target\subfolder\mail-incident-123.txt ...
Sending email...Incident 123 has been reported on the 2008-07-16 by Claus Ibsen.

The person can be contact by:
- email: davsclaus@apache.org
- phone: +45 2962 7576

Summary: bla bla

Details:
more bla bla

This is an auto generated email. You can not reply.
2008-07-19 12:09:24,156 [mponent@1f12c4e] DEBUG FileConsumer - Done processing file:
target\subfolder\mail-incident-123.txt. Status is: OK
```

Sending the email

Sending the email requires access to a SMTP mail server, but the implementation code is very simple:

```
    private void sendEmail(String body) {
        // send the email to your mail server
        String url =
"smtp://someone@localhost?password=secret&to=incident@mycompany.com";
        template.sendBodyAndHeader(url, body, "subject", "New incident reported");
    }
```

And just invoke the method from our consumer:

```
    // okay now we are read to send it as an email
    System.out.println("Sending email...");
    sendEmail(mailBody);
    System.out.println("Email sent");
```

Unit testing mail

For unit testing the consumer part we will use a mock mail framework, so we add this to our **pom.xml**:

```
        <!-- unit testing mail using mock -->
        <dependency>
            <groupId>org.jvnet.mock-javamail</groupId>
            <artifactId>mock-javamail</artifactId>
            <version>1.7</version>
            <scope>test</scope>
        </dependency>
```

Then we prepare our integration to run with or without the consumer enabled. We do this to separate the route into the two parts:

- receive the webservice, transform and save mail file and return OK as repose
- the consumer that listen for mail files and send them as emails

So we change the constructor code a bit:

```
    public ReportIncidentEndpointImpl() throws Exception {
        init(true);
    }

    public ReportIncidentEndpointImpl(boolean enableConsumer) throws Exception {
        init(enableConsumer);
    }

    private void init(boolean enableConsumer) throws Exception {
        // create the camel context that is the "heart" of Camel
        camel = new DefaultCamelContext();

        // get the ProducerTemplate thst is a Spring'ish xxxTemplate based producer
for very
        // easy sending exchanges to Camel.
        template = camel.createProducerTemplate();

        // add the event driven consumer that will listen for mail files and process
them
        if (enableConsumer) {
            addMailSendConsumer();
        }

        // start Camel
        camel.start();
    }
```

Then remember to change the **ReportIncidentEndpointTest** to pass in **false** in the `ReportIncidentEndpointImpl` constructor.
And as always run `mvn clean test` to be sure that the latest code changes works.

Adding new unit test

We are now ready to add a new unit test that tests the consumer part so we create a new test class that has the following code structure:

```
/**
 * Plain JUnit test of our consumer.
 */
public class ReportIncidentConsumerTest extends TestCase {

    private ReportIncidentEndpointImpl endpoint;

    public void testConsumer() throws Exception {
        // we run this unit test with the consumer, hence the true parameter
        endpoint = new ReportIncidentEndpointImpl(true);
   }

}
```

As we want to test the consumer that it can listen for files, read the file content and send it as an email to our mailbox we will test it by asserting that we receive 1 mail in our mailbox and that the mail is the one we expect. To do so we need to grab the mailbox with the mockmail API. This is done as simple as:

```
    public void testConsumer() throws Exception {
        // we run this unit test with the consumer, hence the true parameter
        endpoint = new ReportIncidentEndpointImpl(true);

        // get the mailbox
        Mailbox box = Mailbox.get("incident@mycompany.com");
        assertEquals("Should not have mails", 0, box.size());
```

How do we trigger the consumer? Well by creating a file in the folder it listen for. So we could use plain java.io.File API to create the file, but wait isn't there an smarter solution? ... yes Camel of course. Camel can do amazing stuff in one liner codes with its ProducerTemplate, so we need to get a hold of this baby. We expose this template in our ReportIncidentEndpointImpl but adding this getter:

```
    protected ProducerTemplate getTemplate() {
        return template;
    }
```

Then we can use the template to create the file in **one code line**:

```
        // drop a file in the folder that the consumer listen
        // here is a trick to reuse Camel! so we get the producer template and just
        // fire a message that will create the file for us

endpoint.getTemplate().sendBodyAndHeader("file://target/subfolder?append=false",
"Hello World",
            FileComponent.HEADER_FILE_NAME, "mail-incident-test.txt");
```

Then we just need to wait a little for the consumer to kick in and do its work and then we should assert that we got the new mail. Easy as just:

```
        // let the consumer have time to run
        Thread.sleep(3 * 1000);

        // get the mock mailbox and check if we got mail ;)
        assertEquals("Should have got 1 mail", 1, box.size());
        assertEquals("Subject wrong", "New incident reported",
box.get(0).getSubject());
        assertEquals("Mail body wrong", "Hello World", box.get(0).getContent());
    }
```

The final class for the unit test is:

```
/**
 * Plain JUnit test of our consumer.
 */
public class ReportIncidentConsumerTest extends TestCase {

    private ReportIncidentEndpointImpl endpoint;

    public void testConsumer() throws Exception {
        // we run this unit test with the consumer, hence the true parameter
        endpoint = new ReportIncidentEndpointImpl(true);

        // get the mailbox
        Mailbox box = Mailbox.get("incident@mycompany.com");
        assertEquals("Should not have mails", 0, box.size());

        // drop a file in the folder that the consumer listen
        // here is a trick to reuse Camel! so we get the producer template and just
        // fire a message that will create the file for us

endpoint.getTemplate().sendBodyAndHeader("file://target/subfolder?append=false",
"Hello World",
            FileComponent.HEADER_FILE_NAME, "mail-incident-test.txt");

        // let the consumer have time to run
        Thread.sleep(3 * 1000);

        // get the mock mailbox and check if we got mail ;)
        assertEquals("Should have got 1 mail", 1, box.size());
        assertEquals("Subject wrong", "New incident reported",
box.get(0).getSubject());
        assertEquals("Mail body wrong", "Hello World", box.get(0).getContent());
    }

}
```

End of part 3

Okay we have reached the end of part 3. For now we have only scratched the surface of what Camel is and what it can do. We have introduced Camel into our integration piece by piece and slowly added more and more along the way. And the most important is: **you as the developer never lost control**. We hit a sweet spot in the webservice implementation where we could write our java code. Adding Camel to the mix is just to

use it as a regular java code, nothing magic. We were in control of the flow, we decided when it was time to translate the input to a mail body, we decided when the content should be written to a file. This is very important to not lose control, that the bigger and heavier frameworks tend to do. No names mentioned, but boy do developers from time to time dislike these elephants. And Camel is **no elephant**.

I suggest you download the samples from part 1 to 3 and try them out. It is great basic knowledge to have in mind when we look at some of the features where Camel really excel - **the routing domain language**.

From part 1 to 3 we touched concepts such as::

- Endpoint
- URI configuration
- Consumer
- Producer
- Event Driven Consumer
- Component
- CamelContext
- ProducerTemplate
- Processor
- Type Converter

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part4

Part 4

Introduction

This section is about regular Camel. The examples presented here in this section is much more in common of all the examples we have in the Camel documentation.

If you have been reading the previous 3 parts then, this quote applies:

> *you must unlearn what you have learned*
> Master Yoda, Star Wars IV

So we start all over again! 

Routing

Camel is particular strong as a light-weight and agile **routing** and **mediation** framework. In this part we will introduce the **routing** concept and how we can introduce this into our solution.
Looking back at the figure from the Introduction page we want to implement this routing. Camel has support for expressing this routing logic using Java as a DSL (Domain Specific Language). In fact Camel also has DSL for XML and Scala. In this part we use the Java DSL as its the most powerful and all developers know Java. Later we will introduce the XML version that is very well integrated with Spring.

Before we jump into it, we want to state that this tutorial is about **Developers not loosing control**. In my humble experience one of the key fears of developers is that they are forced into a tool/framework where they loose control and/or power, and the possible is now impossible. So in this part we stay clear with this vision and our starting point is as follows:

- We have generated the webservice source code using the CXF wsdl2java generator and we have our ReportIncidentEndpointImpl.java file where we as a Developer feels home and have the power.

So the starting point is:

```
/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    /**
     * This is the last solution displayed that is the most simple
     */
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // WE ARE HERE !!!
        return null;
    }

}
```

Yes we have a simple plain Java class where we have the implementation of the webservice. The cursor is blinking at the WE ARE HERE block and this is where we feel home. More or less any Java Developers have implemented webservices using a stack such as: Apache AXIS, Apache CXF or some other quite popular framework. They all allow the developer to be in control and implement the code logic as plain Java code. Camel of course doesn't enforce this to be any different. Okay the boss told us to implement the solution from the figure in the Introduction page and we are now ready to code.

RouteBuilder

RouteBuilder is the hearth in Camel of the Java DSL routing. This class does all the heavy lifting of supporting EIP verbs for end-users to express the routing. It does take a little while to get settled and used to, but when you have worked with it for a while you will enjoy its power and realize it is in fact a little language inside Java itself. Camel is the **only** integration framework we are aware of that has Java DSL, all the others are usually **only** XML based.

As an end-user you usually use the **RouteBuilder** as of follows:

- create your own Route class that extends **RouteBuilder**
- implement your routing DSL in the **configure** method

So we create a new class ReportIncidentRoutes and implement the first part of the routing:

```
import org.apache.camel.builder.RouteBuilder;

public class ReportIncidentRoutes extends RouteBuilder {

    public void configure() throws Exception {
        // direct:start is a internal queue to kick-start the routing in our example
        // we use this as the starting point where you can send messages to
direct:start
        from("direct:start")
            // to is the destination we send the message to our velocity endpoint
            // where we transform the mail body
            .to("velocity:MailBody.vm");
    }

}
```

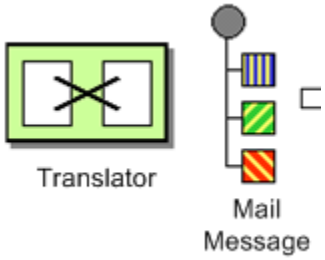
What to notice here is the **configure** method. Here is where all the action is. Here we have the Java DSL langauge, that is expressed using the **fluent builder syntax** that is also known from Hibernate when you build the dynamic queries etc. What you do is that you can stack methods separating with the dot.

In the example above we have a very common routing, that can be distilled from pseudo verbs to actual code with:

- from A to B
- From Endpoint A To Endpoint B
- from("endpointA").to("endpointB")
- from("direct:start").to("velocity:MailBody.vm");

from("direct:start") is the consumer that is kick-starting our routing flow. It will wait for messages to arrive on the direct queue and then dispatch the message.
to("velocity:MailBody.vm") is the producer that will receive a message and let Velocity generate the mail body response.

So what we have implemented so far with our ReportIncidentRoutes RouteBuilder is this part of the picture:



Adding the RouteBuilder

Now we have our RouteBuilder we need to add/connect it to our CamelContext that is the hearth of Camel. So turning back to our webservice

implementation class ReportIncidentEndpointImpl we add this constructor to the code, to create the CamelContext and add the routes from our route builder and finally to start it.

```
    private CamelContext context;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the context
        context = new DefaultCamelContext();

        // append the routes to the context
        context.addRoutes(new ReportIncidentRoutes());

        // at the end start the camel context
        context.start();
    }
```

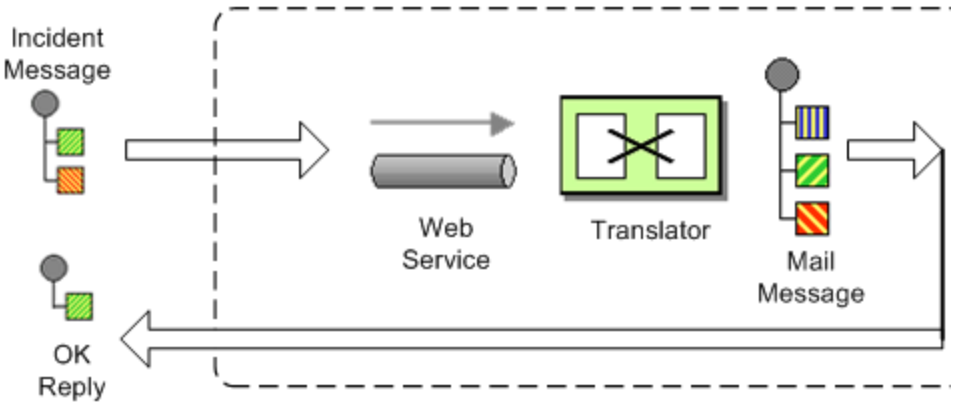
Okay how do you use the routes then? Well its just as before we use a ProducerTemplate to send messages to Endpoints, so we just send to the **direct:start** endpoint and it will take it from there.
So we implement the logic in our webservice operation:

```
    /**
     * This is the last solution displayed that is the most simple
     */
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        Object mailBody = context.createProducerTemplate().sendBody("direct:start",
parameters);
        System.out.println("Body:" + mailBody);

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

Notice that we get the producer template using the **createProducerTemplate** method on the CamelContext. Then we send the input parameters to the **direct:start** endpoint and it will route it **to** the velocity endpoint that will generate the mail body. Since we use **direct** as the consumer endpoint (=from) and its a **synchronous** exchange we will get the response back from the route. And the response is of course the output from the velocity endpoint.

About creating ProducerTemplate
In the example above we create a new `ProducerTemplate` when the `reportIncident` method is invoked. However in reality you should only create the template once and re-use it. See this FAQ entry.

We have now completed this part of the picture:



Unit testing

Now is the time we would like to unit test what we got now. So we call for camel and its great test kit. For this to work we need to add it to the pom.xml

```
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-core</artifactId>
            <version>1.4.0</version>
            <scope>test</scope>
            <type>test-jar</type>
        </dependency>
```

After adding it to the pom.xml you should refresh your Java Editor so it pickups the new jar. Then we are ready to create out unit test class. We create this unit test skeleton, where we **extend** this class `ContextTestSupport`

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

import org.apache.camel.ContextTestSupport;
import org.apache.camel.builder.RouteBuilder;

/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest extends ContextTestSupport {

}
```

`ContextTestSupport` is a supporting unit test class for much easier unit testing with Apache Camel. The class is extending JUnit TestCase itself so you get all its glory. What we need to do now is to somehow tell this unit test class that it should use our route builder as this is the one we gonna test. So we do this by implementing the `createRouteBuilder` method.

```
    @Override
    protected RouteBuilder createRouteBuilder() throws Exception {
        return new ReportIncidentRoutes();
    }
```

That is easy just return an instance of our route builder and this unit test will use our routes.

It is quite common in Camel itself to unit test using routes defined as an anonymous inner class, such as illustrated below:

```
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        public void configure() throws Exception {
            // TODO: Add your routes here, such as:
            from("jms:queue:inbox").to("file://target/out");
        }
    };
}
```

The same technique is of course also possible for end-users of Camel to create parts of your routes and test them separately in many test classes.
However in this tutorial we test the real route that is to be used for production, so we just return an instance of the real one.

We then code our unit test method that sends a message to the route and assert that its transformed to the mail body using the Velocity template.

```
    public void testTransformMailBody() throws Exception {
        // create a dummy input with some input data
        InputReportIncident parameters = createInput();

        // send the message (using the sendBody method that takes a parameters as the
input body)
        // to "direct:start" that kick-starts the route
        // the response is returned as the out object, and its also the body of the
response
        Object out = context.createProducerTemplate().sendBody("direct:start",
parameters);

        // convert the response to a string using camel converters. However we could
also have casted it to
        // a string directly but using the type converters ensure that Camel can
convert it if it wasn't a string
        // in the first place. The type converters in Camel is really powerful and you
will later learn to
        // appreciate them and wonder why its not build in Java out-of-the-box
        String body = context.getTypeConverter().convertTo(String.class, out);

        // do some simple assertions of the mail body
        assertTrue(body.startsWith("Incident 123 has been reported on the 2008-07-16
by Claus Ibsen."));
    }

    /**
     * Creates a dummy request to be used for input
     */
    protected InputReportIncident createInput() {
        InputReportIncident input = new InputReportIncident();
        input.setIncidentId("123");
        input.setIncidentDate("2008-07-16");
        input.setGivenName("Claus");
        input.setFamilyName("Ibsen");
        input.setSummary("bla bla");
        input.setDetails("more bla bla");
        input.setEmail("davsclaus@apache.org");
        input.setPhone("+45 2962 7576");
        return input;
    }
```

Adding the File Backup

The next piece of puzzle that is missing is to store the mail body as a backup file. So we turn back to our route and the EIP patterns. We use the Pipes and Filters pattern here to chain the routing as:

```
    public void configure() throws Exception {
        from("direct:start")
            .to("velocity:MailBody.vm")
            // using pipes-and-filters we send the output from the previous to the
next
            .to("file://target/subfolder");
     }
```

Notice that we just add a 2nd **.to** on the newline. Camel will default use the Pipes and Filters pattern here when there are multi endpoints chained liked this. We could have used the **pipeline** verb to let out stand out that its the Pipes and Filters pattern such as:

```
        from("direct:start")
            // using pipes-and-filters we send the output from the previous to the
next
            .pipeline("velocity:MailBody.vm", "file://target/subfolder");
```

But most people are using the multi **.to** style instead.

We re-run out unit test and verifies that it still passes:

```
Running org.apache.camel.example.reportincident.ReportIncidentRoutesTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.157 sec
```

But hey we have added the file *producer* endpoint and thus a file should also be created as the backup file. If we look in the `target/subfolder` we can see that something happened.
On my humble laptop it created this folder: **target\subfolder\ID-claus-acer**. So the file producer create a sub folder named `ID-claus-acer` wh at is this? Well Camel auto generates an unique filename based on the unique message id if not given instructions to use a fixed filename. In fact it creates another sub folder and name the file as: target\subfolder\ID-claus-acer\3750-1219148558921\1-0 where 1-0 is the file with the mail body. What we want is to use our own filename instead of this auto generated filename. This is archived by adding a header to the message with the filename to use. So we need to add this to our route and compute the filename based on the message content.

Setting the filename

For starters we show the simple solution and build from there. We start by setting a constant filename, just to verify that we are on the right path, to instruct the file producer what filename to use. The file producer uses a special header `FileComponent.HEADER_FILE_NAME` to set the filename.

What we do is to send the header when we "kick-start" the routing as the header will be propagated from the direct queue to the file producer. What we need to do is to use the `ProducerTemplate.sendBodyAndHeader` method that takes **both** a body and a header. So we change out webservice code to include the filename also:

```
    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // create the producer template to use for sending messages
        ProducerTemplate producer = context.createProducerTemplate();
        // send the body and the filename defined with the special header key
        Object mailBody = producer.sendBodyAndHeader("direct:start", parameters,
FileComponent.HEADER_FILE_NAME, "incident.txt");
        System.out.println("Body:" + mailBody);

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

However we could also have used the route builder itself to configure the constant filename as shown below:

```
    public void configure() throws Exception {
        from("direct:start")
            .to("velocity:MailBody.vm")
            // set the filename to a constant before the file producer receives the
message
            .setHeader(FileComponent.HEADER_FILE_NAME, constant("incident.txt"))
            .to("file://target/subfolder");
     }
```

But Camel can be smarter and we want to dynamic set the filename based on some of the input parameters, how can we do this?
Well the obvious solution is to compute and set the filename from the webservice implementation, but then the webservice implementation has such logic and we want this decoupled, so we could create our own POJO bean that has a method to compute the filename. We could then instruct the routing to invoke this method to get the computed filename. This is a string feature in Camel, its Bean binding. So lets show how this can be done:

Using **Bean Language** to compute the filename

First we create our plain java class that computes the filename, and it has 100% no dependencies to Camel what so ever.

```
/**
 * Plain java class to be used for filename generation based on the reported incident
 */
public class FilenameGenerator {

    public String generateFilename(InputReportIncident input) {
        // compute the filename
        return "incident-" + input.getIncidentId() + ".txt";
    }

}
```

The class is very simple and we could easily create unit tests for it to verify that it works as expected. So what we want now is to let Camel invoke this class and its generateFilename with the input parameters and use the output as the filename. Pheeeww is this really possible out-of-the-box in Camel? Yes it is. So lets get on with the show. We have the code that computes the filename, we just need to call it from our route using the Be an Language:

```
    public void configure() throws Exception {
        from("direct:start")
            // set the filename using the bean language and call the FilenameGenerator
class.
            // the 2nd null parameter is optional methodname, to be used to avoid
ambiguity.
            // if not provided Camel will try to figure out the best method to invoke,
as we
            // only have one method this is very simple
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, null))
            .to("velocity:MailBody.vm")
            .to("file://target/subfolder");
    }
```

Notice that we use the **bean** language where we supply the class with our bean to invoke. Camel will instantiate an instance of the class and invoke the suited method. For completeness and ease of code readability we add the method name as the 2nd parameter

```
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
```

Then other developers can understand what the parameter is, instead of `null`.

Now we have a nice solution, but as a sidetrack I want to demonstrate the Camel has other languages out-of-the-box, and that scripting language is a first class citizen in Camel where it etc. can be used in content based routing. However we want it to be used for the filename generation.

⚠ Using a script language to set the filename

We could do as in the previous parts where we send the computed filename as a message header when we "kick-start" the route. But we want to learn new stuff so we look for a different solution using some of Camels many Languages. As OGNL is a favorite language of mine (used by WebWork) so we pick this baby for a Camel ride. For starters we must add it to our pom.xml:

```
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-ognl</artifactId>
            <version>${camel-version}</version>
        </dependency>
```

And remember to refresh your editor so you got the new .jars.
We want to construct the filename based on this syntax: `mail-incident-#ID#.txt` where #ID# is the incident id from the input parameters. As OGNL is a language that can invoke methods on bean we can invoke the `getIncidentId()` on the message body and then concat it with the fixed pre and postfix strings.

In OGNL glory this is done as:

```
"'mail-incident-' + request.body.incidentId + '.txt'"
```

where `request.body.incidentId` computes to:

- **request** is the IN message. See the OGNL for other predefined objects available
- **body** is the body of the in message
- **incidentId** will invoke the `getIncidentId()` method on the body.
  The rest is just more or less regular plain code where we can concat strings.

Now we got the expression to dynamic compute the filename on the fly we need to set it on our route so we turn back to our route, where we can add the OGNL expression:

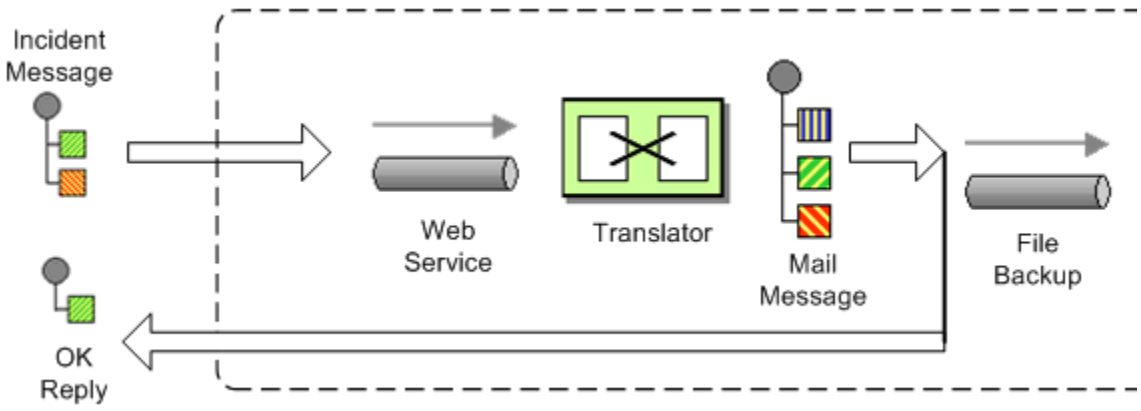
```
    public void configure() throws Exception {
        from("direct:start")
            // we need to set the filename and uses OGNL for this
            .setHeader(FileComponent.HEADER_FILE_NAME,
OgnlExpression.ognl("'mail-incident-' + request.body.incidentId + '.txt'"))
            // using pipes-and-filters we send the output from the previous to
the next
            .pipeline("velocity:MailBody.vm", "file://target/subfolder");
    }
```

And since we are on Java 1.5 we can use the static import of **ognl** so we have:

```
import static org.apache.camel.language.ognl.OgnlExpression.ognl;
...
    .setHeader(FileComponent.HEADER_FILE_NAME, ognl("'mail-incident-' +
request.body.incidentId + '.txt'"))
```

Notice the import static also applies for all the other languages, such as the Bean Language we used previously.

Whatever worked for you we have now implemented the backup of the data files:



Sending the email

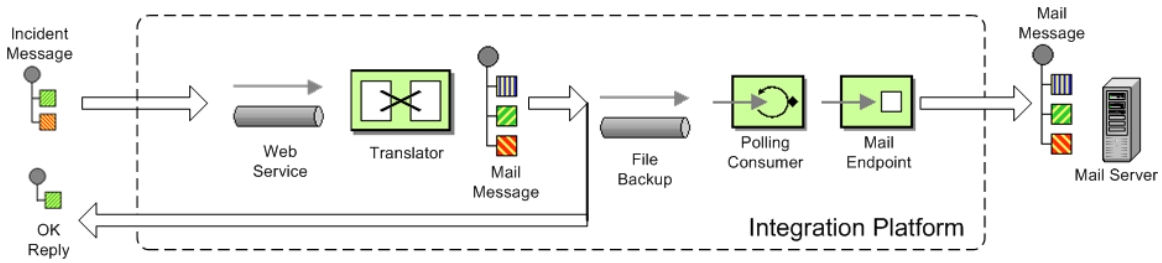
What we need to do before the solution is completed is to actually send the email with the mail body we generated and stored as a file. In the previous part we did this with a File consumer, that we manually added to the CamelContext. We can do this quite easily with the routing.

```
import org.apache.camel.builder.RouteBuilder;

public class ReportIncidentRoutes extends RouteBuilder {

    public void configure() throws Exception {
        // first part from the webservice -> file backup
        from("direct:start")
            .setHeader(FileComponent.HEADER_FILE_NAME, bean(FilenameGenerator.class,
"generateFilename"))
            .to("velocity:MailBody.vm")
            .to("file://target/subfolder");

        // second part from the file backup -> send email
        from("file://target/subfolder")
            // set the subject of the email
            .setHeader("subject", constant("new incident reported"))
            // send the email
            .to("smtp://someone@localhost?password=secret&to=incident@mycompany.com");
    }

}
```

The last 3 lines of code does all this. It adds a file consumer **from("file://target/subfolder")**, sets the mail subject, and finally send it as an email.

The DSL is really powerful where you can express your routing integration logic.
So we completed the last piece in the picture puzzle with just 3 lines of code.

We have now completed the integration:



Conclusion

We have just briefly touched the **routing** in Camel and shown how to implement them using the **fluent builder** syntax in Java. There is much more to the routing in Camel than shown here, but we are learning step by step. We continue in part 5. See you there.

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part5

Part 5

... Continued from Part 4

We continue from part 4 where we have the routing in place. However as you might have noticed we aren't quiet there yet with a nice solution, we are still coding to much. In this part we will look into to address these two concerns:

- Starting Camel automatically
- Using CXF directly

Starting Camel automatically

Our current deployment model is as a war and we have the `web.xml` to help start things. Well in fact we will leverage Spring to start the world 😀 . We use it's ContextListener

```
 <!-- the listener that kick-starts Spring -->
 <listener>

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

Then we need a standard Spring XML file so we create a new file in `src/main/resources` and name it **camel-config.xml**. Before we start editing this XML file we need to link to it from the web.xml file. So we add this snippet to the web.xml:

```
 <!-- location of spring xml files -->
 <context-param>
     <param-name>contextConfigLocation</param-name>
            <param-value>classpath:camel-config.xml</param-value>
 </context-param>
```

Now we are ready to edit the **camel-config.xml** file that is a standard Spring XML bean file. So you can add standard spring beans and whatnot you like to do and can do with Spring.

```
<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>
```

Now we are nearly there, we just need to add Camel to the Spring XML file, so Spring knows Camel exists and can start it. First we need to add Camel to the schema location in the top of the XML file.

```
      ...
      xsi:schemaLocation="
           http://activemq.apache.org/camel/schema/spring
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
           http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

Now we are ready to let Spring and Camel work together. What we need to do is adding a CamelContext to the Spring XML file. Camel ships with a CamelContextFactoryBean that is a Spring factory bean we should use for creating and initializing the SpringCamelContext. SpringCamelContext is extending CamelContext to be Spring aware so Camel and Spring can work nicely together. For instance the Registry will now use Spring bean lookup. So any spring bean can now easily be lookup and used from Camel. Well back to today's lesson. So we can create a SpringCamelContext using the factory bean as illustrated below:

```
  <bean id="camel" class="org.apache.camel.spring.CamelContextFactoryBean"/>
```

However this is not used very often as Spring has support for custom namespace, so Camel has a CamelNamespaceHandler so we can create Camel using nice XML syntax as:

```
   <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
      ...
   </camelContext>
```

Adding route builder

Now we have Camel integrated but we still need to add our route bulder that we did manually from the javacode as:

```
   // append the routes to the context
   context.addRoutes(new ReportIncidentRoutes());
```

There are two solutions to this

- using spring bean
- package scanning

Using a spring bean we just declare the route builder using a regular spring bean:

```
  <bean id="myrouter"
class="org.apache.camel.example.reportincident.ReportIncidentRoutes"/>
```

And then we can refer to it from our CamelContext:

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
   <routeBuilderRef ref="myrouter"/>
</camelContext>
```

So now when Spring start's it will read the **camel-context.xml** file and thus also start Camel as well. As SpringCamelContext is spring lifecycle event aware, Camel will also shutdown when Spring is shutting down. So when you stop the web application Spring will notify this and Camel is also shutdown nice and properly. So as an end user no need to worry.

The package scanning solution is for convenience to refer to a java package and Camel will scan all classes within this package for RouteBuilder classes. If using this then you dont need to declare your route builder as a Spring bean. So the XML can be reduced to.

```
<camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/spring">
   <package>org.apache.camel.example.reportincident</package>
</camelContext>
```

Using CXF directly

Now we have seen how you can leverage Spring to start Camel, in fact it handles the lifecycle of Camel, so you can say Camel is embedded with Spring in your application.

From the very start of this tutorial we have used CXF as the webservice framework and we haven't integrated it directly with Camel as it can do out-of-the-box. Camel ships with a **camel-cxf** component for integrating CXF directly within Camel routing. In our tutorial we are exposing a webservice so we want continue to do this. Before we continue let's recap at what the webservice implementation we have from part 4

```
/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext context;

    public ReportIncidentEndpointImpl() throws Exception {
        // create the context
        context = new DefaultCamelContext();

        // append the routes to the context
        context.addRoutes(new ReportIncidentRoutes());

        // at the end start the camel context
        context.start();
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // create the producer template to use for sending messages
        ProducerTemplate producer = context.createProducerTemplate();
        // send the body and the filename defined with the special header key
        Object mailBody = producer.sendBody("direct:start", parameters);
        System.out.println("Body:" + mailBody);

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

}
```

We have already seen how we can get Spring starting Camel so the constructor method can be removed. What next is that the CamelContext needed in this code should be the one from our **camel-context.xml** file. So we change the code to use a plain setter injection (we can use Spring annotations and whatelse but we keep it simple with a setter):

```
/**
 * The webservice we have implemented.
 */
public class ReportIncidentEndpointImpl implements ReportIncidentEndpoint {

    private CamelContext context;

    public void setCamelContext(CamelContext context) {
        this.context = context;
    }

    public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // create the producer template to use for sending messages
        ProducerTemplate producer = context.createProducerTemplate();
        // send the body and the filename defined with the special header key
        Object mailBody = producer.sendBody("direct:start", parameters);
        System.out.println("Body:" + mailBody);

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

}
```

And then we need to instruct Spring to set this property. Turning back to cxf-config.xml from part 4 we can add a reference to our camel context

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

    <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-servlet.xml"/>

    <!-- implementation of the webservice, and we refer to our camel context with the
id = camel from camel-context.xml -->
    <bean id="reportIncidentEndpoint"
class="org.apache.camel.example.reportincident.ReportIncidentEndpointImpl">
       <property name="context" ref="camel"/>
    </bean>

    <!-- export the webservice using jaxws -->
    <jaxws:endpoint id="reportIncident"
                    implementor="#reportIncidentEndpoint"
                    address="/incident"
                    wsdlLocation="/WEB-INF/wsdl/report_incident.xml"
                    endpointName="s:ReportIncidentPort"
                    serviceName="s:ReportIncidentService"
                    xmlns:s="http://reportincident.example.camel.apache.org"/>
</beans>
```

So now we have two spring XML files

- cxf-config.xml
- camel-config.xml

And since **cxf-config.xml** is dependent on **camel-config.xml** we need to have correct ordering in our web.xml where we have defined the XML files to load by Spring. So we set the **camel-config.xml** before the **cxf-config.xml** so Spring have created the SpringCamelContext and registered it in its registry with the id = camel.

```
    <!-- location of spring xml files -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:camel-config.xml</param-value>
        <param-value>classpath:cxf-config.xml</param-value>
    </context-param>
```

Sidenote on spring XML files
The solution presented here with two spring XML files (cxf-config and camel-config) that is pendent on each other and thus has to be ordered can also be done using a different solution. You can for instance add an **import** in cxf-config and only have the cxf-config listed in web.xml. Another solution is to merge the two files into one combined file. Yes you can add the camelContext in the cxf-config file.

But hey this isn't using CXF directly in the routing? Yes it's not but I wanted to show the halfway solution as well. What we have now is having Spring creating and handling lifecycle of Camel and showing how you can inject CamelContext using standard Spring into whatever code you have. This is very powerful as you can use the solution that you (or your team) already master and is familiar with. If they have Spring experience

then the IoC principle of injecting resources is of course also possible with Camel as well. In fact it's a best practice principle. Later you will learn that you can inject other Camel types such as Endpoint, ProducerTemplate as well.

Using the camel-cxf component

Okay let's continue and try to integrate CXF directly into our routing in Camel. This can be a bit more tricky than at first sight. Our goal is to avoid implementing the `ReportIncidentEndpoint` as we did with our code in `ReportIncidentEndpointImpl` (see above). Camel should be able to handle this automatically and integrate directly within our route.

Before we started our routing using the Direct endpoint with `"direct:start"`. We should replace this with the CXF endpoint.

But before going to far we have to make a few adjustments to the .wsdl file and it's location in our project folder. We move **report_incident.wsdl** f rom src/main/webapp/WEB-INF/wsdl* to **src/main/resources** as we want to be able to refer to it from within our route builder from Java code and thus it should be on the classpath for easy access. Secondly we have upgrade the CXF to a newer version and it identified a minor issue with the .wsdl file itself. We have to give the complexType a name otherwise we get some JAXB error.

So the .wsdl should be changed from:

```
    <xs:element name="inputReportIncident">
        <xs:complexType>
```

To include a name attribute of the complex types:

```
    <xs:element name="inputReportIncident">
        <xs:complexType name="inputReportIncident">
```

Using CXF endpoint

Okay now we are ready to turn our attention to using CXF directly in Camel routing. So we zap `ReportIncidentEndpointImpl` as we no longer need this code. So what's left is:

- `FilenameGenerator.java`
- `ReportIncidentRoutes.java`

  And that is all what's needed, well for now... 

The goal is to replace the previous staring endpoint ("direct:start") to the new starting CXF endpoint.

CXF endpoint can be configured in either or both CXF spring configuration file or/and in the route directly. It accepts one parameter and others are optional. The parameter it **must** have is the service class. The service class is the interface for the WSDL operation's. As we have the **wsdl2j ava** goal to generate this class for us, we have it already as `org.apache.camel.example.reportincident.ReportIncidentEndpoint`.

The other parameter we will provide is the url to the .wsdl file. And finally we must provide the http address we expose the webservice at. The URI is therefore:

```
        // endpoint to our CXF webservice
        String cxfEndpoint =
"cxf://http://localhost:8080/part-five/webservices/incident"
                +
"?serviceClass=org.apache.camel.example.reportincident.ReportIncidentEndpoint"
                + "&wsdlURL=report_incident.wsdl";
```

Then we can replace "direct:start" with our cxf endpoint instead, so it's:

```
        from(cxfEndpoint)...
```

The next issue you might now have guessed is that before (in part 4) we did a traditional codeing style to start a task and return a response to the caller in the method:

```
  public OutputReportIncident reportIncident(InputReportIncident parameters) {
        // create the producer template to use for sending messages
        ProducerTemplate producer = context.createProducerTemplate();
        // send the body and the filename defined with the special header key
        Object mailBody = producer.sendBody("direct:start", parameters);
        System.out.println("Body:" + mailBody);

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

As you can see the method `reportIncident` is invoked with the webservice input parameters and we return the response to to the webservice from the method. But our situation now is that we don't have this method anymore. So how do we return a response to the webservice?

Houston we have a problem! Well of course not but the mindset have to be changed slightly to understand the routing concept, and how it works. So let's step back a bit. What we have here is a webservice that we expose. And our webservice is synchronous request/response based so the caller waits for a response. This is a InOut Message Exchange Pattern. Camel will default use InOut for webservices so we don't need to specify this explicitly. When we have a InOut pattern then Camel will return the response to the original caller when the routes ends. Looking at our route we have:

```
        from(cxfEndpoint)
            // then set the file name using the FilenameGenerator bean
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
            // transform the message using velocity to generate the mail message
            .to("velocity:MailBody.vm")
            // and store the file
            .to("file://target/subfolder")
```

When the route ends after the file endpoint has been processed Camel will return the OUT message to the original caller (the caller of the webservice). However our route currently as it stands have not set any OUT message, so this is what we need to do, transforming (Message Translator EIP) the message into the response. We will therefore use a processor where we have 100% control in the Java code to set the response.

```
        public void process(Exchange exchange) throws Exception {
            // the response we want to send
            OutputReportIncident OK = new OutputReportIncident();
            OK.setCode("0");

            // set the response on the OUT message as we use InOut
            exchange.getOut().setBody(OK);
        }
```

And with the route:

```
        // first part from the webservice -> file backup
        from(cxfEndpoint)
            // then set the file name using the FilenameGenerator bean
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
            // transform the message using velocity to generate the mail message
            .to("velocity:MailBody.vm")
            // and store the file
            .to("file://target/subfolder")
            // return OK as response
            .process(new Processor() {
                public void process(Exchange exchange) throws Exception {
                    // the response we want to send
                    OutputReportIncident OK = new OutputReportIncident();
                    OK.setCode("0");

                    // set the response on the OUT message as we use InOut
                    exchange.getOut().setBody(OK);
                }
            });
```

The route using the inlined processor is a bit ugly as we have high level routing logic combined with low level java code. First of all I wanted to show how flexible Camel is, allowing use as a developer to always be in control and can use Java code for whatever you needs is. First of all we could move the code into a inner class and just refer to it:

```
    private static class OKResponseProcessor implements Processor {
        public void process(Exchange exchange) throws Exception {
            // the response we want to send
            OutputReportIncident OK = new OutputReportIncident();
            OK.setCode("0");

            // set the response on the OUT message as we use InOut
            exchange.getOut().setBody(OK);
        }
    }
```

And then out route is much nicer:

```
        // first part from the webservice -> file backup
        from(cxfEndpoint)
            // then set the file name using the FilenameGenerator bean
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
            // transform the message using velocity to generate the mail message
            .to("velocity:MailBody.vm")
            // and store the file
            .to("file://target/subfolder")
            // return OK as response
            .process(new OKResponseProcessor());
```

Since our response is static and we don't need to any code logic to set it we can use the **transform** DSL in the route to set a constant OUT message. So we refactor the code a bit to loose the processor. First we define the OK response as:

```
        // webservice response for OK
        OutputReportIncident OK = new OutputReportIncident();
        OK.setCode("0");
```

And then we can refer to it in the route as a constant expression:

```
    // return OK as response
    .transform(constant(OK));
```

Important issue regarding using CXF endpoints in Camel

Now we are nearly there, there is an important issue left with using CXF endpoints in Camel. In part 4 we started the route by sending the InputReportIncident object containing the webservice input. Now we are using CXF endpoints directly in our routing so its a CxfExchange that is created and passed in the routing. CxfExchange stores the payload in a CXF holder class `org.apache.cxf.message.MessageContentsList`. So to be able to get our InputReportIncident class we need to get this object from the holder class. For this we show how it's done in Java using a processor, then later we show a nicer solution.

```
public void process(final Exchange exchange) {
    // Get the parameter list
    List parameter = exchange.getIn().getBody(List.class);
    // Get the first object in the list that is our InputReportIncident
    Object input = parameter.get(0);
    // replace with our input
    exchange.getOut().setBody(input);
}
```

Well this isn't the nicest code, but again we want to show how it's done using plain Java, that is actually how Camel also can assist you in this nicer solution - we simply convert the body to the expected type using **convertBodyTo**. This is an important feature in Camel and you can use it for other situations as well.

```
        // first part from the webservice -> file backup
        from(cxfEndpoint)
            // we need to convert the CXF payload to InputReportIncident that
FilenameGenerator and velocity expects
            .convertBodyTo(InputReportIncident.class)
            // then set the file name using the FilenameGenerator bean
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
            // transform the message using velocity to generate the mail message
            .to("velocity:MailBody.vm")
            // and store the file
            .to("file://target/subfolder")
            // return OK as response
            .transform(constant(OK));
```

Now the route is nice and simple.

Unit testing

Now lets turn our attention to unit testing it. From part 4 we have an unit test that is capable of exposing a webservice and send a test request and assert a mail is received. We will refactor this unit test to start up Camel, as it's Camel that should expose the webservice.

As Camel is very flexible we can create a camel context, add the routes and start it in 3 lines of code so we do it:

```
    protected void startCamel() throws Exception {
     camel = new DefaultCamelContext();
        camel.addRoutes(new ReportIncidentRoutes());
        camel.start();
```

And the rest of the unit test is quite self documenting so we print it here in full:

```
/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest extends TestCase {

    private CamelContext camel;

    // should be the same address as we have in our route
    private static String ADDRESS =
"http://localhost:8080/part-five/webservices/incident";

    protected void startCamel() throws Exception {
     camel = new DefaultCamelContext();
        camel.addRoutes(new ReportIncidentRoutes());
        camel.start();
    }

    protected static ReportIncidentEndpoint createCXFClient() {
        // we use CXF to create a client for us as its easier than JAXWS and works
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
```

```
        factory.setServiceClass(ReportIncidentEndpoint.class);
        factory.setAddress(ADDRESS);
        return (ReportIncidentEndpoint) factory.create();
    }

    public void testRendportIncident() throws Exception {
        // start camel
        startCamel();

        // assert mailbox is empty before starting
        Mailbox inbox = Mailbox.get("incident@mycompany.com");
        assertEquals("Should not have mails", 0, inbox.size());

        // create input parameter
        InputReportIncident input = new InputReportIncident();
        input.setIncidentId("123");
        input.setIncidentDate("2008-08-18");
        input.setGivenName("Claus");
        input.setFamilyName("Ibsen");
        input.setSummary("Bla");
        input.setDetails("Bla bla");
        input.setEmail("davsclaus@apache.org");
        input.setPhone("0045 2962 7576");

        // create the webservice client and send the request
        ReportIncidentEndpoint client = createCXFClient();
        OutputReportIncident out = client.reportIncident(input);

        // assert we got a OK back
        assertEquals("0", out.getCode());

        // let some time pass to allow Camel to pickup the file and send it as an
email
        Thread.sleep(3000);

        // assert mail box
        assertEquals("Should have got 1 mail", 1, inbox.size());

        // stop camel
        camel.stop();
```

```
    }
}
```

Conclusion

We have now seen how we have created a much nicer solution leveraging Camel's powerful routing capabilities.

What we have here is routing logic with the help of the code comments could be understood by non developers. This is very powerful. In a later part we will look at some of the tools that Camel provides, for instance a tool to generate a nice diagrams of your routes 😀

```
    public void configure() throws Exception {
        // webservice response for OK
        OutputReportIncident OK = new OutputReportIncident();
        OK.setCode("0");

        // endpoint to our CXF webservice
        String cxfEndpoint =
"cxf://http://localhost:8080/part-five/webservices/incident"
                +
"?serviceClass=org.apache.camel.example.reportincident.ReportIncidentEndpoint"
                + "&wsdlURL=report_incident.wsdl";

        // first part from the webservice -> file backup
        from(cxfEndpoint)
            // we need to convert the CXF payload to InputReportIncident that
FilenameGenerator and velocity expects
            .convertBodyTo(InputReportIncident.class)
            // then set the file name using the FilenameGenerator bean
            .setHeader(FileComponent.HEADER_FILE_NAME,
BeanLanguage.bean(FilenameGenerator.class, "generateFilename"))
            // and create the mail body using velocity templating
            .to("velocity:MailBody.vm")
            // and store the file
            .to("file://target/subfolder")
            // return OK as response
            .transform(constant(OK));

        // second part from the file backup -> send email
        from("file://target/subfolder")
            // set the subject of the email
            .setHeader("subject", constant("new incident reported"))
            // send the email
            .to("smtp://someone@localhost?password=secret&to=incident@mycompany.com");
    }
```

In the next part's look at using XML to create the route instead of Java code. Then it might be even more readable by non developers.

Links

- Introduction
- Part 1
- Part 2
- Part 3

- Part 4
- Part 5
- Part 6

Tutorial-Example-ReportIncident-Part6

Part 6

... Continued from Part 5

We continue from part 5 where we ended up with a complete solution using routes defined in Java code. In this last part of this tutorial we will define the route in XML instead.

Sidebar
As I am writing this, its been 4.5 years since I wrote the first 5 parts of this tutorial. Recently an user on stackoverflow praised this tutorial said it helped him get onboard Camel. Though he was looking for part 6 with the routes in XML. Frankly I have forgot all about adding this part. So lets close the book and get this part into the Camel docs.

The XML code below is included in the example in `camel-example-report-incident` in the directory `src/main/resources/META-INF/spring/`. The file is named `camel-context.xml`

camel-config.xml

After 5 years we are at the end

So finally we managed to add part 6 to this tutorial. Yes we have used a lot of time to write this tutorial, so we hope that it helps you ride the Camel.

Links

- Introduction
- Part 1
- Part 2
- Part 3
- Part 4
- Part 5
- Part 6