Integrating web services and MTOM with existing applications using Fuse ESB

We will walk through a complete application, showing how you can integrate a MTOM-enabled web service and a Spring/JPA application using FuseSource components.

Jos Dirksen
FuseSource Community Member

Car accidents happen every day, which isn't so strange if you imagine that there are some 800 million cars in the world. Nowadays almost every car is insured and the financial consequences of an accident are often handled completely by your insurance company. In Europe we have something called the “European Accident Statement Form”, which both sides fill in when they've had a car accident (see the references for an example of such a form). Each party agrees to what's on the form, and it gets sent to both parties insurance companies, who handle the rest.

Processing this form is done manually and thus is very error prone. What we're seeing is that besides the paper form, some insurance companies also allow you to fill in the details online, which then can be processed digitally. In this article I'm going to show you how you can implement the integration part of this application for an insurance company.

Case of the insurance company

The insurance company we're working for wants to allow third parties to digitally send damage reports. It already has two back-end systems where the data is stored and they want to keep using those. The first system they have is a document management system to store a photo of the crash scene, and the second system they have is an application that stores damage reports in a database so they can be processed later.

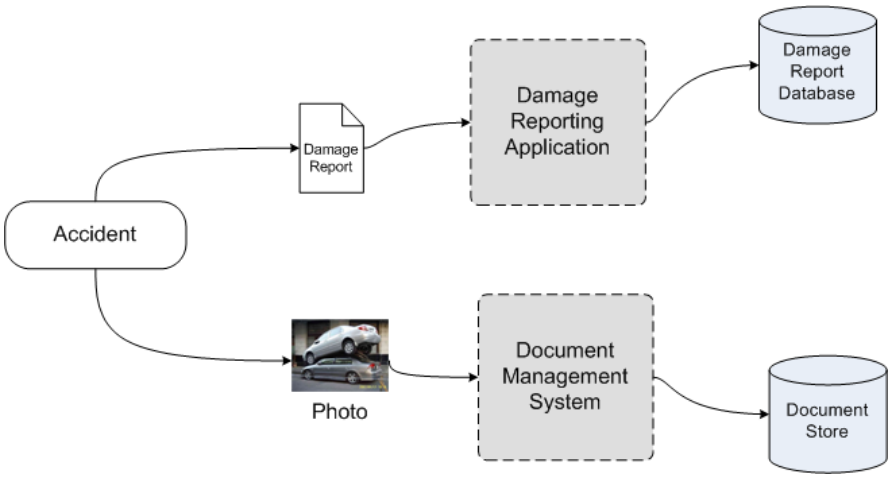

Figure 1. Current applications

This insurance company wants to offer a new web service where a photo of the accident and the complete damage report can be sent in one request. This new web service should also return an identifier that can be used later on to retrieve this damage report again.

In this article we'll take the following steps:

1. Look at the current situation, what back end systems are we going to integrate
2. Set up the development environment using Eclipse and Maven 2
3. Look at how the current applications and the domain model are set up
4. Create the new customer facing web service using MTOM for the photo
5. Integrate with the document management system
6. Integrate with the damage reporting application
7. Test the complete integration flow

Let's start with setting up the development environment. For this we're going to use Maven 2 and Eclipse.

The development environment

When you want to develop with Fuse you only need two tools. Eclipse for the Java and XML coding, and Maven 2 for creating Fuse/JBI specific artifacts. I've used Eclipse ganymede-sr1 that you can get from http://www.eclipse.org/downloads/. After you've installed Eclipse also get and install Maven 2 from http://maven.apache.org/. The version of Maven I've used is 2.0.9. Also make sure you've got Java 6 installed, since all the examples are tested with that version. Finally you of course need a FuseSource environment to work with. Go to the http://fusesource.com/downloads/ website and download the Fuse ESB

3.4 version from there, and run the installer, which allows you to specify where to install Fuse. When you're working through this example, or look at the documentation on the Fuse site, you'll often see references to ServiceMix. The reason is that the Fuse ESB is based on Apache ServiceMix, so you'll see references to ServiceMix in various places.

> ***Optionally install m2eclipse in Eclipse***
> You could also install the m2eclipse plugin. This plugin allows you to easily create new Maven 2 projects using a wizard where you can base your project on a selected archetype. Besides that it also simplifies editing the Maven 2 POM file, adding dependencies and running specific Maven goals. Explaining how the m2eclipe plugin works is outside the scope of this article, but if you want to have excellent integration between Maven 2 and Eclipse go to the m2eclipse website (http://m2eclipse.codehaus.org/) and follow the instructions there to install this eclipse plugin.

The next thing we need to do is download all the projects that are used for this example. We've provided a zip file which you can download from http://fusesource.com/resources/collateral/. After you've download this archive you can start up Eclipse. You will be greeted with a welcome screen that you can click past to see an empty workspace. The next thing to do is import the projects. Use "`File->Import...`" from the menu and select from the "`General`" folder the "`Existing projects into workspace`" option and click "`Next`". On the next screen click the "`Select archive file`" option and use "`Browse...`" to select the archive file you've just downloaded. Make sure all the projects shown are checked and click "`Finish`". If all went well you'll see a number of projects in the project explorer and we're ready to start developing. Let's start by looking at the existing applications.

Details of the existing applications

I mentioned that there were already two applications at this insurance company with which we have to integrate. Let's first look at the document management system (DMS). This DMS isn't really a DMS, what this company does, it just stores the received files in a directory and stores the name of the file in the database together the rest of the damage report. So to integrate with this system, we only need to configure Fuse in such a way that the photo part of the damage report is sent to the specific directory. Since we want to be able to uniquely identify the photo later, we also need to make sure the photo gets a unique filename.
The damage reporting application is a simple spring application which uses JPA/Hibernate to persist its data. Figure 2 shows the components of this application.

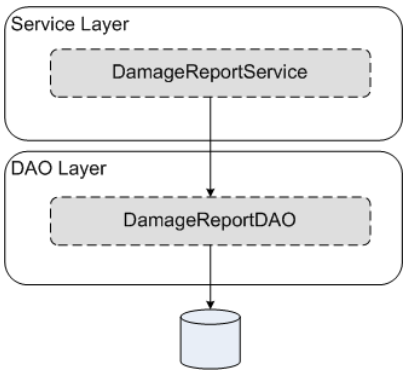

Figure 2. Diagram of the spring damage reporting application

As you can see in this figure this application has a DAO layer which handles the persistency and has a service layer that provides functionality to other applications. The `DamageReportService` exposes functionality to work with `DamageReports`. A `DamageReport` contains all the information the two parties in a car accident should submit. Figure 3 shows this `DamageReport` class in more detail.

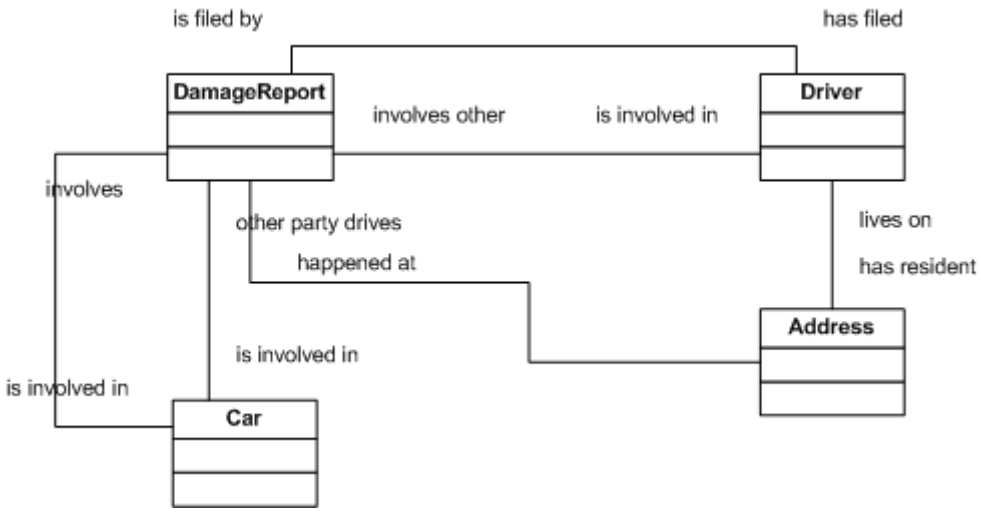

Figure 3. Diagram showing the domain model

I won't get into too much detail on this application. The complete source code for this application is included in the zip file, and if you look at you eclipse workspace you should see the following two projects:

- **insurance-db**: This project contains the domain model and the DAO layer of the damage reporting application. It uses JPA annotations to define the mapping to the database and defaults to using an in memory HSQL database and everything is tied together using Spring.
- **insurance-service**: This service, which is also Spring-based, makes use of the **insurance-db** project and provides a service layer on top of the DAOs provided by the **insurance-db** project. This service contains the `DamageReportServiceImpl` with which we'll integrate.

These two projects, as are all the other projects, are Maven 2 projects. Before we can use Eclipse to explore these projects we first have to create the Eclipse specific property files. You can do this by going to the directory where these projects are installed and run `mvn eclipse:eclipse` from there. So go to the `<workspace>/insurance-db` directory and run `mvn eclipse:eclipse`. Before we do the same for the service project we first need to install the `insurance-db` component. From the same directory you ran the previous maven command, now run `mvn install`. This will install this component in your local Maven 2 repository.

> ***Working with Maven2 and dependencies***
> A very strong point of Maven2 is that it forces you to think about dependencies. We mentioned that the `insurance-service` project depends on the `insurance-db` project. To resolve this dependency Maven 2 checks a number of online repositories to see if it can find the artifact that fulfills this dependency. For most of the dependencies Maven 2 will find a match online, which it will download and store in your local repository. However the dependencies on local projects Maven 2 won't be able to find online. To avoid missing dependencies you must run `mvn install` on these projects to install them to your local repository. That's also the reason why you have to first install the `insurance-db` project before you can run or install the `insurance-service` project.

Now go to the `<workspace>/insurance-service` directory and execute the same command. The output, from the last command, should look something like this:

```
$ mvn eclipse:eclipse
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'eclipse'.
[INFO] ------------------------------------------------------------------------
[INFO] Building insurance-service
[INFO]    task-segment: [eclipse:eclipse]
[INFO] ------------------------------------------------------------------------
[INFO] Preparing eclipse:eclipse
[INFO] No goals needed for project - skipping
[INFO] [eclipse:eclipse]
[INFO] Using source status cache: /workspace-insurance/insurance-service/target/mvn-
eclipse-cache.properties
[INFO] Wrote settings to /workspace-insurance/insurance-
service/.settings/org.eclipse.jdt.core.prefs
[INFO] Wrote Eclipse project for "insurance-service" to /workspace-insurance/insurance-
```

```
service.
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Thu Dec 10 15:36:34 CET 2008
[INFO] Final Memory: 6M/82M
[INFO] ------------------------------------------------------------------------
```

Code listing 1: mvn eclipse:eclipse output

If you now go to your Eclipse environment and refresh these two projects, Eclipse now sees them as normal Java projects. We won't go into detail here about these two projects; we'll only look at the spring configuration of the service project, since we'll be using that one later in our integration flow.

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

    <import resource="classpath:insurance-db-beans.xml"/>

    <bean id="damageReportService"
       class="com.iona.examples.insurance.service.impl.DamageReportServiceImpl">
        <property name="damageReportDao" ref="damageReportDao"/>
    </bean>
</beans>
```

Code listing 2: Damage report service application context

As you can see in listing 2, we've defined a `damageReportService` bean, which points to our implementation. This is the bean that we'll use when we integrate the Fuse integration flow with this application. Before we look at the Fuse part also run `mvn install` on the `insurance-service` project.

Fuse-based solution

Now that we know how the back office applications look, it's time to start with the Fuse part. In figure 4 shows a complete overview of the flow we're going to create.

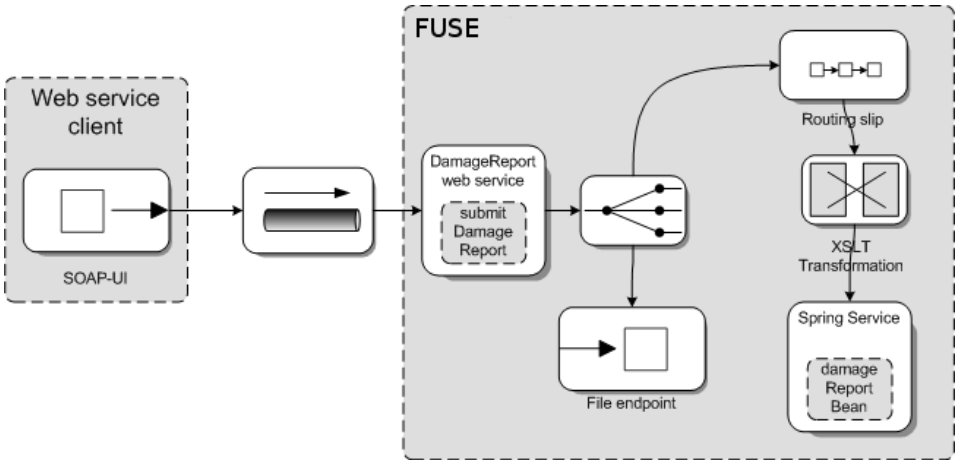

Figure 4. Overview of the FuseSource solution

In this diagram you can see the complete flow of a message. A message in this scenario takes the following steps:

1. A web service client, in this case SOAP-UI, sends a SOAP message. This SOAP message contains the complete damage report. The client will also send a photo of the accident as attachment to this SOAP message. We'll use MTOM to optimize the transmission of this photo.
2. The web service, which is hosted in Fuse will receive the message and the attachment. This web service doesn't contain any business logic, it just receives the message and sends it to the next component.
3. The next component is a custom Camel-based router. This router adds some header properties, as we'll see later in this article, and sends the message to two different components.
4. The first of these components is another Camel-based router. This router uses the “routing slip” pattern to first calls the XSLT transformation component, and use the result from this transformation

as the input for the next component which is the Spring-based service we discussed earlier.

5. The first Camel-based router also sends the message to the File component. This component takes the original attachment, the photo, and writes that to the file system using a unique name.

As you can see, we have to take some steps before we can deliver the message to the DMS and to the Spring service. The first thing we'll do is look at how to create the web service with which we can receive the damage report and the attachment.

Create the customer facing web service

Exposing a web service in Fuse is very straightforward. You just have to create a WSDL file and use the `servicemix-cxf-bc` binding component to create a web service from it. Creating a WSDL file from scratch, however, is boring and very time consuming work. What you often see is that instead of starting with nothing, you take an existing Java interface and let tools such as CXF generate the WSDL for you. This is also the approach we will take.
If you look at the `<workspace>/insurance-webservice` project in your Eclipse workspace, you'll notice that the only source file in there is the `InsuranceService` interface.

```
@WebService(name="InsuranceServicePortType",
targetNamespace="http://iona.com/insurance")
public interface InsuranceService {

       /**
        * Webservice interface we're exposing. For this interface we want to receive
        * a damagereport and a photo.
        *
        * @param report the report to receive
        * @param photo the photo to receive
        * @return a long determining the id that is associated with this report
        */
       public long submitDamageReport(@WebParam(name = "report")DamageReport report,
                                      @WebParam(name="photo") Document photo);

}
```

Code listing 3: InsuranceService interface that is used to generate the WSDL from.

This Java interface contains a number of JAX-WS annotations that explain how this interface should be described in a WSDL file. If you've looked at the interface of the `DamageReportService` in the `<workspace>/insurance-service` project you might notice that in the latter we didn't specify a photo attribute. The reason is that for the `DamageReportService` from the `<workspace>/insurance-service` project, we're only interested in the `DamageReport` not in the photo. The photo is stored in the DMS. The exposed web service however should be able to receive both elements at the same time.

To generate the WSDL we've configured Maven 2 to use CXF to generate a WSDL file. To generate this file yourself just go to the `<workspace>/insurance-webservice` and run `mvn install`. In the `target/generated/wsdl` directory you can now find your generated WSDL file. We're almost done, we just need to make some small alterations to the generated WSDL before we can expose it using Fuse:

- **Add the xmime namespace**: we're going to work with MTOM. To indicate in a WSDL that a certain element can be used as an MTOM attachment we need to give it a specific type from the xmime namespace. So in the header of the WSDL file we added the following namespace definition:

```
xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
```

- **Correctly define the photo parameter**: In the generated WSDL file, the document parameter wasn't defined correctly. Since we have to change this to something MTOM can work with I created a new complex type and made sure the photo element was of that type:

```
<xs:complexType name="document">
  <xs:sequence>
    <xs:element minOccurs="0" name="name" type="xs:string" />
    <xs:element minOccurs="0" name="data" type="xmime:base64Binary"
                xmime:expectedContentTypes="application/octet-stream" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="submitDamageReport">
  <xs:sequence>
    <xs:element minOccurs="0" name="report" type="tns:damageReport" />
    <xs:element minOccurs="0" name="photo" type="tns:document" />
```

```
  </xs:sequence>
</xs:complexType>
```

- **Setting the correct soap address**: the last part we have to change defines on which URL our web service will listen. By default the URL generated is `http://localhost:9090/hello`, we need to change this to an URL which more expresses the service we've exposed. So this was changed to `http://localhost:9090/InsuranceService`.

If you look at the `InsuranceService.wsdl` file in the `<workspace</insurance-fuse-cxf-bc-su` project you can see the complete WSDL with the changes. Now that we've got the WSDL describing the web service we want to expose, the next step is to configure the `servicemix-cxf-bc` to do this for us.

Fuse provides maven archetypes for us that we can use to kickstart a project. With these archetypes you get a basic Maven and Fuse configuration that you can extend later on with your own functionality. The Fuse related projects you see in your workspace have been created in this manner. Fuse itself provides a shortcut to create these archetypes for you. Go to your Fuse installation and for the `servicemix-cxf-bc` component you can use the following command, which is located in the `bin` directory, to create a new project:

```
smx-arch su cxf-bc
                  -DgroupId=com.iona.examples.insurance
                  -DartifactId=insurance-fuse-cxf-bc-su
```

For the examples in this article you don't have to create any new projects. All of them have already been created in the same manner as shown here.

In `src/main/resources` directory of the `<workspace>/insurance-fuse-cxf-bc-su` project you will find two files. The WSDL file we've discussed earlier and an `xbean.xml` file. This `xbean.xml` file is used to configure Fuse. Each component has there own specific `xbean.xml` configuration. The following listing shows the configured `xbean.xml` with which we expose the WSDL as a web service.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:dr="http://iona.com/insurance"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/cxfbc/1.0
         http://servicemix.apache.org/schema/servicemix-cxf-bc-2008.01.xsd
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <cxfbc:consumer wsdl="classpath:InsuranceService.wsdl"
                      endpoint="dr:InsuranceServicePortTypePort"
                      service="dr:InsuranceServiceService"
                      targetService="dr:routing"
                      targetEndpoint="splitterEndpoint"
                      mtomEnabled="true" />

</beans>
```

Code listing 4: Configuration of the servicemix-cxf-bc to expose a MTOM enabled web service.

In this listing we see a single XML element with a number of attributes. This is the entire configuration that is needed to create our web service. The following table explains the attributes.

Name	*Description*
wsdl	The location of the WSDL file that is to be exposed. In this case it points to the WSDL file we created.
endpoint	In a WSDL file you can have multiple ports. Here we define which port from our WSDL we want to use.
service	And we can also have multiple services in a WSDL file. So we also define the service from our WSDL we'd like to use.
targetService	This attribute defines the service we want to call when a message is received on this web service. In our case the service points to first Camel router where we'll split the message.
targetEndpoint	A service can have multiple endpoints, so we also define the target endpoint of the service we want to call.

mtomEnabled	As the name of this property implies. If we set this to true, the web service will support MTOM, and be able to process SOAP calls that have MTOM attachments.

That's it for the configuration of the first part of the integration flow. When we deploy this, as we'll show at the end of this article, we'll get a web service running on `http://localhost:9090/insuranceService` that will forward messages it receives to the Camel routing component we'll discuss in the next section.

Use Fuse Mediation Router to multicast the message

The next part of our message flow is handled by the `servicemix-camel` component. Camel is a stand-alone routing engine that can also be used without Fuse, and provides great support for many of the "Enterprise Integration Patterns". If you want to learn more on Camel, look at the Camel site at http://activemq.apache.org/camel. If you want to learn more about the productized and supported version look at the Fuse Mediation Router at http://fusesource.com/products/enterprise_camel.

For our scenario we want this router to do a couple of things:

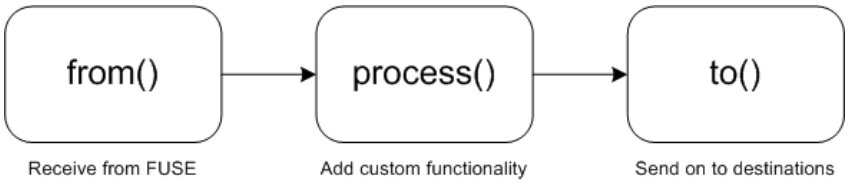

Figure 5. Multicasting router steps in Camel

- **Receive messages**: The web service we've configured in the previous section sends it's messages to the `dr:routing` service on the `splitterEndpoint`. So we need to configure our Camel route to receive messages on this service and endpoint.
- **Create a unique filename for the photo**: This is the last place we've got a single message, since we'll send a message to the DMS and to the damage reporting service. This is thus the place where we need to generate the unique id for the photo and make sure both messages have a reference to that id.
- **Multicast the message**: The final step this router should do is multicast this message to the DMS and to the damage reporting service.

You can configure Camel two different ways. Camel provides a Spring/XML-based configuration and Camel provides a Java-based DSL you can use to configure the routes. In this example we'll be using the Java-based DSL for the configuration. Let's first look at this configuration, after which we'll look at how to run this in Fuse.

```
public class InsuranceRecipientList extends RouteBuilder {

  private final String RECIPIENT_DOCUMENT =
   "jbi:endpoint:http://iona.com/insurance/damage/ServiceRoutingSlip";
  private final String RECIPIENT_SERVICE =
   "jbi:endpoint:http://iona.com/insurance/fileSenderService/dmsEndpoint?mep=in-only";

  private final String PHOTO_ID = "photoID";

  public void configure() {
    // we read from the incoming endpoint and set the headers for the
    // routing slip
    from("jbi:endpoint:http://iona.com/insurance/routing/splitterEndpoint").
    // we use the processor to generate a unique id for the attachment
    // change the name of the file to store, and set the id on the
    // original XML message.
    process(new Processor() {
      public void process(Exchange exchange) throws Exception {
        Message inMessage = exchange.getIn();
        Set<String> attachmentNames = inMessage.getAttachmentNames();
        inMessage.setHeader(PHOTO_ID, generateID((String)
attachmentNames.toArray()[0]));
      }
    }).
    // now we multicast the message to the document service
```

```
    // and the insurance service.
    to(RECIPIENT_SERVICE,RECIPIENT_DOCUMENT);
  }

  /**
   * This method returns an unique id for the photo
   *
   * @param photoName
   *            the name of the photo
   * @return a unique id, based on the name of the photo
   */
  private String generateID(String photoName) {
    return System.currentTimeMillis() + photoName;
  }
}
```

Code listing 5: Camel based Java DSL route

In this listing you can see by reading through the comments the steps a message will go through when it's received by this route:

1. **Receive the message**: This is done by the first statement in the source code. If you look at listing 5, you can see the `from()` method. With this method you define the service name and the endpoint this route is listening on. If you look back at the part where we configured the web service you can see that the value configured here matches the target endpoint and service we've specified for the web service, just in another form. The first part (`http://iona.com/insurance`) is the name space, the second part (`routing`) is the service name, and the final part (`splitterEndpoint`) is the name of the endpoint. So any message that is received by the web service is forwarded to this Camel route.
2. **Create a custom filename**: The next part of this route is used to create a unique file name for the photo. This is done by the `process()` method, which takes a `Processor` as its argument. A `Processor` allows you to quickly add some custom functionality to the message route. In this case, we get the attachment from the message. The name from this attachment, which is the filename of photo we sent using MTOM, is prefixed with a timestamp and added to the message as a property, so that we can use this later on in the other FuseSource components.
3. **Send the message on**: The last thing we do is send the message to the next two components. This is done by using to `to()` method. You can use the same format, as we saw in step one, to add destinations. The message will be multicasted to all these specified destinations. One thing to note here is the `mep=in-only` we added to the DMS endpoint. This is needed since the component that's listening on that destination can only receive one-way message, it won't return a result.

And that's it for the configuration part at least. We now need to configure the Fuse artifact that we can use to add this route to Fuse. This is done in pretty much the same way as we did for the CXF-based web service. I've created a new Maven project using the Fuse provided archetypes, which you can see in Eclipse if you look for the `<workspace>/insurance-fuse-camel-su` project. In this project you'll find the Java class I just described and a `camel-context.xml` file which is used to configure the FuseSource Camel component. Let's look at this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/spring
http://activemq.apache.org/camel/schema/spring/camel-spring.xsd
    ">

  <camelContext id="camelContext" useJmx="true"
              xmlns="http://activemq.apache.org/camel/schema/spring">
    <package>com.iona.examples.insurance.routes</package>
  </camelContext>

</beans>
```

Code listing 5: Configure Fuse to read the Camel Routes

This configuration will be parsed when it's deployed to Fuse and the package specified in the `<package>` tag is searched for Camel routes. Our route is in that package so it will be picked up and started.

On to the next part. We've now sent the message to two different destinations. One of them will write the message to the database using the Spring service, and the other one will just store the photo in the file

system. We'll start by looking at the latter one.

Write the attachment using a file endpoint

FuseSource also provides components for reading and writing to the file system. We'll use that component to write our attachment to the file system using the name we generated in the previous section. If you look at the `<workspace>/insurance-fuse-file-sender-su` project in your workspace (again based on one of Fuse archetypes), you can see an `xbean.xml` file containing this component's configuration.

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

<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:dr="http://iona.com/insurance"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/file/1.0
       http://servicemix.apache.org/schema/servicemix-file-2008.01.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

 <file:sender service="dr:fileSenderService"
              endpoint="dmsEndpoint"
              directory="fuse-work/test-out">
    <property name="marshaler">
      <bean class="com.iona.examples.insurance.file.InsurancePhotoFileMarshaller" />
    </property>
  </file:sender>

</beans>
```

Code listing 5: Configure Fuse to read the Camel Routes

Most of the items in the configuration will start to look familiar by now. The following table quickly summarizes the elements and attributes you see in this configuration.

Name	***Description***
file:sender	A sender can be used to write a message to a directory. We also have a reader that, as the name implies, reads messages from a directory and sends them on to some other destination.
endpoint	The endpoint, together with the service, define the destination this sender is listening on
service	The service, together with the endpoint, define the destination this sender is listening on
directory	The directory, relative to the home directory of Fuse ESB, where the files will be stored
property name=marshaler	A marshaler is used to write the message in a non-standard way
bean class=””	The implementation of the marshaler we'll use

Normally when a `file:sender` receives a message on it will write the content to the specified directory. In our case this would mean we would write out the received XML. This isn't something we want. We want to write the attachment using our generated name. To accomplish this we have to create a custom marshaler. The implementation of this marshaler is shown next.

```
public class InsurancePhotoFileMarshaller extends BinaryFileMarshaler {

    public String getOutputName(MessageExchange exchange, NormalizedMessage message)
                                                              throws MessagingException {
        return (String) message.getProperty("photoID");
    }

       @SuppressWarnings("unchecked")
       public void writeMessage(MessageExchange exchange,
                                NormalizedMessage message,
                                OutputStream out, String path)
                                         throws IOException, JBIException {

         Set<String> names = message.getAttachmentNames();
```

```
        DataHandler handler = message.getAttachment((String) names.toArray()[0]);
        InputStream is = handler.getInputStream();
        FileUtil.copyInputStream(is, out);
      }
}
```
Code listing 6: A custom marshaler that writes out the attachment using the generated name

This marshaler is based on the already available `BinaryFileMarshaler` and overrides a couple of methods to implement our custom behavior. The first method that is shown in the `getOutputName()` method. This method should return the name that is used as filename of the output file. In our case we just retrieve the property we set using the Camel route and return that. The other method that is overridden is the `writeMessage()` method. Here we just pass in the `InputStream` to our photo and use the `FileUtil` to write out the photo.
With this marshaler attached to the `file:sender` the photo is correctly written to the filesystem.

Now that we've the message flow to the DMS finished let's look at the message flow to the Spring service. For that we need to take three more steps.

Exposing the Spring based application in Fuse

Before we look at how we're going to get the message in the correct format to the Spring service, lets first look at how we can expose the Spring application in Fuse. For this we make use of one of the standard components Fuse provides: `servicemix-cxf-se`. With this component you can expose (annotated) POJOs to other services in Fuse. To keep things separated we'll create a simple class that services as a web service facade. This class is annotated with JAX-WS annotations and will delegate all the requests to our spring implementation.

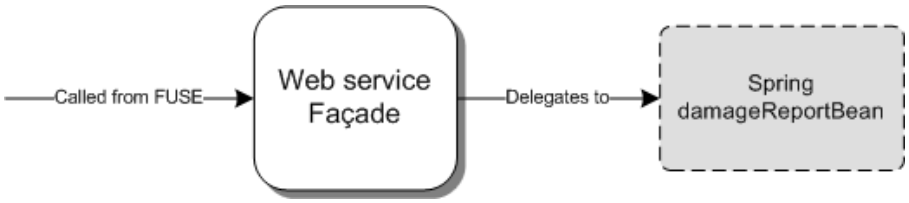

Figure 6. Web service facade for the Spring implementation

Let's first look at how to create the web service facade. We can create this class and the associated JAXB classes from scratch or once again, generate them. We'll take this last option, especially since we already got a WSDL file that almost matches the interface we want to provide. The only thing that changes is that the previous WSDL had an operation that took two parameters - the new one will only take one parameter. We'll also change the namespace of this new WSDL to http://iona.com/insurance/internal to avoid naming conflicts between the generated JAXB classes and the domain model we've specified ourselves. I won't go into the details of the WSDL file here. If you're interested how this WSDL looks, you can find it in the `<workspace>/insurance-fuse-cxf-se-su` project. In this project you can also find the `pom.xml` file that describes how, based on this new WSDL, the JAXB classes and a server stub are generated. If you make changes to this WSDL, you can simply run `mvn install` once again to regenerate all the classes.

Now that we've got the generated classes, we'll use the JAXB classes together with JAX-WS annotations to create the web service facade, which is shown in the following code listing.

```
@WebService(targetNamespace = "http://iona.com/insurance",
       serviceName = "DamageReportService")
public class DamageReportWebserviceImpl {

    private DamageReportService damageReportService;

    @WebResult(name = "return", targetNamespace = "")
    @RequestWrapper(localName = "submitDamageReport",
             targetNamespace = "http://iona.com/insurance",
             className = "com.iona.insurance.internal.SubmitDamageReport")
    @ResponseWrapper(localName = "submitDamageReportResponse",
             targetNamespace = "http://iona.com/insurance",
             className = "com.iona.insurance.internal.SubmitDamageReportResponse")

      @WebMethod
       public long submitDamageReport(@WebParam(name = "report",
```

```
                        targetNamespace = "")DamageReport report) {
       // implementation of convertReport not shown
       return damageReportService.submitDamageReport(convertReport(report));
       }

      @WebMethod(exclude=true)
       public void setDamageReportService(DamageReportService damageReportService) {
              this.damageReportService = damageReportService;
       }
}
```

Code listing 7: The web service facade which delegates to the real service

In this class you can see a number of JAX-WS annotations that describe how this class should be exposed by Fuse. This pretty much matches the definitions from the WSDL file we defined at the beginning of this article. The major change is that the `submitDamageReport` method now only takes one argument instead of two.
The implementation of this method converts the received objects to our own domain model and invokes the real implementation. You can also see that we've specified a getter and a setter for the `DamageReportService`. This is done because we'll use Spring to inject the real implementation into this class.

Let's look at the `xbean.xml` configuration to see how we can inject the real implementation into this class and expose this web service facade to Fuse.

```
<beans xmlns:cxfse="http://servicemix.apache.org/cxfse/1.0"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/cxfse/1.0
       http://servicemix.apache.org/schema/servicemix-cxf-se-2008.01.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <import resource="classpath:insurance-service-beans.xml"/>

    <cxfse:endpoint>
       <cxfse:pojo>
        <bean class="com.iona.examples.insurance.webservice.DamageReportWebserviceImpl"
>
            <property name="damageReportService" ref="damageReportService"/>
          </bean>
       </cxfse:pojo>
    </cxfse:endpoint>
</beans>
```

Code listing 8: xbean configuration which exposes the web service facade and inject the real service

In this configuration you can see that we first import the real services using the `import` statement. This imports the `insurance-service-beans.xml` file from the `<workspace>/insurance-service` project (we've used maven to add a dependency to the service project). Next we define a `cxfse:endpoint`. We've used the `cxfse:pojo` element to point to the web service facade we've just shown. You can also see that by using standard Spring dependency injection, we inject the real service (`damageReportService`) into the web service facade.
With this configuration the web service facade is exposed in Fuse, and calls made to this service will be, after conversion, delegated to the real implementation.

However, we can't just forward the received SOAP message to this service. The received SOAP message contains two parameters, the report and the photo, and this service can handle only one parameter. We have to transform the received message so that this service can work with it. Luckily also for this Fuse provides a standard component.

Transform the message to the correct format

For transformation Fuse provides an XSLT services based on Saxon. In your workspace you'll find a project named `<workspace>/insurance-service-xslt-su`. This project was also created using one of Fuse archetypes. The only thing that was altered was the default XSLT that was created. We just mentioned that we needed to transform the incoming SOAP message so that our web service facade can work with it. This transformation needs to do the following things:

- **Remove the photo argument**: our web service facade only takes a single argument, so we have to make sure that this element is removed from the request message.
- **Add the generated ID to the message**: Remember that we generated a unique ID for the photo

and that we stored our photo using that ID? We also need to make sure that this ID is also known by the damage report service. If not we can never correlate which photo belonged to which damage report.

Both of these steps are very easy to accomplish using this component. In the following listing the XSLT is shown that does this.

```
<xsl:stylesheet xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
  version='1.0'>

  <!-- The output should be XML -->
  <xsl:output method="xml" indent="yes" encoding="ISO-8859-1" />
  <!-- This is used to get the header from fuse  -->
  <xsl:param name="photoID" />

  <!--
    This template replaces the current empty photoID, with the one which
    was generated
  -->
  <xsl:template match="//photoID">
    <photoID>
      <xsl:value-of select="$photoID" />
    </photoID>
  </xsl:template>

  <!--
    Our insurance service doesn't do anything with binary data, so
    we just remove that element
  -->
  <xsl:template match="//photo" />

  <xsl:template match="article">
    <html>
      <xsl:apply-templates />
    </html>
  </xsl:template>

  <!--
    The following two xsl:templates make sure everything else is matched
    and copied to the new XML document
  -->
  <xsl:template match="/">
      <xsl:copy>
        <xsl:copy-of select="attribute::*" />
        <xsl:apply-templates />
      </xsl:copy>
  </xsl:template>

  <!--
    Match and copy all the nodes and attributes
   -->
  <xsl:template match="node() | @*">
    <xsl:copy>
      <xsl:apply-templates select="node() | @*" />
    </xsl:copy>
  </xsl:template>

</xsl:stylesheet>
```

Code listing 9: XSLT transformation to prepare the message for the web service facade

I won't go into too much detail for this XSLT since nothing really complicating is happening in this transformation. What we do is, we remove the photo argument from the input message using the `<xsl:template match="//photo" />` template, and we replace the current `photoID` element, which is located in the `DamageReport` element, with the specified parameter. You might wonder how we get the `photoID` parameter, since we don't specify its value. This resolving is handled by Fuse. If we specify `<xsl:param name="photoID" />` Fuse will check if he can find such a parameter in the message properties. If this is the case it will use the found value as the value for the property. In our case this means that the generated ID we set using the Camel route is used as value for this XSLT parameter.

With this transformation the SOAP message will have the correct format and can be processed by the exposed web service facade we defined in the previous section. The only thing we need to do is configure the XSLT to be used by Fuse. This is done by the following xbean.xml file.

```
<beans xmlns:saxon="http://servicemix.apache.org/saxon/1.0"
       xmlns="http://www.springframework.org/schema/beans"
```

```
       xmlns:ins="http://iona.com/insurance"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/saxon/1.0
http://servicemix.apache.org/schema/servicemix-saxon-2008.01.xsd
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <saxon:xslt service="ins:transformationService"
              endpoint="documentTransformation"
              resource="classpath:transform.xsl" />

</beans>
```
Code listing 10: Configuration of the XSLT transformation for Fuse

With this configuration the transformation is exposed in Fuse and we can call this transformation by sending a message to the specified service and endpoint name.

Well, we're almost done. The only missing link is how we get the message from the Camel route we defined earlier to the web service facade, but first make sure it's transformed by the XSLT component. For this we define another Camel route. This route will implement the routing slip pattern.

Using Camel to create a routing slip

With the routing slip pattern, we can define a number of services that get called one after the other and where the output from the first service is used as the input for the next and so forth. So we'll configure this Camel route in such a way, that it'll first call the transformation and next it will call our web service facade.

Figure 7. Routing slip implementation in Camel

For this we'll use the Java-based DSL again, which is shown next.

```
public class DamageServiceRoutingSlip extends RouteBuilder {

       private final String ROUTING_SLIP_HEADER = "routingSlipHeader";
       private final String ROUTING_SLIP_VALUE =
   "jbi:endpoint:http://iona.com/insurance/transformationService/documentTransformation"
 + "#jbi:service:http://iona.com/insurance/DamageReportService";

    public void configure() {
      from("jbi:endpoint:http://iona.com/insurance/damage/ServiceRoutingSlip").
      process(new Processor() {
             public void process(Exchange exchange) throws Exception {
                    Message inMessage = exchange.getIn();
                    exchange.setProperty(ROUTING_SLIP_HEADER, ROUTING_SLIP_VALUE);
                    inMessage.setHeader(ROUTING_SLIP_HEADER, ROUTING_SLIP_VALUE);
             }

      }).
      // now we apply the routing slip to send the message
      routingSlip(ROUTING_SLIP_HEADER, "#");
    }
}
```
Code listing 11: Routing slip configuration for the transformation and the damageService

In this Camel route we again start with the `from()` method to define the destination. This destination is the same one as we specified as one of the destinations for the previous Camel route. Once a message is received the routing slip is configured. This is done by adding specific headers to the message, which we've done using the `process()` method using a custom `Processor`. Finally the `routingSlip()` method is invoked, where we first send the message to the transformation service, and then send the result from the transformation service to the web service facade. This route resides in the same project as the other Camel route, and is already picked up in the configuration we've shown for the other Camel route.

So we're done... We wrote some custom code, created some XML files, added an XSLT transformation and that's it. The only thing left to do is package everything up in a format Fuse can understand and deploy it to Fuse.

Package everything and deploy to Fuse

So far we created a number of different projects. All these projects make use of a specific Fuse component. These projects in Fuse/JBI terms are called service unit. A service unit can be deployed to a specific component and that component will then read the configuration and expose or consume services. In Fuse you don't just deploy service units, you package related service units together to form a service assembly. This service assembly is then deployed to Fuse.

What we'll do here is make a Fuse assembly that contains all the service units we've created in this article. Luckily we don't have to do this by hand. Just as for the service units, Fuse also provides an archetype for a service assembly. If you look at the <workspace>/insurance-fuse-sa project, you'll see the assembly we've configured for this article.

There isn't much to see in this assembly, the only interesting file is the pom.xml file. This file holds references to all the service units you want to deploy to Fuse. For instance the transformation service unit we described earlier is added as a maven dependency to the pom.xml file.

```
    <dependency>
       <groupId>com.iona.examples.insurance</groupId>
       <artifactId>insurance-fuse-saxon-xslt-su</artifactId>
       <version>0.0.1-SNAPSHOT</version>
    </dependency>
```

If you look at this file, you'll also see all the other service-units we created.

The final step is to install this service assembly in Fuse. Before we do this, you have to make sure all the dependencies of this service assembly are installed. So go through all the projects in the workspace and from a command prompt run mvn install. This will install all the service units in your local maven repository. Once you've done that you can run mvn install on the insurance-fuse-sa project. This will create a insurance-fuse-sa-1.0.SNAPSHOT.zip file in the project's target directory containing all the service units. If you copy this file to your Fuse's installation hotdeploy directory, and assuming you have Fuse running, it will install and start all the service units in the corresponding components.

Now you can for instance use soap-ui to test this complete flow. If you don't have soap-ui yet, download it from http://www.soapui.org/ and install it. Startup soap-ui and you'll be presented with an empty screen. From the menu select File->New WSDL project. In the “New WSDL project” screen enter a name for the project and for the initial WSDL type: http://localhost:9090/InsuranceService?wsdl .If you now select “OK”, a project will be created with a sample request, replace this sample request with the one added to this article. This way you have a completely filled in request you can use for the service call.

Note: make sure you use valid data types in the request XML. Use the right-click | Validate action within soap-ui to ensure your request is valid. Otherwise you’ll see Unmarshalling errors in Fuse ESB.

We have one more step to do before we can send this request to our service, and that is adding an attachment. At the bottom of the request editor you can see the text “Attachments(0)”. Click this text and you'll see a view where you can add attachments, click on the + to add a new attachment. Browse to a file and add it as attachment. Click on the part column to assign it to the data part of the message. Now look at the request properties and set “Enable MTOM” to true. With all this in place you can hit the “play” button and send the message. You'll now see all kinds of logging in the Fuse console where you can follow the message's flow through Fuse. You'll also see here that the Spring service is called and if you look at the <Fuse ESB>/fuse-work/test-out directory, you can find the photo you sent.

Conclusions

In this article you've seen how you can use a number of the components FuseSource provides to create message flows. We've seen how you can use the servicemix-cxf-bc to expose a MTOM enabled webservice and use the servicemix-camel component to route messages. We've also shown you how to use the fuse-saxon component for XSLT transformations and how to customize the output of the file:sender. The last thing we showed was how to invoke existing Spring services using the servicemix-cxf-se component.
As you've seen Fuse provides an excellent environment to handle message flows. Fuse ESB provides a set of components you can use to create complex message flows and handle real integration problems.

As a final note I'd just like to give some general tips, which will help in working with Fuse. From experience

most problems or strange behavior you'll encounter can often be addresses to one of the following points:

- **Maven issues**: Remember that the service assembly will take the latest versions from your local repository. If you make a change to one of the service units, you'll have to `mvn install` before the assembly plugin will use this new version. If you see old versions or old libraries make sure you run a `mvn clean` before the install.
- **Destination naming issues**: Personally, these are the errors I make the most. Services communicate with each other using a service's name and the endpoint. The service name is a fully qualified name, so make sure if you call a service you include its namespace. Also check for trailing slashes or other spelling errors. The Camel component, in particular, will throw very strange errors if a destination can't be found.
- **Debug** Fuse: If you're adding custom code or invoking POJOs hosted in Fuse it's nice to be able to debug these applications. If you set the `SERVICEMIX_DEBUG` system property, Fuse will start in debug mode. You can then use Eclipse (or any other IDE) to attach a debug session to Fuse on port 5005.
- **Strange behavior**: If you debug Fuse, deploy lost of new service units and don't shutdown Fuse correctly you might get some strange behavior. If after a restart this strange behavior doesn't go away, the easiest thing to do to get a fresh Fuse is just stop Fuse and delete the `data/smx` directory from the Fuse installation and start Fuse once again. Now everything will be deployed again from the hot deploy directory.
- **Beware of CLASSPATH issues**: This shouldn't come as a big surprise. If you deploy complete applications to Fuse that have their own dependencies, you're bound to run into classpath issues. There isn't a single solution for this, but if you use the m2eclipse plugin, you can use this plugin to visualize your dependencies to see what causes the problem.

I hope you've enjoyed reading this article and have learned something about Fuse and how it can help in solving your integration issues. Should you have any problems running this scenario just let me know. You can contact met at jos.dirksen@gmail.com.

References:

European accident statement form:
http://paris.franglo.com/classifieds/european_accident_statement_forms_in_english-o345.html
m2eclipse:
http://m2eclipse.codehaus.org
Eclipse download:
http://www.eclipse.org/downloads/
Maven 2:
http://maven.apache.org
FuseSource:
http://fusesource.com/
SOAP-UI:
http://www.soap-ui.org