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 Event 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 **getBody** 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