

Apache Camel



USER GUIDE

Version 2.9.1

Copyright 2007-2011, Apache Software Foundation

Table of Contents

	Table of Contents	ii
Chapter 1	Introduction.....	1
Chapter 1	Quickstart.....	1
Chapter 1	Getting Started	7
Chapter 1	Architecture.....	18
Chapter 1	Enterprise Integration Patterns	35
Chapter 1	Cook Book.....	40
Chapter 1	Tutorials	111
Chapter 1	Language Appendix.....	0
Chapter 1	DataFormat Appendix.....	0
Chapter 1	Pattern Appendix	0
Chapter 1	Component Appendix	0
	Index.....	0

Introduction

Apache Camel is a versatile open-source integration framework (with powerful Bean Integration) based on known Enterprise Integration Patterns. Camel empowers you to define routing and mediation rules in a variety of domain-specific languages, including a Java-based Fluent API, Spring or Blueprint XML Configuration files, and a Scala DSL. This means you get smart completion of routing rules in your IDE, whether in a Java, Scala or XML editor.

Apache Camel uses URIs to work directly with any kind of Transport or messaging model such as HTTP, ActiveMQ, JMS, JBI, SCA, MINA or CXF, as well as pluggable Components and Data Format options. Apache Camel is a small library with minimal dependencies for easy embedding in any Java application. Apache Camel lets you work with the same API regardless which kind of Transport is used - so learn the API once and you can interact with all the Components provided out-of-box.

Apache Camel has powerful Bean Binding and seamless integration with popular frameworks such as Spring, Blueprint and Guice. Camel also has extensive support for unit testing your routes.

The following projects can leverage Apache Camel as a routing and mediation engine:

- Apache ServiceMix - the most popular and powerful distributed open source ESB and JBI container
- Apache ActiveMQ - the most popular and powerful open source message broker
- Apache CXF - a smart web services suite (JAX-WS and JAX-RS)
- Apache Karaf - a small OSGi based runtime in which applications can be deployed
- Apache MINA - a high-performance NIO-driven networking framework

So don't get the hump - try Camel today! 😊



Too many buzzwords - what exactly is Camel?

Okay, so the description above is technology focused.

There's a great discussion about Camel at Stack Overflow. We suggest you view the post, read the comments, and browse the suggested links for more details.

CHAPTER 2

◦ ◦ ◦ ◦

Quickstart

To start using Apache Camel quickly, you can read through some simple examples in this chapter. For readers who would like a more thorough introduction, please skip ahead to Chapter 3.

WALK THROUGH AN EXAMPLE CODE

This mini-guide takes you through the source code of a simple example.

Camel can be configured either by using Spring or directly in Java - which this example does.

This example is available in the `examples\camel-example-jms-file` directory of the Camel distribution.

We start with creating a `CamelContext` - which is a container for Components, Routes etc:

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

There is more than one way of adding a Component to the `CamelContext`. You can add components implicitly - when we set up the routing - as we do here for the `FileComponent`:

```
context.addRoutes(new RouteBuilder() {  
    public void configure() {  
        from("test-jms:queue:test.queue").to("file://test");  
    }  
});
```

or explicitly - as we do here when we add the `JMS Component`:

```
ConnectionFactory connectionFactory = new  
ActiveMQConnectionFactory("vm://localhost?broker.persistent=false");  
// Note we can explicit name the component  
context.addComponent("test-jms",  
JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));
```

The above works with any JMS provider. If we know we are using ActiveMQ we can use an even simpler form using the `activeMQComponent()` method while specifying the `brokerURL` used to connect to ActiveMQ

```
camelContext.addComponent("activemq",
    activeMQComponent("vm://localhost?broker.persistent=false"));
```

In normal use, an external system would be firing messages or events directly into Camel through one of its Components but we are going to use the `ProducerTemplate` which is a really easy way for testing your configuration:

```
ProducerTemplate template = context.createProducerTemplate();
```

Next you **must** start the camel context. If you are using Spring to configure the camel context this is automatically done for you; though if you are using a pure Java approach then you just need to call the `start()` method

```
camelContext.start();
```

This will start all of the configured routing rules.

So after starting the `CamelContext`, we can fire some objects into camel:

```
for (int i = 0; i < 10; i++) {
    template.sendBody("test-jms:queue:test.queue", "Test Message: " + i);
}
```

WHAT HAPPENS?

From the `ProducerTemplate` - we send objects (in this case text) into the `CamelContext` to the Component `test-jms:queue:test.queue`. These text objects will be converted automatically into JMS Messages and posted to a JMS Queue named `test.queue`. When we set up the Route, we configured the `FileComponent` to listen of the `test.queue`.

The `FileComponent` will take messages off the Queue, and save them to a directory named `test`. Every message will be saved in a file that corresponds to its destination and message id.

Finally, we configured our own listener in the Route - to take notifications from the `FileComponent` and print them out as text.

That's it!

If you have the time then use 5 more minutes to Walk through another example that demonstrates the Spring DSL (XML based) routing.

WALK THROUGH ANOTHER EXAMPLE

Introduction

We continue the walk from Walk through an Example. This time we take a closer look at the routing and explains a few pointers so you wont walk into a bear trap, but can enjoy a walk after hours to the local pub for a large beer



First we take a moment to look at the Enterprise Integration Patterns that is the base pattern catalog for integrations. In particular we focus on the Pipes and Filters EIP pattern, that is a central pattern. This is used for: route through a sequence of processing steps, each performing a specific function - much like the Java Servlet Filters.

Pipes and filters

In this sample we want to process a message in a sequence of steps where each steps can perform their specific function. In our example we have a JMS queue for receiving new orders. When an order is received we need to process it in several steps:

- validate
- register
- send confirm email

This can be created in a route like this:

```
<route>
  <from uri="jms:queue:order"/>
  <pipeline>
    <bean ref="validateOrder"/>
    <bean ref="registerOrder"/>
    <bean ref="sendConfirmEmail"/>
  </pipeline>
</route>
```

Where as the bean ref is a reference for a spring bean id, so we define our beans using regular Spring XML as:

```
<bean id="validateOrder" class="com.mycompany.MyOrderValidator"/>
```



Camel 1.4.0 change

In Camel 1.4.0, CamelTemplate has been marked as @deprecated. ProducerTemplate should be used instead and its created from the CamelContext itself.

```
ProducerTemplate template = context.createProducerTemplate();
```

Our validator bean is a plain POJO that has no dependencies to Camel what so ever. So you can implement this POJO as you like. Camel uses rather intelligent Bean Binding to invoke your POJO with the payload of the received message. In this example we will **not** dig into this how this happens. You should return to this topic later when you got some hands on experience with Camel how it can easily bind routing using your existing POJO beans.

So what happens in the route above. Well when an order is received from the JMS queue the message is routed like Pipes and Filters:

1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as input to the registerOrder bean
3. the output from registerOrder bean is sent as input to the sendConfirmEmail bean

Using Camel Components

In the route lets imagine that the registration of the order has to be done by sending data to a TCP socket that could be a big mainframe. As Camel has many Components we will use the camel-mina component that supports TCP connectivity. So we change the route to:

```
<route>
  <from uri="jms:queue:order"/>
  <bean ref="validateOrder"/>
  <to uri="mina:tcp://mainframeip:4444?textline=true"/>
  <bean ref="sendConfirmEmail"/>
</route>
```

What we now have in the route is a to type that can be used as a direct replacement for the bean type. The steps is now:

1. payload from the JMS is sent as input to the validateOrder bean
2. the output from validateOrder bean is sent as text to the mainframe using TCP



Pipeline is default

In the route above we specify pipeline but it can be omitted as its default, so you can write the route as:

```
<route>
  <from uri="jms:queue:order" />
  <bean ref="validateOrder" />
  <bean ref="registerOrder" />
  <bean ref="sendConfirmEmail" />
</route>
```

This is commonly used not to state the pipeline.

An example where the pipeline needs to be used, is when using a multicast and "one" of the endpoints to send to (as a logical group) is a pipeline of other endpoints. For example.

```
<route>
  <from uri="jms:queue:order" />
  <multicast>
    <to uri="log:org.company.log.Category" />
    <pipeline>
      <bean ref="validateOrder" />
      <bean ref="registerOrder" />
      <bean ref="sendConfirmEmail" />
    </pipeline>
  </multicast>
</route>
```

The above sends the order (from `jms:queue:order`) to two locations at the same time, our log component, and to the "pipeline" of beans which goes one to the other. If you consider the opposite, sans the `<pipeline>`

```
<route>
  <from uri="jms:queue:order" />
  <multicast>
    <to uri="log:org.company.log.Category" />
    <bean ref="validateOrder" />
    <bean ref="registerOrder" />
    <bean ref="sendConfirmEmail" />
  </multicast>
</route>
```

you would see that multicast would not "flow" the message from one bean to the next, but rather send the order to all 4 endpoints (1x log, 3x bean) in

parallel, which is not (for this example) what we want. We need the message to flow to the validateOrder, then to the registerOrder, then the sendConfirmEmail so adding the pipeline, provides this facility.

3. the output from mainframe is sent back as input to the sendConfirmEmail bean

What to notice here is that the to is not the end of the route (the world 🤖) in this example it's used in the middle of the Pipes and Filters. In fact we can change the bean types to to as well:

```
<route>
  <from uri="jms:queue:order"/>
  <to uri="bean:validateOrder"/>
  <to uri="mina:tcp://mainframeip:4444?textline=true"/>
  <to uri="bean:sendConfirmEmail"/>
</route>
```

As the to is a generic type we must state in the uri scheme which component it is. So we must write **bean:** for the Bean component that we are using.

Conclusion

This example was provided to demonstrate the Spring DSL (XML based) as opposed to the pure Java DSL from the first example. And as well to point about that the to doesn't have to be the last node in a route graph.

This example is also based on the **in-only** message exchange pattern. What you must understand as well is the **in-out** message exchange pattern, where the caller expects a response. We will look into this in another example.

See also

- Examples
- Tutorials
- User Guide

Getting Started with Apache Camel

THE ENTERPRISE INTEGRATION PATTERNS (EIP) BOOK

The purpose of a "patterns" book is not to advocate new techniques that the authors have invented, but rather to document existing best practices within a particular field. By doing this, the authors of a patterns book hope to spread knowledge of best practices and promote a vocabulary for discussing architectural designs.

One of the most famous patterns books is *Design Patterns: Elements of Reusable Object-oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, commonly known as the "Gang of Four" (GoF) book. Since the publication of *Design Patterns*, many other pattern books, of varying quality, have been written. One famous patterns book is called *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* by Gregor Hohpe and Bobby Woolf. It is common for people to refer to this book by its initials *EIP*. As the subtitle of *EIP* suggests, the book focuses on design patterns for asynchronous messaging systems. The book discusses 65 patterns. Each pattern is given a textual name and most are also given a graphical symbol, intended to be used in architectural diagrams.

THE CAMEL PROJECT

Camel (<http://camel.apache.org>) is an open-source, Java-based project that helps the user implement many of the design patterns in the *EIP* book. Because Camel implements many of the design patterns in the *EIP* book, it would be a good idea for people who work with Camel to have the *EIP* book as a reference.

ONLINE DOCUMENTATION FOR CAMEL

The documentation is all under the Documentation category on the right-side menu of the Camel website (also available in PDF form. Camel-related books are also available, in particular the Camel in Action book, presently serving as the Camel bible--it has a free Chapter One (pdf), which is highly recommended to read to get more familiar with Camel.

A useful tip for navigating the online documentation

The breadcrumbs at the top of the online Camel documentation can help you navigate between parent and child subsections.

For example, if you are on the "Languages" documentation page then the left-hand side of the reddish bar contains the following links.

```
Apache Camel > Documentation > Architecture > Languages
```

As you might expect, clicking on "Apache Camel" takes you back to the home page of the Apache Camel project, and clicking on "Documentation" takes you to the main documentation page. You can interpret the "Architecture" and "Languages" buttons as indicating you are in the "Languages" section of the "Architecture" chapter. Adding browser bookmarks to pages that you frequently reference can also save time.

ONLINE JAVADOC DOCUMENTATION

The Apache Camel website provides Javadoc documentation. It is important to note that the Javadoc documentation is spread over several *independent* Javadoc hierarchies rather than being all contained in a single Javadoc hierarchy. In particular, there is one Javadoc hierarchy for the *core* APIs of Camel, and a separate Javadoc hierarchy for each component technology supported by Camel. For example, if you will be using Camel with ActiveMQ and FTP then you need to look at the Javadoc hierarchies for the core API and Spring API.

CONCEPTS AND TERMINOLOGY FUNDAMENTAL TO CAMEL

In this section some of the concepts and terminology that are fundamental to Camel are explained. This section is not meant as a complete Camel tutorial, but as a first step in that direction.

Endpoint

The term *endpoint* is often used when talking about inter-process communication. For example, in client-server communication, the client is one endpoint and the server is the other endpoint. Depending on the context, an endpoint might refer to an *address*, such as a host:port pair for TCP-based communication, or it might refer to a *software entity* that is contactable at that address. For example, if somebody uses "www.example.com:80" as an example of an endpoint, they might be referring to the actual port at that host name (that is, an address), or they might be referring to the web server (that is, software contactable at that address). Often, the distinction between the address and software contactable at that address is not an important one.

Some middleware technologies make it possible for several software entities to be contactable at the same physical address. For example, CORBA is an object-oriented, remote-procedure-call (RPC) middleware standard. If a CORBA server process contains several objects then a client can communicate with any of these objects at the same *physical* address (host:port), but a client communicates with a particular object via that object's *logical* address (called an *IOR* in CORBA terminology), which consists of the physical address (host:port) plus an id that uniquely identifies the object within its server process. (An IOR contains some additional information that is not relevant to this present discussion.) When talking about CORBA, some people may use the term "endpoint" to refer to a CORBA server's *physical address*, while other people may use the term to refer to the *logical address* of a single CORBA object, and other people still might use the term to refer to any of the following:

- The physical address (host:port) of the CORBA server process
- The logical address (host:port plus id) of a CORBA object.
- The CORBA server process (a relatively heavyweight software entity)
- A CORBA object (a lightweight software entity)

Because of this, you can see that the term *endpoint* is ambiguous in at least two ways. First, it is ambiguous because it might refer to an address or to a software entity contactable at that address. Second, it is ambiguous in the *granularity* of what it refers to: a heavyweight versus lightweight software entity, or physical address versus logical address. It is useful to understand that different people use the term *endpoint* in slightly different (and hence ambiguous) ways because Camel's usage of this term might be different to whatever meaning you had previously associated with the term.

Camel provides out-of-the-box support for endpoints implemented with many different communication technologies. Here are some examples of the Camel-supported endpoint technologies.

- A JMS queue.
- A web service.

- A file. A file may sound like an unlikely type of endpoint, until you realize that in some systems one application might write information to a file and, later, another application might read that file.
- An FTP server.
- An email address. A client can send a message to an email address, and a server can read an incoming message from a mail server.
- A POJO (plain old Java object).

In a Camel-based application, you create (Camel wrappers around) some endpoints and connect these endpoints with *routes*, which I will discuss later in Section 4.8 ("Routes, RouteBuilders and Java DSL"). Camel defines a Java interface called `Endpoint`. Each Camel-supported endpoint has a class that implements this `Endpoint` interface. As I discussed in Section 3.3 ("Online Javadoc documentation"), Camel provides a separate Javadoc hierarchy for each communications technology supported by Camel. Because of this, you will find documentation on, say, the `JmsEndpoint` class in the JMS Javadoc hierarchy, while documentation for, say, the `FtpEndpoint` class is in the FTP Javadoc hierarchy.

CamelContext

A `CamelContext` object represents the Camel runtime system. You typically have one `CamelContext` object in an application. A typical application executes the following steps.

1. Create a `CamelContext` object.
2. Add endpoints and possibly Components, which are discussed in Section 4.5 ("Components") to the `CamelContext` object.
3. Add routes to the `CamelContext` object to connect the endpoints.
4. Invoke the `start()` operation on the `CamelContext` object. This starts Camel-internal threads that are used to process the sending, receiving and processing of messages in the endpoints.
5. Eventually invoke the `stop()` operation on the `CamelContext` object. Doing this gracefully stops all the endpoints and Camel-internal threads.

Note that the `CamelContext.start()` operation does not block indefinitely. Rather, it starts threads internal to each Component and Endpoint and then `start()` returns. Conversely, `CamelContext.stop()` waits for all the threads internal to each Endpoint and Component to terminate and then `stop()` returns.

If you neglect to call `CamelContext.start()` in your application then messages will not be processed because internal threads will not have been created.

If you neglect to call `CamelContext.stop()` before terminating your application then the application may terminate in an inconsistent state. If

you neglect to call `CamelContext.stop()` in a JUnit test then the test may fail due to messages not having had a chance to be fully processed.

CamelTemplate

Camel used to have a class called `CamelClient`, but this was renamed to be `CamelTemplate` to be similar to a naming convention used in some other open-source projects, such as the `TransactionTemplate` and `JmsTemplate` classes in Spring.

The `CamelTemplate` class is a thin wrapper around the `CamelContext` class. It has methods that send a `Message` or `Exchange` to an `Endpoint`. This provides a way to enter messages into source endpoints, so that the messages will move along routes to destination endpoints.

The Meaning of URL, URI, URN and IRI

Some Camel methods take a parameter that is a *URI* string. Many people know that a URI is "something like a URL" but do not properly understand the relationship between URI and URL, or indeed its relationship with other acronyms such as IRI and URN.

Most people are familiar with *URLs* (uniform resource locators), such as "http://...", "ftp://...", "mailto:...". Put simply, a URL specifies the *location* of a resource.

A *URI* (uniform resource identifier) is a URL or a URN. So, to fully understand what URI means, you need to first understand what is a URN.

URN is an acronym for *uniform resource name*. There are many "unique identifier" schemes in the world, for example, ISBNs (globally unique for books), social security numbers (unique within a country), customer numbers (unique within a company's customers database) and telephone numbers. Each "unique identifier" scheme has its own notation. A URN is a wrapper for different "unique identifier" schemes. The syntax of a URN is "urn:<scheme-name>:<unique-identifier>". A URN uniquely identifies a *resource*, such as a book, person or piece of equipment. By itself, a URN does not specify the *location* of the resource. Instead, it is assumed that a *registry* provides a mapping from a resource's URN to its location. The URN specification does not state what form a registry takes, but it might be a database, a server application, a wall chart or anything else that is convenient. Some hypothetical examples of URNs are "urn:employee:08765245", "urn:customer:uk:3458:hul8" and "urn:foo:0000-0000-9E59-0000-5E-2". The <scheme-name> ("employee", "customer" and "foo" in these examples) part

of a URN implicitly defines how to parse and interpret the <unique-identifier> that follows it. An arbitrary URN is meaningless unless: (1) you know the semantics implied by the <scheme-name>, and (2) you have access to the registry appropriate for the <scheme-name>. A registry does not have to be public or globally accessible. For example, "urn:employee:08765245" might be meaningful only within a specific company.

To date, URNs are not (yet) as popular as URLs. For this reason, URI is widely misused as a synonym for URL.

IRI is an acronym for *internationalized resource identifier*. An IRI is simply an internationalized version of a URI. In particular, a URI can contain letters and digits in the US-ASCII character set, while a IRI can contain those same letters and digits, and *also* European accented characters, Greek letters, Chinese ideograms and so on.

Components

Component is confusing terminology; *EndpointFactory* would have been more appropriate because a Component is a factory for creating Endpoint instances. For example, if a Camel-based application uses several JMS queues then the application will create one instance of the JmsComponent class (which implements the Component interface), and then the application invokes the createEndpoint() operation on this JmsComponent object several times. Each invocation of JmsComponent.createEndpoint() creates an instance of the JmsEndpoint class (which implements the Endpoint interface). Actually, application-level code does not invoke Component.createEndpoint() directly. Instead, application-level code normally invokes CamelContext.getEndpoint(); internally, the CamelContext object finds the desired Component object (as I will discuss shortly) and then invokes createEndpoint() on it.

Consider the following code.

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

The parameter to getEndpoint() is a URI. The URI *prefix* (that is, the part before ":",) specifies the name of a component. Internally, the CamelContext object maintains a mapping from names of components to Component objects. For the URI given in the above example, the CamelContext object would probably map the pop3 prefix to an instance of the MailComponent class. Then the CamelContext object invokes createEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword") on that MailComponent object. The createEndpoint() operation splits the URI into its component parts and uses these parts to create and configure an

Endpoint object.

In the previous paragraph, I mentioned that a `CamelContext` object maintains a mapping from component names to `Component` objects. This raises the question of how this map is populated with named `Component` objects. There are two ways of populating the map. The first way is for application-level code to invoke `CamelContext.addComponent(String componentName, Component component)`. The example below shows a single `MailComponent` object being registered in the map under 3 different names.

```
Component mailComponent = new org.apache.camel.component.mail.MailComponent();
myCamelContext.addComponent("pop3", mailComponent);
myCamelContext.addComponent("imap", mailComponent);
myCamelContext.addComponent("smtp", mailComponent);
```

The second (and preferred) way to populate the map of named `Component` objects in the `CamelContext` object is to let the `CamelContext` object perform lazy initialization. This approach relies on developers following a convention when they write a class that implements the `Component` interface. I illustrate the convention by an example. Let's assume you write a class called `com.example.myproject.FooComponent` and you want Camel to automatically recognize this by the name "foo". To do this, you have to write a properties file called "META-INF/services/org/apache/camel/component/foo" (without a ".properties" file extension) that has a single entry in it called `class`, the value of which is the fully-scoped name of your class. This is shown below.

Listing 1. META-INF/services/org/apache/camel/component/foo

```
class=com.example.myproject.FooComponent
```

If you want Camel to also recognize the class by the name "bar" then you write another properties file in the same directory called "bar" that has the same contents. Once you have written the properties file(s), you create a jar file that contains the `com.example.myproject.FooComponent` class and the properties file(s), and you add this jar file to your `CLASSPATH`. Then, when application-level code invokes `createEndpoint("foo:...")` on a `CamelContext` object, Camel will find the "foo" properties file on the `CLASSPATH`, get the value of the `class` property from that properties file, and use reflection APIs to create an instance of the specified class.

As I said in Section 4.1 ("Endpoint"), Camel provides out-of-the-box support for numerous communication technologies. The out-of-the-box support consists of classes that implement the `Component` interface plus properties files that enable a `CamelContext` object to populate its map of named `Component` objects.

Earlier in this section I gave the following example of calling `CamelContext.getEndpoint()`.

```
myCamelContext.getEndpoint("pop3://john.smith@mailserv.example.com?password=myPassword");
```

When I originally gave that example, I said that the parameter to `getEndpoint()` was a URI. I said that because the online Camel documentation and the Camel source code both claim the parameter is a URI. In reality, the parameter is restricted to being a URL. This is because when Camel extracts the component name from the parameter, it looks for the first ":", which is a simplistic algorithm. To understand why, recall from Section 4.4 ("The Meaning of URL, URI, URN and IRI") that a URI can be a URL or a URN. Now consider the following calls to `getEndpoint`.

```
myCamelContext.getEndpoint("pop3:...");  
myCamelContext.getEndpoint("jms:...");  
myCamelContext.getEndpoint("urn:foo:...");  
myCamelContext.getEndpoint("urn:bar:...");
```

Camel identifies the components in the above example as "pop3", "jms", "urn" and "urn". It would be more useful if the latter components were identified as "urn:foo" and "urn:bar" or, alternatively, as "foo" and "bar" (that is, by skipping over the "urn:" prefix). So, in practice you must identify an endpoint with a URL (a string of the form "<scheme>:...") rather than with a URN (a string of the form "urn:<scheme>:..."). This lack of proper support for URNs means the you should consider the parameter to `getEndpoint()` as being a URL rather than (as claimed) a URI.

Message and Exchange

The Message interface provides an abstraction for a single message, such as a request, reply or exception message.

There are concrete classes that implement the Message interface for each Camel-supported communications technology. For example, the `JmsMessage` class provides a JMS-specific implementation of the Message interface. The public API of the Message interface provides get- and set-style methods to access the *message id*, *body* and individual *header* fields of a message.

The Exchange interface provides an abstraction for an exchange of messages, that is, a request message and its corresponding reply or exception message. In Camel terminology, the request, reply and exception messages are called *in*, *out* and *fault* messages.

There are concrete classes that implement the Exchange interface for each Camel-supported communications technology. For example, the `JmsExchange` class provides a JMS-specific implementation of the Exchange interface. The

public API of the Exchange interface is quite limited. This is intentional, and it is expected that each class that implements this interface will provide its own technology-specific operations.

Application-level programmers rarely access the Exchange interface (or classes that implement it) directly. However, many classes in Camel are generic types that are instantiated on (a class that implements) Exchange. Because of this, the Exchange interface appears a lot in the generic signatures of classes and methods.

Processor

The Processor interface represents a class that processes a message. The signature of this interface is shown below.

Listing 1. Processor

```
package org.apache.camel;
public interface Processor {
    void process(Exchange exchange) throws Exception;
}
```

Notice that the parameter to the `process()` method is an Exchange rather than a Message. This provides flexibility. For example, an implementation of this method initially might call `exchange.getIn()` to get the input message and process it. If an error occurs during processing then the method can call `exchange.setException()`.

An application-level developer might implement the Processor interface with a class that executes some business logic. However, there are many classes in the Camel library that implement the Processor interface in a way that provides support for a design pattern in the EIP book. For example, `ChoiceProcessor` implements the message router pattern, that is, it uses a cascading if-then-else statement to route a message from an input queue to one of several output queues. Another example is the `FilterProcessor` class which discards messages that do not satisfy a stated *predicate* (that is, condition).

Routes, RouteBuilders and Java DSL

A *route* is the step-by-step movement of a Message from an input queue, through arbitrary types of decision making (such as filters and routers) to a destination queue (if any). Camel provides two ways for an application developer to specify routes. One way is to specify route information in an XML file. A discussion of that approach is outside the scope of this document. The other way is through what Camel calls a Java *DSL* (domain-specific language).

Introduction to Java DSL

For many people, the term "domain-specific language" implies a compiler or interpreter that can process an input file containing keywords and syntax specific to a particular domain. This is *not* the approach taken by Camel. Camel documentation consistently uses the term "Java DSL" instead of "DSL", but this does not entirely avoid potential confusion. The Camel "Java DSL" is a class library that can be used in a way that looks almost like a DSL, except that it has a bit of Java syntactic baggage. You can see this in the example below. Comments afterwards explain some of the constructs used in the example.

Listing 1. Example of Camel's "Java DSL"

```
RouteBuilder builder = new RouteBuilder() {
    public void configure() {
        from("queue:a").filter(header("foo").isEqualTo("bar")).to("queue:b");
        from("queue:c").choice()
            .when(header("foo").isEqualTo("bar")).to("queue:d")
            .when(header("foo").isEqualTo("cheese")).to("queue:e")
            .otherwise().to("queue:f");
    }
};
CamelContext myCamelContext = new DefaultCamelContext();
myCamelContext.addRoutes(builder);
```

The first line in the above example creates an object which is an instance of an anonymous subclass of `RouteBuilder` with the specified `configure()` method.

The `CamelContext.addRoutes(RouterBuilder builder)` method invokes `builder.setContext(this)` so the `RouteBuilder` object knows which `CamelContext` object it is associated with and then invokes `builder.configure()`. The body of `configure()` invokes methods such as `from()`, `filter()`, `choice()`, `when()`, `isEqualTo()`, `otherwise()` and `to()`. The `RouteBuilder.from(String uri)` method invokes `getEndpoint(uri)` on the `CamelContext` associated with the `RouteBuilder` object to get the specified `Endpoint` and then puts a `FromBuilder` "wrapper" around this `Endpoint`. The `FromBuilder.filter(Predicate predicate)` method creates a `FilterProcessor` object for the `Predicate` (that is, condition) object built from the `header("foo").isEqualTo("bar")` expression. In this way, these operations incrementally build up a `Route` object (with a `RouteBuilder` wrapper around it) and add it to the `CamelContext` object associated with the `RouteBuilder`.

Critique of Java DSL

The online Camel documentation compares Java DSL favourably against the alternative of configuring routes and endpoints in a XML-based Spring configuration file. In particular, Java DSL is less verbose than its XML counterpart. In addition, many integrated development environments (IDEs) provide an auto-completion feature in their editors. This auto-completion feature works with Java DSL, thereby making it easier for developers to write Java DSL.

However, there is another option that the Camel documentation neglects to consider: that of writing a parser that can process DSL stored in, say, an external file. Currently, Camel does not provide such a DSL parser, and I do not know if it is on the "to do" list of the Camel maintainers. I think that a DSL parser would offer a significant benefit over the current Java DSL. In particular, the DSL would have a syntactic definition that could be expressed in a relatively short BNF form. The effort required by a Camel user to learn how to use DSL by reading this BNF would almost certainly be significantly less than the effort currently required to study the API of the RouterBuilder classes.

Continue Learning about Camel

Return to the main Getting Started page for additional introductory reference information.

Architecture

Camel uses a Java based Routing Domain Specific Language (DSL) or an Xml Configuration to configure routing and mediation rules which are added to a CamelContext to implement the various Enterprise Integration Patterns.

At a high level Camel consists of a CamelContext which contains a collection of Component instances. A Component is essentially a factory of Endpoint instances. You can explicitly configure Component instances in Java code or an IoC container like Spring or Guice, or they can be auto-discovered using URIs.

An Endpoint acts rather like a URI or URL in a web application or a Destination in a JMS system; you can communicate with an endpoint; either sending messages to it or consuming messages from it. You can then create a Producer or Consumer on an Endpoint to exchange messages with it.

The DSL makes heavy use of pluggable Languages to create an Expression or Predicate to make a truly powerful DSL which is extensible to the most suitable language depending on your needs. The following languages are supported

- Bean Language for using Java for expressions
- Constant
- the unified EL from JSP and JSF
- Header
- XPath
- Mvel
- OGNL
- Ref Language
- Property
- Scala DSL
- Scripting Languages such as
 - BeanShell
 - JavaScript
 - Groovy
 - Python
 - PHP
 - Ruby
- Simple

- File Language
- Spring Expression Language
- SQL
- Tokenizer
- XPath
- XQuery

Most of these languages is also supported used as Annotation Based Expression Language.

For a full details of the individual languages see the Language Appendix

URIS

Camel makes extensive use of URIs to allow you to refer to endpoints which are lazily created by a Component if you refer to them within Routes

Current Supported URIs

Component / ArtifactId / URI	Description
AHC / camel-ahc <code>ahc:hostname:[port]</code>	To call external HTTP services using Async Http Client
AMQP / camel-amqp <code>amqp:[topic:]destinationName</code>	For Messaging with AMQP protocol
APNS / camel-apns <code>apns:notify[?options]</code>	For sending notifications to Apple iOS devices
Atom / camel-atom <code>atom:uri</code>	Working with Apache Abdera for atom integration, such as consuming an atom feed.
AWS-SDB / camel-aws <code>aws-sdb://domainName[?options]</code>	For working with Amazon's SimpleDB (SDB).

AWS-SES / camel-aws

```
aws-ses://from[?options]
```

For working with Amazon's Simple Email Service (SES).

AWS-SNS / camel-aws

```
aws-sns://topicname[?options]
```

For Messaging with Amazon's Simple Notification Service (SNS).

AWS-SQS / camel-aws

```
aws-sqs://queueName[?options]
```

For Messaging with Amazon's Simple Queue Service (SQS).

AWS-S3 / camel-aws

```
aws-s3://bucketname[?options]
```

For working with Amazon's Simple Storage Service (S3).

Bean / camel-core

```
bean:beanName[?method=someMethod]
```

Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).

Bean Validation / camel-bean-validator

```
bean-validator:something
```

Validates the payload of a message using the Java Validation API (JSR 303 and JAXP Validation) and its reference implementation Hibernate Validator

Browse / camel-core

```
browse:someName
```

Provides a simple `BrowsableEndpoint` which can be useful for testing, visualisation tools or debugging. The exchanges sent to the endpoint are all available to be browsed.

Cache / camel-cache

```
cache://cachename[?options]
```

The cache component facilitates creation of caching endpoints and processors using EHCache as the cache implementation.

Class / camel-core

```
class:className[?method=someMethod]
```

Uses the Bean Binding to bind message exchanges to beans in the Registry. Is also used for exposing and invoking POJO (Plain Old Java Objects).

Cometd / camel-cometd

```
cometd://host:port/channelname
```

Used to deliver messages using the jetty cometd implementation of the bayeux protocol

Context / camel-context

```
context:camelContextId:localEndpointName
```

Used to refer to endpoints within a separate CamelContext to provide a simple black box composition approach so that routes can be combined into a CamelContext and then used as a black box component inside other routes in other CamelContexts

Crypto (Digital Signatures) / camel-crypto

```
crypto:sign:name[?options]  
crypto:verify:name[?options]
```

Used to sign and verify exchanges using the Signature Service of the Java Cryptographic Extension.

CXF / camel-cxf

```
cxf:address[?serviceClass=...]
```

Working with Apache CXF for web services integration

CXF Bean / camel-cxf

```
cxf:bean name
```

Process the exchange using a JAX WS or JAX RS annotated bean from the registry. Requires less configuration than the above CXF Component

CXFRS / camel-cxf

```
cxfrs:address[?resourcesClasses=...]
```

Working with Apache CXF for REST services integration

DataSet / camel-core

```
dataset:name
```

For load & soak testing the DataSet provides a way to create huge numbers of messages for sending to Components or asserting that they are consumed correctly

Direct / camel-core

```
direct:name
```

Synchronous call to another endpoint

DNS / camel-dns

```
dns:operation
```

To lookup domain information and run DNS queries using DNSJava

EJB / camel-ejb

```
ejb:ejbName[?method=someMethod]
```

Uses the Bean Binding to bind message exchanges to EJBs. It works like the Bean component but just for accessing EJBs. Supports EJB 3.0 onwards.

Event / camel-spring

```
event://default  
spring-event://default
```

Working with Spring ApplicationEvents

EventAdmin / camel-eventadmin

```
eventadmin:topic
```

Receiving OSGi EventAdmin events

Exec / camel-exec

```
exec://executable[?options]
```

For executing system commands

File / camel-core

```
file://nameOfFileOrDirectory
```

Sending messages to a file or polling a file or directory.
Camel 1.x use this link File.

Flatpack / camel-flatpack

```
flatpack:[fixed|delim]:configFile
```

Processing fixed width or delimited files or messages using the FlatPack library

FreeMarker / camel-freemarker

```
freemarker:someTemplateResource
```

Generates a response using a FreeMarker template

FTP / camel-ftp

```
ftp://host[:port]/fileName
```

Sending and receiving files over FTP. **Camel 1.x use this link FTP.**

FTPS / camel-ftp

```
ftps://host[:port]/fileName
```

Sending and receiving files over FTP Secure (TLS and SSL).

GAuth / camel-gae

```
gauth://name[?options]
```

Used by web applications to implement an OAuth consumer. See also Camel Components for Google App Engine.

GHttp / camel-gae

```
ghttp://hostname[:port][/path][?options]
ghttp:///path[?options]
```

Provides connectivity to the URL fetch service of Google App Engine but can also be used to receive messages from servlets. See also Camel Components for Google App Engine.

GLogin / camel-gae

```
glogin://hostname[:port][?options]
```

Used by Camel applications outside Google App Engine (GAE) for programmatic login to GAE applications. See also Camel Components for Google App Engine.

GTask / camel-gae

```
gtask://queue-name
```

Supports asynchronous message processing on Google App Engine by using the task queueing service as message queue. See also Camel Components for Google App Engine.

GMail / camel-gae

```
gmail://user@gmail.com[?options]
gmail://user@googlemail.com[?options]
```

Supports sending of emails via the mail service of Google App Engine. See also Camel Components for Google App Engine.

Hazelcast / camel-hazelcast

```
hazelcast://[type]:cachename[?options]
```

Hazelcast is a data grid entirely implemented in Java (single jar). This component supports map, multimap, seda, queue, set, atomic number and simple cluster support.

HDFS / camel-hdfs

```
hdfs://path[?options]
```

For reading/writing from/to an HDFS filesystem

HL7 / camel-hl7

```
mina:tcp://hostname[:port]
```

For working with the HL7 MLLP protocol and the HL7 model using the HAPI library

HTTP / camel-http

```
http://hostname[:port]
```

For calling out to external HTTP servers using Apache HTTP Client 3.x

HTTP4 / camel-http4

```
http4://hostname[:port]
```

For calling out to external HTTP servers using Apache HTTP Client 4.x

iBatis / camel-ibatis

```
ibatis://statementName
```

Performs a query, poll, insert, update or delete in a relational database using Apache iBatis

IMap / camel-mail

```
imap://hostname[:port]
```

Receiving email using IMap

IRC / camel-irc

```
irc:host[:port]/#room
```

For IRC communication

JavaSpace / camel-javaspaces

```
javaspaces:jini://host?spaceName=mySpace?...
```

Sending and receiving messages through JavaSpace

JBI / servicemix-camel

```
jbi:serviceName
```

For JBI integration such as working with Apache ServiceMix

jclouds / jclouds

```
jclouds:[blobstore|computeservice]:provider
```

For interacting with cloud compute & blobstore service via jclouds

JCR / camel-jcr

```
jcr://user:password@repository/path/to/node
```

Storing a message in a JCR compliant repository like Apache Jackrabbit

JDBC / camel-jdbc

```
jdbc:dataSourceName?options
```

For performing JDBC queries and operations

Jetty / camel-jetty

```
jetty:url
```

For exposing services over HTTP

JMS / camel-jms

```
jms:[topic:]destinationName
```

Working with JMS providers

JMX / camel-jmx

```
jmx://platform?options
```

For working with JMX notification listeners

JPA / camel-jpa

```
jpa://entityName
```

For using a database as a queue via the JPA specification for working with OpenJPA, Hibernate or TopLink

JT/400 / camel-jt400

```
jt400://user:pwd@system/<path_to_dtaq>
```

For integrating with data queues on an AS/400 (aka System i, IBM i, i5, ...) system

Kestrel / camel-kestrel

```
kestrel://[addresslist/]queuename[?options]
```

For producing to or consuming from Kestrel queues

Krati / camel-krati

```
krati://[path to datastore/][?options]
```

For producing to or consuming to Krati datastores

Language / camel-core

```
language://languageName[:script][?options]
```

Executes Languages scripts

LDAP / camel-ldap

```
ldap:host[:port]?base=... [&scope=<scope>]
```

Performing searches on LDAP servers (<scope> must be one of object|onelevel|subtree)

Log / camel-core

```
log:loggingCategory[?level=ERROR]
```

Uses Jakarta Commons Logging to log the message exchange to some underlying logging system like log4j

Lucene / camel-lucene

```
lucene:searcherName:insert[?analyzer=<analyzer>]  
lucene:searcherName:query[?analyzer=<analyzer>]
```

Uses Apache Lucene to perform Java-based indexing and full text based searches using advanced analysis/tokenization capabilities

Mail / camel-mail

```
mail://user-info@host:port
```

Sending and receiving email

MINA / camel-mina

```
[tcp|udp|vm]:host[:port]
```

Working with Apache MINA

Mock / camel-core

```
mock:name
```

For testing routes and mediation rules using mocks

MSV / camel-msv

```
msv:someLocalOrRemoteResource
```

Validates the payload of a message using the MSV Library

MyBatis / camel-mybatis

```
mybatis://statementName
```

Performs a query, poll, insert, update or delete in a relational database using MyBatis

Nagios / camel-nagios

```
nagios://host[:port]?options
```

Sending passive checks to Nagios using JSendNSCA

Netty / camel-netty

```
netty:tcp//host[:port]?options  
netty:udp//host[:port]?options
```

Working with TCP and UDP protocols using Java NIO based capabilities offered by the JBoss Netty community project

Pax-Logging / camel-paxlogging

```
paxlogging:appender
```

Receiving Pax-Logging events in OSGi

POP / camel-mail

```
pop3://user-info@host:port
```

Receiving email using POP3 and JavaMail

Printer / camel-printer

```
lpr://host:port/path/to/printer[?options]
```

The printer component facilitates creation of printer endpoints to local, remote and wireless printers. The endpoints provide the ability to print camel directed payloads when utilized on camel routes.

Properties / camel-core

```
properties://key[?options]
```

The properties component facilitates using property placeholders directly in endpoint uri definitions.

Quartz / camel-quartz

```
quartz://groupName/timerName
```

Provides a scheduled delivery of messages using the Quartz scheduler

Quickfix / camel-quickfix

```
quickfix-server:config file  
quickfix-client:config-file
```

Implementation of the QuickFix for Java engine which allow to send/receive FIX messages

Ref / camel-core

```
ref:name
```

Component for lookup of existing endpoints bound in the Registry.

Restlet / camel-restlet

```
restlet:restletUrl[?options]
```

Component for consuming and producing Restful resources using Restlet

RMI / camel-rmi

```
rmi://host[:port]
```

Working with RMI

RNC / camel-jing

```
rnc:/relativeOrAbsoluteUri
```

Validates the payload of a message using RelaxNG Compact Syntax

RNG / camel-jing

```
rng:/relativeOrAbsoluteUri
```

Validates the payload of a message using RelaxNG

Routebox / camel-routebox

```
routebox:routeboxName[?options]
```

Facilitates the creation of specialized endpoints that offer encapsulation and a strategy/map based indirection service to a collection of camel routes hosted in an automatically created or user injected camel context

RSS / camel-rss

```
rss:uri
```

Working with ROME for RSS integration, such as consuming an RSS feed.

SEDA / camel-core

```
seda:name
```

Asynchronous call to another endpoint in the same Camel Context

SERVLET / camel-servlet

```
servlet:uri
```

For exposing services over HTTP through the servlet which is deployed into the Web container.

SFTP / camel-ftp

```
sftp://host[:port]/fileName
```

Sending and receiving files over SFTP (FTP over SSH).
Camel 1.x use this link FTP.

Sip / camel-sip

```
sip://user@host[:port]?[options]  
sips://user@host[:port]?[options]
```

Publish/Subscribe communication capability using the Telecom SIP protocol. RFC3903 - Session Initiation Protocol (SIP) Extension for Event

SMTP / camel-mail

```
smtp://user-info@host[:port]
```

Sending email using SMTP and JavaMail

SMPP / camel-smpp

```
smpp://user-info@host[:port]?options
```

To send and receive SMS using Short Messaging Service Center using the JSMPP library

SNMP / camel-snmp

```
snmp://host[:port]?options
```

Polling OID values and receiving traps using SNMP via SNMP4J library

Solr / camel-solr

```
solr://host[:port]/solr?[options]
```

Uses the Solrj client API to interface with an Apache Lucene Solr server

SpringIntegration / camel-spring-integration

```
spring-integration:defaultChannelName
```

The bridge component of Camel and Spring Integration

Spring Web Services / camel-spring-ws

```
spring-ws:[mapping-type:]address?[options]
```

Client-side support for accessing web services, and server-side support for creating your own contract-first web services using Spring Web Services

SQL / camel-sql

```
sql:select * from table where id=#
```

Performing SQL queries using JDBC

SSH component / camel-ssh

```
ssh:[username[:password]@]host[:port][?options]
```

For sending commands to a SSH server

StAX / camel-stax

```
stax:contentHandlerClassName
```

Process messages through a SAX ContentHandler.

Stream / camel-stream

```
stream:[in|out|err|file]
```

Read or write to an input/output/error/file stream rather like unix pipes

StringTemplate / camel-stringtemplate

```
string-template:someTemplateResource
```

Generates a response using a String Template

TCP / camel-mina

```
mina:tcp://host:port
```

Working with TCP protocols using Apache MINA

Test / camel-spring

```
test:expectedMessagesEndpointUri
```

Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint

Timer / camel-core

```
timer://name
```

A timer endpoint

Twitter / camel-twitter

```
twitter://[endpoint]?[options]
```

A twitter endpoint

UDP / camel-mina

```
mina:udp://host:port
```

Working with UDP protocols using Apache MINA

Validation / camel-core (camel-spring for Camel 2.8 or older)

```
validation:someLocalOrRemoteResource
```

Validates the payload of a message using XML Schema and JAXP Validation

Velocity / camel-velocity

```
velocity:someTemplateResource
```

Generates a response using an Apache Velocity template

VM / camel-core

```
vm:name
```

Asynchronous call to another endpoint in the same JVM

XMPP / camel-xmpp

```
xmpp://host:port/room
```

Working with XMPP and Jabber

XQuery / camel-saxon

```
xquery:someXQueryResource
```

Generates a response using an XQuery template

XSLT / camel-core (camel-spring for Camel 2.8 or older)

```
xslt:someTemplateResource
```

Generates a response using an XSLT template

Zookeeper / camel-zookeeper

```
zookeeper://host:port/path
```

Working with ZooKeeper cluster(s)

URI's for external components

Other projects and companies have also created Camel components to integrate additional functionality into Camel. These components may be provided under licenses that are not compatible with the Apache License, use libraries that are not compatible, etc... These components are not supported by the Camel team, but we provide links here to help users find the additional functionality.

Component / ArtifactId / URI	License	Description
ActiveMQ / activemq-camel <code>activemq:[topic:]destinationName</code>	Apache	For JMS Messaging with Apache ActiveMQ
ActiveMQ Journal / activemq-core <code>activemq.journal:directory-on-filesystem</code>	Apache	Uses ActiveMQ's fast disk journaling implementation to store message bodies in a rolling log file
Db4o / camel-db4o in camel-extra <code>db4o://className</code>	GPL	For using a db4o datastore as a queue via the db4o library
Esper / camel-esper in camel-extra <code>esper:name</code>	GPL	Working with the Esper Library for Event Stream Processing
Hibernate / camel-hibernate in camel-extra <code>hibernate://entityName</code>	GPL	For using a database as a queue via the Hibernate library
NMR / servicemix-nmr <code>nmr://serviceName</code>	Apache	Integration with the Normalized Message Router BUS in ServiceMix 4.x

Scalate / scalate-camel

```
scalate:templateName
```

Apache

Uses the given Scalate template to transform the message

Smooks / camel-smooks in camel-extra.

```
unmarshal(edi)
```

GPL

For working with EDI parsing using the Smooks library. This component is **deprecated** as Smooks now provides Camel integration out of the box

For a full details of the individual components see the Component Appendix

Enterprise Integration Patterns

Camel supports most of the Enterprise Integration Patterns from the excellent book of the same name by Gregor Hohpe and Bobby Woolf. Its a highly recommended book, particularly for users of Camel.

PATTERN INDEX

There now follows a list of the Enterprise Integration Patterns from the book along with examples of the various patterns using Apache Camel

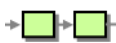
Messaging Systems



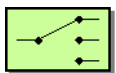
Message Channel How does one application communicate with another using messaging?



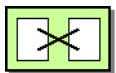
Message How can two applications connected by a message channel exchange a piece of information?



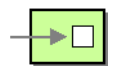
Pipes and Filters How can we perform complex processing on a message while maintaining independence and flexibility?



Message Router How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?








Message Translator How can systems using different data formats communicate with each other using messaging?


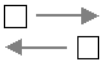
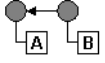



Message Endpoint How does an application connect to a messaging channel to send and receive messages?

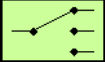

Messaging Channels

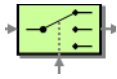
	Point to Point Channel	How can the caller be sure that exactly one receiver will receive the document or perform the call?
	Publish Subscribe Channel	How can the sender broadcast an event to all interested receivers?
	Dead Letter Channel	What will the messaging system do with a message it cannot deliver?
	Guaranteed Delivery	How can the sender make sure that a message will be delivered, even if the messaging system fails?
	Message Bus	What is an architecture that enables separate applications to work together, but in a de-coupled fashion such that applications can be easily added or removed without affecting the others?

Message Construction

	Event Message	How can messaging be used to transmit events from one application to another?
	Request Reply	When an application sends a message, how can it get a response from the receiver?
	Correlation Identifier	How does a requestor that has received a reply know which request this is the reply for?
	Return Address	How does a replier know where to send the reply?

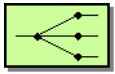
Message Routing

	Content Based Router	How do we handle a situation where the implementation of a single logical function (e.g., inventory check) is spread across multiple physical systems?
	Message Filter	How can a component avoid receiving uninteresting messages?



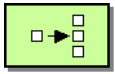
Dynamic Router

How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency?



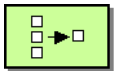
Recipient List

How do we route a message to a list of (static or dynamically) specified recipients?



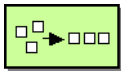
Splitter

How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?



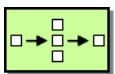
Aggregator

How do we combine the results of individual, but related messages so that they can be processed as a whole?



Resequencer

How can we get a stream of related but out-of-sequence messages back into the correct order?

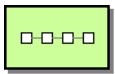


Composed Message Processor

How can you maintain the overall message flow when processing a message consisting of multiple elements, each of which may require different processing?

Scatter-Gather

How do you maintain the overall message flow when a message needs to be sent to multiple recipients, each of which may send a reply?



Routing Slip

How do we route a message consecutively through a series of processing steps when the sequence of steps is not known at design-time and may vary for each message?

Throttler

How can I throttle messages to ensure that a specific endpoint does not get overloaded, or we don't exceed an agreed SLA with some external service?

Sampling

How can I sample one message out of many in a given period to avoid downstream route does not get overloaded?

Delayer

How can I delay the sending of a message?

Load Balancer

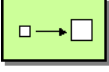
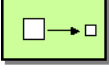
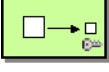
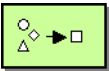
How can I balance load across a number of endpoints?

Multicast







How can I route a message to a number of endpoints at the same time?





Loop How can I repeat processing a message in a loop?

Message Transformation

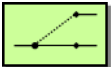

	Content Enricher	How do we communicate with another system if the message originator does not have all the required data items available?
	Content Filter	How do you simplify dealing with a large message, when you are interested only in a few data items?
	Claim Check	How can we reduce the data volume of message sent across the system without sacrificing information content?
	Normalizer	How do you process messages that are semantically equivalent, but arrive in a different format?
	Sort	How can I sort the body of a message?
	Validate	How can I validate a message?

Messaging Endpoints

	Messaging Mapper	How do you move data between domain objects and the messaging infrastructure while keeping the two independent of each other?
	Event Driven Consumer	How can an application automatically consume messages as they become available?
	Polling Consumer	How can an application consume a message when the application is ready?
	Competing Consumers	How can a messaging client process multiple messages concurrently?
	Message Dispatcher	How can multiple consumers on a single channel coordinate their message processing?
	Selective Consumer	How can a message consumer select which messages it wishes to receive?
	Durable Subscriber	How can a subscriber avoid missing messages while it's not listening for them?

	Idempotent Consumer	How can a message receiver deal with duplicate messages?
	Transactional Client	How can a client control its transactions with the messaging system?
	Messaging Gateway	How do you encapsulate access to the messaging system from the rest of the application?
	Service Activator	How can an application design a service to be invoked both via various messaging technologies and via non-messaging techniques?

System Management

	Detour	How can you route a message through intermediate steps to perform validation, testing or debugging functions?
	Wire Tap	How do you inspect messages that travel on a point-to-point channel?
	Log	How can I log processing a message?

For a full breakdown of each pattern see the Book Pattern Appendix

This document describes various recipes for working with Camel

- Bean Integration describes how to work with beans and Camel in a loosely coupled way so that your beans do not have to depend on any Camel APIs
 - Annotation Based Expression Language binds expressions to method parameters
 - Bean Binding defines which methods are invoked and how the Message is converted into the parameters of the method when it is invoked
 - Bean Injection for injecting Camel related resources into your POJOs
 - Parameter Binding Annotations for extracting various headers, properties or payloads from a Message
 - POJO Consuming for consuming and possibly routing messages from Camel
 - POJO Producing for producing camel messages from your POJOs
 - RecipientList Annotation for creating a Recipient List from a POJO method
 - Using Exchange Pattern Annotations describes how pattern annotations can be used to change the behaviour of method invocations
- Hiding Middleware describes how to avoid your business logic being coupled to any particular middleware APIs allowing you to easily switch from in JVM SEDA to JMS, ActiveMQ, Hibernate, JPA, JDBC, iBATIS or JavaSpace etc.
- Visualisation describes how to visualise your Enterprise Integration Patterns to help you understand your routing rules
- Business Activity Monitoring (BAM) for monitoring business processes across systems
- Extract Transform Load (ETL) to load data into systems or databases
- Testing for testing distributed and asynchronous systems using a messaging approach
 - Camel Test for creating test cases using a single Java class for all your configuration and routing
 - Spring Testing uses Spring Test together with either XML or Java Config to dependency inject your test classes
 - Guice uses Guice to dependency inject your test classes

- Templating is a great way to create service stubs to be able to test your system without some back end system.
- Database for working with databases
- Parallel Processing and Ordering on how using parallel processing and SEDA or JMS based load balancing can be achieved.
- Asynchronous Processing in Camel Routes.
- Implementing Virtual Topics on other JMS providers shows how to get the effect of Virtual Topics and avoid issues with JMS durable topics
- Camel Transport for CXF describes how to put the Camel context into the CXF transport layer.
- Fine Grained Control Over a Channel describes how to deliver a sequence of messages over a single channel and then stopping any more messages being sent over that channel. Typically used for sending data over a socket and then closing the socket.
- EventNotifier to log details about all sent Exchanges shows how to let Camels EventNotifier log all sent to endpoint events and how long time it took.
- Loading routes from XML files into an existing CamelContext.
- Using MDC logging with Camel
- Running Camel standalone and have it keep running shows how to keep Camel running when you run it standalone.
- Hazelcast Idempotent Repository Tutorial shows how to avoid to consume duplicated messages in a clustered environment.
- How to use Camel as a HTTP proxy between a client and server shows how to use Camel as a HTTP adapter/proxy between a client and HTTP service.

BEAN INTEGRATION

Camel supports the integration of beans and POJOs in a number of ways

Annotations

If a bean is defined in Spring XML or scanned using the Spring component scanning mechanism and a `<camelContext>` is used or a `CamelBeanPostProcessor` then we process a number of Camel annotations to do various things such as injecting resources or producing, consuming or routing messages.

- POJO Consuming to consume and possibly route messages from Camel
- POJO Producing to make it easy to produce camel messages from your POJOs

- `DynamicRouter` Annotation for creating a Dynamic Router from a POJO method
- `RecipientList` Annotation for creating a Recipient List from a POJO method
- `RoutingSlip` Annotation for creating a Routing Slip for a POJO method
- Bean Injection to inject Camel related resources into your POJOs
- Using Exchange Pattern Annotations describes how the pattern annotations can be used to change the behaviour of method invocations with Spring Remoting or POJO Producing

Bean Component

The Bean component allows one to invoke a particular method. Alternately the Bean component supports the creation of a proxy via `ProxyHelper` to a Java interface; which the implementation just sends a message containing a `BeanInvocation` to some Camel endpoint.

Spring Remoting

We support a Spring Remoting provider which uses Camel as the underlying transport mechanism. The nice thing about this approach is we can use any of the Camel transport Components to communicate between beans. It also means we can use Content Based Router and the other Enterprise Integration Patterns in between the beans; in particular we can use Message Translator to be able to convert what the on-the-wire messages look like in addition to adding various headers and so forth.

Annotation Based Expression Language

You can also use any of the Languages supported in Camel to bind expressions to method parameters when using Bean Integration. For example you can use any of these annotations:

Annotation	Description
<code>@Bean</code>	Inject a Bean expression
<code>@BeanShell</code>	Inject a BeanShell expression
<code>@Constant</code>	Inject a Constant expression
<code>@EL</code>	Inject an EL expression
<code>@Groovy</code>	Inject a Groovy expression
<code>@Header</code>	Inject a Header expression



Bean binding

Whenever Camel invokes a bean method via one of the above methods (Bean component, Spring Remoting or POJO Consuming) then the **Bean Binding** mechanism is used to figure out what method to use (if it is not explicit) and how to bind the Message to the parameters possibly using the Parameter Binding Annotations or using a method name option.

@JavaScript	Inject a JavaScript expression
@MVEL	Inject a Mvel expression
@OGNL	Inject an OGNL expression
@PHP	Inject a PHP expression
@Python	Inject a Python expression
@Ruby	Inject a Ruby expression
@Simple	Inject an Simple expression
@XPath	Inject an XPath expression
@XQuery	Inject an XQuery expression

Example:

```
public class Foo {  
    @MessageDriven(uri = "activemq:my.queue")  
    public void doSomething(@XPath("/foo/bar/text()") String correlationID, @Body  
String body) {  
        // process the inbound message here  
    }  
}
```

Advanced example using @Bean

And an example of using the the @Bean binding annotation, where you can use a Pojo where you can do whatever java code you like:

```

public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Bean("myCorrelationIdGenerator") String correlationID,
@Body String body) {
        // process the inbound message here
    }
}

```

And then we can have a spring bean with the id **myCorrelationIdGenerator** where we can compute the id.

```

public class MyIdGenerator {

    private UserManager userManager;

    public String generate(@Header(name = "user") String user, @Body String payload)
throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}

```

The Pojo MyIdGenerator has one public method that accepts two parameters. However we have also annotated this one with the `@Header` and `@Body` annotation to help Camel know what to bind here from the Message from the Exchange being processed.

Of course this could be simplified a lot if you for instance just have a simple id generator. But we wanted to demonstrate that you can use the Bean Binding annotations anywhere.

```

public class MySimpleIdGenerator {

    public static int generate() {
        // generate a unique id
        return 123;
    }
}

```

And finally we just need to remember to have our bean registered in the Spring Registry:

```

<bean id="myCorrelationIdGenerator" class="com.mycompany.MySimpleIdGenerator"/>

```


Example using Groovy

In this example we have an Exchange that has a User object stored in the in header. This User object has methods to get some user information. We want to use Groovy to inject an expression that extracts and concatenates the fullname of the user into the fullName parameter.

```
public void doSomething(@Groovy("${request.header['user'].firstName
$request.header['user'].familyName) String fullName, @Body String body) {
    // process the inbound message here
}
```

Groovy supports GStrings that is like a template where we can insert \$ placeholders that will be evaluated by Groovy.

BEAN BINDING

Bean Binding in Camel defines both which methods are invoked and also how the Message is converted into the parameters of the method when it is invoked.

Choosing the method to invoke

The binding of a Camel Message to a bean method call can occur in different ways, in the following order of importance:

- if the message contains the header **CamelBeanMethodName** then that method is invoked, converting the body to the type of the method's argument.
 - From **Camel 2.8** onwards you can qualify parameter types to select exactly which method to use among overloads with the same name (see below for more details).
 - From **Camel 2.9** onwards you can specify parameter values directly in the method option (see below for more details).
- you can explicitly specify the method name in the DSL or when using POJO Consuming or POJO Producing
- if the bean has a method marked with the `@Handler` annotation, then that method is selected
- if the bean can be converted to a Processor using the Type Converter mechanism, then this is used to process the message. The ActiveMQ component uses this mechanism to allow any JMS MessageListener to be invoked directly by Camel without having to write any integration glue code. You can use the same mechanism to integrate Camel into any other messaging/remoting frameworks.

- if the body of the message can be converted to a BeanInvocation (the default payload used by the ProxyHelper) component - then that is used to invoke the method and pass its arguments
- otherwise the type of the body is used to find a matching method; an error is thrown if a single method cannot be chosen unambiguously.
- you can also use Exchange as the parameter itself, but then the return type must be void.
- if the bean class is private (or package-private), interface methods will be preferred (from **Camel 2.9** onwards) since Camel can't invoke class methods on such beans

In cases where Camel cannot choose a method to invoke, an `AmbiguousMethodCallException` is thrown.

By default the return value is set on the outbound message body.

Parameter binding

When a method has been chosen for invocation, Camel will bind to the parameters of the method.

The following Camel-specific types are automatically bound:

- `org.apache.camel.Exchange`
- `org.apache.camel.Message`
- `org.apache.camel.CamelContext`
- `org.apache.camel.TypeConverter`
- `org.apache.camel.spi.Registry`
- `java.lang.Exception`

So, if you declare any of these types, they will be provided by Camel. **Note that Exception will bind to the caught exception of the Exchange** - so it's often usable if you employ a Pojo to handle, e.g., an `onException` route.

What is most interesting is that Camel will also try to bind the body of the Exchange to the first parameter of the method signature (albeit not of any of the types above). So if, for instance, we declare a parameter as `String` body, then Camel will bind the IN body to this type. Camel will also automatically convert to the type declared in the method signature.

Let's review some examples:

Below is a simple method with a body binding. Camel will bind the IN body to the body parameter and convert it to a `String`.

```
public String doSomething(String body)
```

In the following sample we got one of the automatically-bound types as well - for instance, a `Registry` that we can use to lookup beans.

```
public String doSomething(String body, Registry registry)
```

We can use Exchange as well:

```
public String doSomething(String body, Exchange exchange)
```

You can also have multiple types:

```
public String doSomething(String body, Exchange exchange, TypeConverter converter)
```

And imagine you use a Pojo to handle a given custom exception `InvalidOrderException` - we can then bind that as well:

```
public String badOrder(String body, InvalidOrderException invalid)
```

Notice that we can bind to it even if we use a sub type of `java.lang.Exception` as Camel still knows it's an exception and can bind the cause (if any exists).

So what about headers and other stuff? Well now it gets a bit tricky - so we can use annotations to help us, or specify the binding in the method name option.

See the following sections for more detail.

Binding Annotations

You can use the Parameter Binding Annotations to customize how parameter values are created from the Message

Examples

For example, a Bean such as:

```
public class Bar {  
  
    public String doSomething(String body) {  
        // process the in body and return whatever you want  
        return "Bye World";  
    }  
}
```

Or the Exchange example. Notice that the return type must be **void** when there is only a single parameter:

```
public class Bar {
    public void doSomething(Exchange exchange) {
        // process the exchange
        exchange.getIn().setBody("Bye World");
    }
}
```

@Handler

You can mark a method in your bean with the @Handler annotation to indicate that this method should be used for Bean Binding.

This has an advantage as you need not specify a method name in the Camel route, and therefore do not run into problems after renaming the method in an IDE that can't find all its references.

```
public class Bar {
    @Handler
    public String doSomething(String body) {
        // process the in body and return whatever you want
        return "Bye World";
    }
}
```

Parameter binding using method option

Available as of Camel 2.9

Camel uses the following rules to determine if it's a parameter value in the method option

- The value is either true or false which denotes a boolean value
- The value is a numeric value such as 123 or 7
- The value is a String enclosed with either single or double quotes
- The value is null which denotes a null value
- It can be evaluated using the Simple language, which means you can use, e.g., body, header.foo and other Simple tokens. Notice the tokens must be enclosed with \${ }.

Any other value is consider to be a type declaration instead - see the next section about specifying types for overloaded methods.

When invoking a Bean you can instruct Camel to invoke a specific method by providing the method name:

```
.bean(OrderService.class, "doSomething")
```

Here we tell Camel to invoke the `doSomething` method - Camel handles the parameters' binding. Now suppose the method has 2 parameters, and the 2nd parameter is a boolean where we want to pass in a true value:

```
public void doSomething(String payload, boolean highPriority) {  
    ...  
}
```

This is now possible in **Camel 2.9** onwards:

```
.bean(OrderService.class, "doSomething(*, true)")
```

In the example above, we defined the first parameter using the wild card symbol `*`, which tells Camel to bind this parameter to any type, and let Camel figure this out. The 2nd parameter has a fixed value of `true`. Instead of the wildcard symbol we can instruct Camel to use the message body as shown:

```
.bean(OrderService.class, "doSomething(${body}, true)")
```

The syntax of the parameters is using the Simple expression language so we have to use `${ }` placeholders in the body to refer to the message body.

If you want to pass in a null value, then you can explicit define this in the method option as shown below:

```
.to("bean:orderService?method=doSomething(null, true)")
```

Specifying `null` as a parameter value instructs Camel to force passing a null value.

Besides the message body, you can pass in the message headers as a `java.util.Map`:

```
.bean(OrderService.class, "doSomethingWithHeaders(${body}, ${headers})")
```

You can also pass in other fixed values besides booleans. For example, you can pass in a String and an integer:

```
.bean(MyBean.class, "echo('World', 5)")
```

In the example above, we invoke the `echo` method with two parameters. The first has the content 'World' (without quotes), and the 2nd has the value of 5. Camel will automatically convert these values to the parameters' types.

Having the power of the Simple language allows us to bind to message headers and other values such as:

```
.bean(OrderService.class, "doSomething(${body}, ${header.high})")
```

You can also use the OGNL support of the Simple expression language. Now suppose the message body is an object which has a method named `asXml`. To invoke the `asXml` method we can do as follows:

```
.bean(OrderService.class, "doSomething(${body.asXml}, ${header.high})")
```

Instead of using `.bean` as shown in the examples above, you may want to use `.to` instead as shown:

```
.to("bean:orderService?method=doSomething(${body.asXml}, ${header.high})")
```

Using type qualifiers to select among overloaded methods

Available as of Camel 2.8

If you have a Bean with overloaded methods, you can now specify parameter types in the method name so Camel can match the method you intend to use.

Given the following bean:

Listing 1. MyBean

```
public static final class MyBean {

    public String hello(String name) {
        return "Hello " + name;
    }

    public String hello(String name, @Header("country") String country) {
        return "Hello " + name + " you are from " + country;
    }

    public String times(String name, @Header("times") int times) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < times; i++) {
            sb.append(name);
        }
        return sb.toString();
    }

    public String times(byte[] data, @Header("times") int times) {
        String s = new String(data);
        StringBuilder sb = new StringBuilder();
    }
}
```

```

        for (int i = 0; i < times; i++) {
            sb.append(s);
            if (i < times - 1) {
                sb.append(",");
            }
        }
        return sb.toString();
    }

    public String times(String name, int times, char separator) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < times; i++) {
            sb.append(name);
            if (i < times - 1) {
                sb.append(separator);
            }
        }
        return sb.toString();
    }
}
}

```

Then the MyBean has 2 overloaded methods with the names hello and times. So if we want to use the method which has 2 parameters we can do as follows in the Camel route:

Listing 1. Invoke 2 parameter method

```

from("direct:start")
    .bean(MyBean.class, "hello(String,String)")
    .to("mock:result");

```

We can also use a * as wildcard so we can just say we want to execute the method with 2 parameters we do

Listing 1. Invoke 2 parameter method using wildcard

```

from("direct:start")
    .bean(MyBean.class, "hello(*,*)")
    .to("mock:result");

```

By default Camel will match the type name using the simple name, e.g. any leading package name will be disregarded. However if you want to match using the FQN, then specify the FQN type and Camel will leverage that. So if you have a com.foo.MyOrder and you want to match against the FQN, and **not** the simple name "MyOrder", then follow this example:

```

.bean(OrderService.class, "doSomething(com.foo.MyOrder)")

```

i Camel currently only supports either specifying parameter binding or type per parameter in the method name option. You **cannot** specify both at the same time, such as

```
doSomething(com.foo.MyOrder ${body}, boolean ${header.high})
```

This may change in the future.

Bean Injection

We support the injection of various resources using `@EndpointInject`. This can be used to inject

- Endpoint instances which can be used for testing when used with Mock endpoints; see the Spring Testing for an example.
- `ProducerTemplate` instances for POJO Producing
- client side proxies for POJO Producing which is a simple approach to Spring Remoting

Parameter Binding Annotations

Annotations can be used to define an Expression or to extract various headers, properties or payloads from a Message when invoking a bean method (see Bean Integration for more detail of how to invoke bean methods) together with being useful to help disambiguate which method to invoke.

If no annotations are used then Camel assumes that a single parameter is the body of the message. Camel will then use the Type Converter mechanism to convert from the expression value to the actual type of the parameter.

The core annotations are as follows

Annotation	Meaning	Parameter
<code>@Body</code>	To bind to an inbound message body	Â
<code>@ExchangeException</code>	To bind to an Exception set on the exchange (Camel 2.0)	Â
<code>@Header</code>	To bind to an inbound message header	String name of the header
<code>@Headers</code>	To bind to the Map of the inbound message headers	Â



camel-core

The annotations below are all part of **camel-core** and thus does not require **camel-spring** or Spring. These annotations can be used with the Bean component or when invoking beans in the DSL

@OutHeaders	To bind to the Map of the outbound message headers	Â
@Property	To bind to a named property on the exchange	String name of the property
@Properties	To bind to the property map on the exchange	Â
@Handler	Camel 2.0: Not part as a type parameter but stated in this table anyway to spread the good word that we have this annotation in Camel now. See more at Bean Binding.	Â

The follow annotations @Headers, @OutHeaders and @Properties binds to the backing java.util.Map so you can alter the content of these maps directly, for instance using the put method to add a new entry. See the OrderService class at Exception Clause for such an example.

Since **Camel 2.0**, you can use @Header("myHeader") and @Property("myProperty") instead of @Header(name="myHeader") and @Property(name="myProperty") as **Camel 1.x** does.

Example

In this example below we have a @Consume consumer (like message driven) that consumes JMS messages from the activemq queue. We use the @Header and @Body parameter binding annotations to bind from the JMSMessage to the method parameters.

```
public class Foo {
    @Consume(uri = "activemq:my.queue")
    public void doSomething(@Header("JMSCorrelationID") String correlationID, @Body
String body) {
```

```

        // process the inbound message here
    }
}

```

In the above Camel will extract the value of `Message.getJMSCorrelationID()`, then using the Type Converter to adapt the value to the type of the parameter if required - it will inject the parameter value for the **correlationID** parameter. Then the payload of the message will be converted to a `String` and injected into the **body** parameter.

You don't need to use the `@Consume` annotation; as you could use the Camel DSL to route to the beans method

Using the DSL to invoke the bean method

Here is another example which does not use POJO Consuming annotations but instead uses the DSL to route messages to the bean method

```

public class Foo {
    public void doSomething(@Header("JMSCorrelationID") String correlationID, @Body
String body) {
        // process the inbound message here
    }
}

```

The routing DSL then looks like this

```

from("activemq:someQueue").
to("bean:myBean");

```

Here **myBean** would be looked up in the Registry (such as JNDI or the Spring Application Context), then the body of the message would be used to try figure out what method to call.

If you want to be explicit you can use

```

from("activemq:someQueue").
to("bean:myBean?methodName=doSomething");

```

And here we have a nifty example for you to show some great power in Camel. You can mix and match the annotations with the normal parameters, so we can have this example with annotations and the Exchange also:

```

public void doSomething(@Header("user") String user, @Body String body, Exchange
exchange) {
    exchange.getIn().setBody(body + "MyBean");
}

```

Annotation Based Expression Language

You can also use any of the Languages supported in Camel to bind expressions to method parameters when using Bean Integration. For example you can use any of these annotations:

Annotation	Description
@Bean	Inject a Bean expression
@BeanShell	Inject a BeanShell expression
@Constant	Inject a Constant expression
@EL	Inject an EL expression
@Groovy	Inject a Groovy expression
@Header	Inject a Header expression
@JavaScript	Inject a JavaScript expression
@MVEL	Inject a Mvel expression
@OGNL	Inject an OGNL expression
@PHP	Inject a PHP expression
@Python	Inject a Python expression
@Ruby	Inject a Ruby expression
@Simple	Inject an Simple expression
@XPath	Inject an XPath expression
@XQuery	Inject an XQuery expression

Example:

```

public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@XPath("/foo/bar/text()") String correlationID, @Body

```

```
String body) {
    // process the inbound message here
}
}
```

Advanced example using @Bean

And an example of using the the @Bean binding annotation, where you can use a Pojo where you can do whatever java code you like:

```
public class Foo {

    @MessageDriven(uri = "activemq:my.queue")
    public void doSomething(@Bean("myCorrelationIdGenerator") String correlationID,
    @Body String body) {
        // process the inbound message here
    }
}
```

And then we can have a spring bean with the id **myCorrelationIdGenerator** where we can compute the id.

```
public class MyIdGenerator {

    private UserManager userManager;

    public String generate(@Header(name = "user") String user, @Body String payload)
    throws Exception {
        User user = userManager.lookupUser(user);
        String userId = user.getPrimaryId();
        String id = userId + generateHashCodeForPayload(payload);
        return id;
    }
}
```

The Pojo MyIdGenerator has one public method that accepts two parameters. However we have also annotated this one with the @Header and @Body annotation to help Camel know what to bind here from the Message from the Exchange being processed.

Of course this could be simplified a lot if you for instance just have a simple id generator. But we wanted to demonstrate that you can use the Bean Binding annotations anywhere.

```
public class MySimpleIdGenerator {
```

```

public static int generate() {
    // generate a unique id
    return 123;
}
}

```

And finally we just need to remember to have our bean registered in the Spring Registry:

```

<bean id="myCorrelationIdGenerator" class="com.mycompany.MySimpleIdGenerator"/>

```

Example using Groovy

In this example we have an Exchange that has a User object stored in the in header. This User object has methods to get some user information. We want to use Groovy to inject an expression that extracts and concatenates the fullname of the user into the fullName parameter.

```

public void doSomething(@Groovy("${request.header['user'].firstName
$request.header['user'].familyName) String fullName, @Body String body) {
    // process the inbound message here
}

```

Groovy supports GStrings that is like a template where we can insert \$ placeholders that will be evaluated by Groovy.

@MessageDriven or @Consume

To consume a message you use either the @MessageDriven annotation or from 1.5.0 the @Consume annotation to mark a particular method of a bean as being a consumer method. The uri of the annotation defines the Camel Endpoint to consume from.

e.g. lets invoke the onCheese() method with the String body of the inbound JMS message from ActiveMQ on the cheese queue; this will use the Type Converter to convert the JMS ObjectMessage or BytesMessage to a String - or just use a TextMessage from JMS

```

public class Foo {

    @Consume(uri="activemq:cheese")
    public void onCheese(String name) {
        ...
    }
}

```



@MessageDriven is @deprecated

@MessageDriven is deprecated in Camel 1.x. You should use @Consume instead. Its removed in Camel 2.0.

```
}  
}
```

The Bean Binding is then used to convert the inbound Message to the parameter list used to invoke the method .

What this does is basically create a route that looks kinda like this

```
from(uri).bean(theBean, "methodName");
```

Using context option to apply only a certain CamelContext

Available as of Camel 2.0

See the warning above.

You can use the context option to specify which CamelContext the consumer should only apply for. For example:

```
@Consume(uri="activemq:cheese", context="camel-1")  
public void onCheese(String name) {
```

The consumer above will only be created for the CamelContext that have the context id = camel-1. You set this id in the XML tag:

```
<camelContext id="camel-1" ...>
```

Using an explicit route

If you want to invoke a bean method from many different endpoints or within different complex routes in different circumstances you can just use the normal routing DSL or the Spring XML configuration file.

For example

```
from(uri).beanRef("myBean", "methodName");
```



When using more than one CamelContext

When you use more than 1 CamelContext you might end up with each of them creating a POJO Consuming.

In Camel 2.0 there is a new option on **@Consume** that allows you to specify which CamelContext id/name you want it to apply for.

which will then look up in the Registry and find the bean and invoke the given bean name. (You can omit the method name and have Camel figure out the right method based on the method annotations and body type).

Use the Bean endpoint

You can always use the bean endpoint

```
from(uri).to("bean:myBean?method=methodName");
```

Which approach to use?

Using the @MessageDriven/@Consume annotations are simpler when you are creating a simple route with a single well defined input URI.

However if you require more complex routes or the same bean method needs to be invoked from many places then please use the routing DSL as shown above.

There are two different ways to send messages to any Camel Endpoint from a POJO

@EndpointInject

To allow sending of messages from POJOs you can use @EndpointInject() annotation. This will inject either a ProducerTemplate or CamelTemplate so that the bean can send message exchanges.

e.g. lets send a message to the **foo.bar** queue in ActiveMQ at some point

```
public class Foo {  
    @EndpointInject(uri="activemq:foo.bar")  
    ProducerTemplate producer;  
}
```

```

public void doSomething() {
    if (whatever) {
        producer.sendBody("<hello>world!</hello>");
    }
}
}

```

The downside of this is that your code is now dependent on a Camel API, the `ProducerTemplate`. The next section describes how to remove this

Hiding the Camel APIs from your code using `@Produce`

We recommend Hiding Middleware APIs from your application code so the next option might be more suitable.

You can add the `@Produce` annotation to an injection point (a field or property setter) using a `ProducerTemplate` **or** using some interface you use in your business logic. e.g.

```

public interface MyListener {
    String sayHello(String name);
}

public class MyBean {
    @Produce(uri = "activemq:foo")
    protected MyListener producer;

    public void doSomething() {
        // lets send a message
        String response = producer.sayHello("James");
    }
}

```

Here Camel will automatically inject a smart client side proxy at the `@Produce` annotation - an instance of the `MyListener` instance. When we invoke methods on this interface the method call is turned into an object and using the Camel Spring Remoting mechanism it is sent to the endpoint - in this case the ActiveMQ endpoint to queue **foo**; then the caller blocks for a response.

If you want to make asynchronous message sends then use an `@InOnly` annotation on the injection point.

`@RECIPIENTLIST` ANNOTATION

As of 1.5.0 we now support the use of `@RecipientList` on a bean method to easily create a dynamic Recipient List using a Java method.

Simple Example using @Consume and @RecipientList

```
package com.acme.foo;

public class RouterBean {

    @Consume(uri = "activemq:foo")
    @RecipientList
    public String[] route(String body) {
        return new String[]{"activemq:bar", "activemq:whatnot"};
    }
}
```

For example if the above bean is configured in Spring when using a **<camelContext>** element as follows

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

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

        <bean id="myRecipientList" class="com.acme.foo.RouterBean"/>

    </beans>
```

then a route will be created consuming from the **foo** queue on the ActiveMQ component which when a message is received the message will be forwarded to the endpoints defined by the result of this method call - namely the **bar** and **whatnot** queues.

How it works

The return value of the @RecipientList method is converted to either a java.util.Collection / java.util.Iterator or array of objects where each element is converted to an Endpoint or a String, or if you are only going to route to a single endpoint then just return either an Endpoint object or an object that can be converted to a String. So the following methods are all valid

```
@RecipientList
public String[] route(String body) { ... }
```

```

@RecipientList
public List<String> route(String body) { ... }

@RecipientList
public Endpoint route(String body) { ... }

@RecipientList
public Endpoint[] route(String body) { ... }

@RecipientList
public Collection<Endpoint> route(String body) { ... }

@RecipientList
public URI route(String body) { ... }

@RecipientList
public URI[] route(String body) { ... }

```

Then for each endpoint or URI the message is forwarded a separate copy to that endpoint.

You can then use whatever Java code you wish to figure out what endpoints to route to; for example you can use the Bean Binding annotations to inject parts of the message body or headers or use Expression values on the message.

More Complex Example Using DSL

In this example we will use more complex Bean Binding, plus we will use a separate route to invoke the Recipient List

```

public class RouterBean2 {

    @RecipientList
    public String route(@Header("customerID") String custID String body) {
        if (custID == null) return null;
        return "activemq:Customers.Orders." + custID;
    }
}

public class MyRouteBuilder extends RouteBuilder {
    protected void configure() {
        from("activemq:Orders.Incoming").recipientList(bean("myRouterBean", "route"));
    }
}

```

Notice how we are injecting some headers or expressions and using them to determine the recipients using Recipient List EIP. See the Bean Integration for more details.

USING EXCHANGE PATTERN ANNOTATIONS

When working with POJO Producing or Spring Remoting you invoke methods which typically by default are InOut for Request Reply. That is there is an In message and an Out for the result. Typically invoking this operation will be synchronous, the caller will block until the server returns a result.

Camel has flexible Exchange Pattern support - so you can also support the Event Message pattern to use InOnly for asynchronous or one way operations. These are often called 'fire and forget' like sending a JMS message but not waiting for any response.

From 1.5 onwards Camel supports annotations for specifying the message exchange pattern on regular Java methods, classes or interfaces.

Specifying InOnly methods

Typically the default InOut is what most folks want but you can customize to use InOnly using an annotation.

```
public interface Foo {
    Object someInOutMethod(String input);
    String anotherInOutMethod(Cheese input);

    @InOnly
    void someInOnlyMethod(Document input);
}
```

The above code shows three methods on an interface; the first two use the default InOut mechanism but the **someInOnlyMethod** uses the InOnly annotation to specify it as being a oneway method call.

Class level annotations

You can also use class level annotations to default all methods in an interface to some pattern such as

```
@InOnly
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);
}
```

Annotations will also be detected on base classes or interfaces. So for example if you created a client side proxy for

```
public class MyFoo implements Foo {
    ...
}
```

Then the methods inherited from Foo would be InOnly.

Overloading a class level annotation

You can overload a class level annotation on specific methods. A common use case for this is if you have a class or interface with many InOnly methods but you want to just annotate one or two methods as InOut

```
@InOnly
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);

    @InOut
    String someInOutMethod(String input);
}
```

In the above Foo interface the **someInOutMethod** will be InOut

Using your own annotations

You might want to create your own annotations to represent a group of different bits of metadata; such as combining synchrony, concurrency and transaction behaviour.

So you could annotate your annotation with the @Pattern annotation to default the exchange pattern you wish to use.

For example lets say we want to create our own annotation called @MyAsyncService

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})

// lets add the message exchange pattern to it
@Pattern(ExchangePattern.InOnly)

// lets add some other annotations - maybe transaction behaviour?

public @interface MyAsyncService {
}
```

Now we can use this annotation and Camel will figure out the correct exchange pattern...

```
public interface Foo {
    void someInOnlyMethod(Document input);
    void anotherInOnlyMethod(String input);

    @MyAsyncService
    String someInOutMethod(String input);
}
```

When writing software these days, its important to try and decouple as much middleware code from your business logic as possible.

This provides a number of benefits...

- you can choose the right middleware solution for your deployment and switch at any time
- you don't have to spend a large amount of time learning the specifics of any particular technology, whether its JMS or JavaSpace or Hibernate or JPA or iBATIS whatever

For example if you want to implement some kind of message passing, remoting, reliable load balancing or asynchronous processing in your application we recommend you use Camel annotations to bind your services and business logic to Camel Components which means you can then easily switch between things like

- in JVM messaging with SEDA
- using JMS via ActiveMQ or other JMS providers for reliable load balancing, grid or publish and subscribe
- for low volume, but easier administration since you're probably already using a database you could use
 - Hibernate or JPA to use an entity bean / table as a queue
 - iBATIS to work with SQL
 - JDBC for raw SQL access
- use JavaSpace

How to decouple from middleware APIs

The best approach when using remoting is to use Spring Remoting which can then use any messaging or remoting technology under the covers. When using Camel's implementation you can then use any of the Camel Components along with any of the Enterprise Integration Patterns.

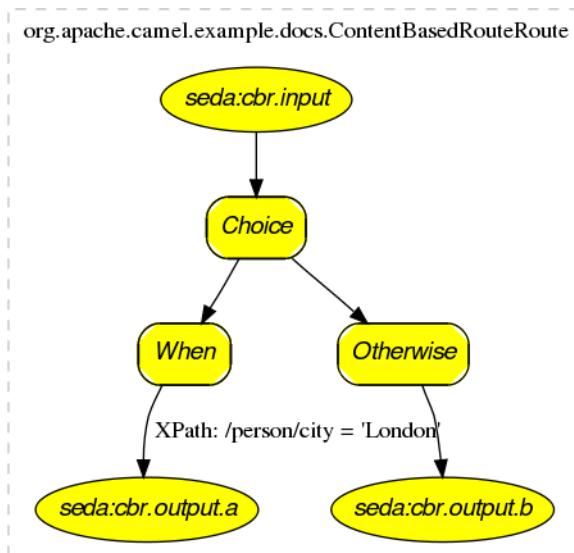
Another approach is to bind Java beans to Camel endpoints via the Bean Integration. For example using POJO Consuming and POJO Producing you can avoid using any Camel APIs to decouple your code both from middleware

APIs *and* Camel APIs! 😊

VISUALISATION

Camel supports the visualisation of your Enterprise Integration Patterns using the GraphViz DOT files which can either be rendered directly via a suitable GraphViz tool or turned into HTML, PNG or SVG files via the Camel Maven Plugin.

Here is a typical example of the kind of thing we can generate



If you click on the actual generated html you will see that you can navigate from an EIP node to its pattern page, along with getting hover-over tool tips ec.

How to generate

See Camel Dot Maven Goal or the other maven goals Camel Maven Plugin

For OS X users

If you are using OS X then you can open the DOT file using graphviz which will then automatically re-render if it changes, so you end up with a real time graphical representation of the topic and queue hierarchies!

Also if you want to edit the layout a little before adding it to a wiki to distribute to your team, open the DOT file with OmniGraffle then just edit away 😊

BUSINESS ACTIVITY MONITORING

The **Camel BAM** module provides a Business Activity Monitoring (BAM) framework for testing business processes across multiple message exchanges on different Endpoint instances.

Consider, for example, a simple system in which you submit Purchase Orders into system A and then receive Invoices from system B. You might want to test that, for a given Purchase Order, you receive a matching Invoice from system B within a specific time period.

How Camel BAM Works

Camel BAM uses a Correlation Identifier on an input message to determine the *Process Instance* to which it belongs. The process instance is an entity bean which can maintain state for each *Activity* (where an activity typically maps to a single endpoint - such as the submission of Purchase Orders or the receipt of Invoices).

You can then add rules to be triggered when a message is received on any activity - such as to set time expectations or perform real time reconciliation of values across activities.

Simple Example

The following example shows how to perform some time based rules on a simple business process of 2 activities - A and B - which correspond with Purchase Orders and Invoices in the example above. If you would like to experiment with this scenario, you may edit this Test Case, which defines the activities and rules, and then tests that they work.

```
return new ProcessBuilder(jpaTemplate, transactionTemplate) {
    public void configure() throws Exception {

        // let's define some activities, correlating on an XPath on the message bodies
        ActivityBuilder a = activity("seda:a").name("a")
            .correlate(xpath("/hello/@id"));

        ActivityBuilder b = activity("seda:b").name("b")
            .correlate(xpath("/hello/@id"));

        // now let's add some rules
        b.starts().after(a.completes())
            .expectWithin(seconds(1))
            .errorIfOver(seconds(errorTimeout)).to("mock:overdue");
    }
};
```

As you can see in the above example, we first define two activities, and then rules to specify when we expect them to complete for a process instance and when an error condition should be raised. The ProcessBuilder is a RouteBuilder and can be added to any CamelContext.

Complete Example

For a complete example please see the BAM Example, which is part of the standard Camel Examples

Use Cases

In the world of finance, a common requirement is tracking trades. Often a trader will submit a Front Office Trade which then flows through the Middle Office and Back Office through various systems to settle the trade so that money is exchanged. You may wish to test that the front and back office trades match up within a certain time period; if they don't match or a back office trade does not arrive within a required amount of time, you might signal an alarm.

EXTRACT TRANSFORM LOAD (ETL)

The ETL (Extract, Transform, Load) is a mechanism for loading data into systems or databases using some kind of Data Format from a variety of sources; often files then using Pipes and Filters, Message Translator and possible other Enterprise Integration Patterns.

So you could query data from various Camel Components such as File, HTTP or JPA, perform multiple patterns such as Splitter or Message Translator then send the messages to some other Component.

To show how this all fits together, try the ETL Example

MOCK COMPONENT

Testing of distributed and asynchronous processing is notoriously difficult. The Mock, Test and DataSet endpoints work great with the Camel Testing Framework to simplify your unit and integration testing using Enterprise Integration Patterns and Camel's large range of Components together with the powerful Bean Integration.

The Mock component provides a powerful declarative testing mechanism, which is similar to jMock in that it allows declarative expectations to be created on any Mock endpoint before a test begins. Then the test is run, which typically fires messages to one or more endpoints, and finally the

expectations can be asserted in a test case to ensure the system worked as expected.

This allows you to test various things like:

- The correct number of messages are received on each endpoint,
- The correct payloads are received, in the right order,
- Messages arrive on an endpoint in order, using some Expression to create an order testing function,
- Messages arrive match some kind of Predicate such as that specific headers have certain values, or that parts of the messages match some predicate, such as by evaluating an XPath or XQuery Expression.

Note that there is also the Test endpoint which is a Mock endpoint, but which uses a second endpoint to provide the list of expected message bodies and automatically sets up the Mock endpoint assertions. In other words, it's a Mock endpoint that automatically sets up its assertions from some sample messages in a File or database, for example.

URI format

```
mock:someName[?options]
```

Where **someName** can be any string that uniquely identifies the endpoint.

You can append query options to the URI in the following format, `?option=value&option=value&...`

Options

Option	Default	Description
reportGroup	null	A size to use a throughput logger for reporting

Simple Example

Here's a simple example of Mock endpoint in use. First, the endpoint is resolved on the context. Then we set an expectation, and then, after the test has run, we assert that our expectations have been met.

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);  
resultEndpoint.expectedMessageCount(2);  
  
// send some messages  
...
```



Mock endpoints keep received Exchanges in memory indefinitely

Remember that Mock is designed for testing. When you add Mock endpoints to a route, each Exchange sent to the endpoint will be stored (to allow for later validation) in memory until explicitly reset or the JVM is restarted. If you are sending high volume and/or large messages, this may cause excessive memory use. If your goal is to test deployable routes inline, consider using `NotifyBuilder` or `AdviceWith` in your tests instead of adding Mock endpoints to routes directly.

```
// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

You typically always call the `assertIsSatisfied()` method to test that the expectations were met after running a test.

Camel will by default wait 10 seconds when the `assertIsSatisfied()` is invoked. This can be configured by setting the `setResultWaitTime(millis)` method.

When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method.

Using `assertPeriod`

Available as of Camel 2.7

When the assertion is satisfied then Camel will stop waiting and continue from the `assertIsSatisfied` method. That means if a new message arrives on the mock endpoint, just a bit later, that arrival will not affect the outcome of the assertion. Suppose you do want to test that no new messages arrives after a period thereafter, then you can do that by setting the `setAssertPeriod` method, for example:

```
MockEndpoint resultEndpoint = context.resolveEndpoint("mock:foo", MockEndpoint.class);
resultEndpoint.setAssertPeriod(5000);
resultEndpoint.expectedMessageCount(2);
```

```
// send some messages
...

// now lets assert that the mock:foo endpoint received 2 messages
resultEndpoint.assertIsSatisfied();
```

Setting expectations

You can see from the javadoc of `MockEndpoint` the various helper methods you can use to set expectations. The main methods are as follows:

Method	Description
<code>expectedMessageCount(int)</code>	To define the expected message count on the endpoint.
<code>expectedMinimumMessageCount(int)</code>	To define the minimum number of expected messages on the endpoint.
<code>expectedBodiesReceived(...)</code>	To define the expected bodies that should be received (in order).
<code>expectedHeaderReceived(...)</code>	To define the expected header that should be received
<code>expectsAscending(Expression)</code>	To add an expectation that messages are received in order, using the given Expression to compare messages.
<code>expectsDescending(Expression)</code>	To add an expectation that messages are received in order, using the given Expression to compare messages.
<code>expectsNoDuplicates(Expression)</code>	To add an expectation that no duplicate messages are received; using an Expression to calculate a unique identifier for each message. This could be something like the <code>JMSMessageID</code> if using JMS, or some unique reference number within the message.

Here's another example:

```
resultEndpoint.expectedBodiesReceived("firstMessageBody", "secondMessageBody",
"thirdMessageBody");
```

Adding expectations to specific messages

In addition, you can use the `message(int messageIndex)` method to add assertions about a specific message that is received.

For example, to add expectations of the headers or body of the first message (using zero-based indexing like `java.util.List`), you can use the following code:

```
resultEndpoint.message(0).header("foo").isEqualTo("bar");
```

There are some examples of the `Mock` endpoint in use in the `camel-core` processor tests.

Mocking existing endpoints

Available as of Camel 2.7

Camel now allows you to automatic mock existing endpoints in your Camel routes.

Suppose you have the given route below:

Listing 1. Route

```
@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

            from("direct:foo").transform(constant("Bye World"));
        }
    };
}
```

You can then use the `adviceWith` feature in Camel to mock all the endpoints in a given route from your unit test, as shown below:

Listing 1. `adviceWith` mocking all endpoints

```
public void testAdvisedMockEndpoints() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new
AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock all endpoints
            mockEndpoints();
        }
    });

    getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello World");
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // all the endpoints was mocked
    assertNotNull(context.hasEndpoint("mock:direct:start"));
    assertNotNull(context.hasEndpoint("mock:direct:foo"));
    assertNotNull(context.hasEndpoint("mock:log:foo"));
}
```

How it works

Important: The endpoints are still in action, what happens is that a Mock endpoint is injected and receives the message first, it then delegate the message to the target endpoint. You can view this as a kind of intercept and delegate or endpoint listener.

Notice that the mock endpoints is given the uri `mock:<endpoint>`, for example `mock:direct:foo`. Camel logs at INFO level the endpoints being mocked:

```
INFO  Advised endpoint [direct://foo] with mock endpoint [mock:direct:foo]
```

Its also possible to only mock certain endpoints using a pattern. For example to mock all `log` endpoints you do as shown:

Listing 1. adviceWith mocking only log endpoints using a pattern

```
public void testAdvisedMockEndpointsWithPattern() throws Exception {
    // advice the first route using the inlined AdviceWith route builder
    // which has extended capabilities than the regular route builder
    context.getRouteDefinitions().get(0).adviceWith(context, new
AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            // mock only log endpoints
            mockEndpoints("log*");
        }
    });

    // now we can refer to log:foo as a mock and set our expectations
    getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");

    getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

    template.sendBody("direct:start", "Hello World");

    assertMockEndpointsSatisfied();

    // additional test to ensure correct endpoints in registry
    assertNotNull(context.hasEndpoint("direct:start"));
    assertNotNull(context.hasEndpoint("direct:foo"));
    assertNotNull(context.hasEndpoint("log:foo"));
    assertNotNull(context.hasEndpoint("mock:result"));
    // only the log:foo endpoint was mocked
    assertNotNull(context.hasEndpoint("mock:log:foo"));
    assertNull(context.hasEndpoint("mock:direct:start"));
    assertNull(context.hasEndpoint("mock:direct:foo"));
}
```



Mocked endpoints are without parameters

Endpoints which are mocked will have their parameters stripped off. For example the endpoint "log:foo?showAll=true" will be mocked to the following endpoint "mock:log:foo". Notice the parameters has been removed.

The pattern supported can be a wildcard or a regular expression. See more details about this at Intercept as its the same matching function used by Camel.

Mocking existing endpoints using the camel-test component

Instead of using the `adviseWith` to instruct Camel to mock endpoints, you can easily enable this behavior when using the `camel-test` Test Kit. The same route can be tested as follows. Notice that we return "*" from the `isMockEndpoints` method, which tells Camel to mock all endpoints. If you only want to mock all log endpoints you can return "log*" instead.

Listing 1. isMockEndpoints using camel-test kit

```
public class IsMockEndpointsJUnit4Test extends CamelTestSupport {

    @Override
    public String isMockEndpoints() {
        // override this method and return the pattern for which endpoints to mock.
        // use * to indicate all
        return "*";
    }

    @Test
    public void testMockAllEndpoints() throws Exception {
        // notice we have automatic mocked all endpoints and the name of the
        endpoints is "mock:uri"
        getMockEndpoint("mock:direct:start").expectedBodiesReceived("Hello World");
        getMockEndpoint("mock:direct:foo").expectedBodiesReceived("Hello World");
        getMockEndpoint("mock:log:foo").expectedBodiesReceived("Bye World");
        getMockEndpoint("mock:result").expectedBodiesReceived("Bye World");

        template.sendBody("direct:start", "Hello World");

        assertMockEndpointsSatisfied();

        // additional test to ensure correct endpoints in registry
        assertNotNull(context.hasEndpoint("direct:start"));
        assertNotNull(context.hasEndpoint("direct:foo"));
    }
}
```



Mind that mocking endpoints causes the messages to be copied when they arrive on the mock. That means Camel will use more memory. This may not be suitable when you send in a lot of messages.

```
assertNotNull(context.hasEndpoint("log:foo"));
assertNotNull(context.hasEndpoint("mock:result"));
// all the endpoints was mocked
assertNotNull(context.hasEndpoint("mock:direct:start"));
assertNotNull(context.hasEndpoint("mock:direct:foo"));
assertNotNull(context.hasEndpoint("mock:log:foo"));
}

@Override
protected RouteBuilder createRouteBuilder() throws Exception {
    return new RouteBuilder() {
        @Override
        public void configure() throws Exception {
            from("direct:start").to("direct:foo").to("log:foo").to("mock:result");

            from("direct:foo").transform(constant("Bye World"));
        }
    };
}
```

Mocking existing endpoints with XML DSL

If you do not use the `camel-test` component for unit testing (as shown above) you can use a different approach when using XML files for routes. The solution is to create a new XML file used by the unit test and then include the intended XML file which has the route you want to test.

Suppose we have the route in the `camel-route.xml` file:

Listing 1. camel-route.xml

```
<!-- this camel route is in the camel-route.xml file -->
<camelContext xmlns="http://camel.apache.org/schema/spring">

    <route>
        <from uri="direct:start"/>
        <to uri="direct:foo"/>
        <to uri="log:foo"/>
        <to uri="mock:result"/>
    </route>

```

```

<route>
  <from uri="direct:foo"/>
  <transform>
    <constant>Bye World</constant>
  </transform>
</route>

</camelContext>

```

Then we create a new XML file as follows, where we include the `camel-route.xml` file and define a spring bean with the class `org.apache.camel.impl.InterceptSendToMockEndpointStrategy` which tells Camel to mock all endpoints:

Listing 1. test-camel-route.xml

```

<!-- the Camel route is defined in another XML file -->
<import resource="camel-route.xml"/>

<!-- bean which enables mocking all endpoints -->
<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy"/>

```

Then in your unit test you load the new XML file (`test-camel-route.xml`) instead of `camel-route.xml`.

To only mock all Log endpoints you can define the pattern in the constructor for the bean:

```

<bean id="mockAllEndpoints"
class="org.apache.camel.impl.InterceptSendToMockEndpointStrategy">
  <constructor-arg index="0" value="log*"/>
</bean>

```

Testing with arrival times

Available as of Camel 2.7

The Mock endpoint stores the arrival time of the message as a property on the Exchange.

```

Date time = exchange.getProperty(Exchange.RECEIVED_TIMESTAMP, Date.class);

```

You can use this information to know when the message arrived on the mock. But it also provides foundation to know the time interval between the previous and next message arrived on the mock. You can use this to set expectations using the `arrives` DSL on the Mock endpoint.

For example to say that the first message should arrive between 0-2 seconds before the next you can do:

```
mock.message(0).arrives().noLaterThan(2).seconds().beforeNext();
```

You can also define this as that 2nd message (0 index based) should arrive no later than 0-2 seconds after the previous:

```
mock.message(1).arrives().noLaterThan(2).seconds().afterPrevious();
```

You can also use between to set a lower bound. For example suppose that it should be between 1-4 seconds:

```
mock.message(1).arrives().between(1, 4).seconds().afterPrevious();
```

You can also set the expectation on all messages, for example to say that the gap between them should be at most 1 second:

```
mock.allMessages().arrives().noLaterThan(1).seconds().beforeNext();
```

See Also

- [Configuring Camel](#)
- [Component](#)
- [Endpoint](#)
- [Getting Started](#)
- [Spring Testing](#)
- [Testing](#)

TESTING

Testing is a crucial activity in any piece of software development or integration. Typically Camel Riders use various different technologies wired together in a variety of patterns with different expression languages together with different forms of Bean Integration and Dependency Injection so its very easy for things to go wrong! 😊. Testing is the crucial weapon to ensure that things work as you would expect.

Camel is a Java library so you can easily wire up tests in whatever unit testing framework you use (JUnit 3.x, 4.x or TestNG). However the Camel project has tried to make the testing of Camel as easy and powerful as possible so we have introduced the following features.



time units

In the example above we use seconds as the time unit, but Camel offers milliseconds, and minutes as well.

Testing mechanisms

The following mechanisms are supported

Name	Component	Description
Camel Test	camel-test	Is a standalone Java library letting you easily create Camel test cases using a single Java class for all your configuration and routing without using Spring or Guice for Dependency Injection which does not require an in depth knowledge of Spring+SpringTest or Guice
Spring Testing	camel-test-spring	Uses Spring Test together with either XML or Java Config to dependency inject your test classes. Notice camel-test-spring is a new component in Camel 2.10 onwards. For older Camel release use camel-test which has built-in Spring Testing.
Blueprint Testing	camel-test-blueprint	Camel 2.10: Provides the ability to do unit testing on blueprint configurations
Guice	camel-guice	Uses Guice to dependency inject your test classes

In all approaches the test classes look pretty much the same in that they all reuse the Camel binding and injection annotations.

Camel Test Example

Here is the Camel Test example.

```
public class FilterTest extends CamelTestSupport {  
  
    @EndpointInject(uri = "mock:result")  
    protected MockEndpoint resultEndpoint;  
  
    @Produce(uri = "direct:start")  
    protected ProducerTemplate template;  
  
}
```

```

@Test
public void testSendMatchingMessage() throws Exception {
    String expectedBody = "<matched/>";

    resultEndpoint.expectedBodiesReceived(expectedBody);

    template.sendBodyAndHeader(expectedBody, "foo", "bar");

    resultEndpoint.assertIsSatisfied();
}

@Test
public void testSendNotMatchingMessage() throws Exception {
    resultEndpoint.expectedMessageCount(0);

    template.sendBodyAndHeader("<notMatched/>", "foo", "notMatchedHeaderValue");

    resultEndpoint.assertIsSatisfied();
}

@Override
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {

            from("direct:start").filter(header("foo").isEqualTo("bar")).to("mock:result");
        }
    };
}
}

```

Notice how it derives from the Camel helper class **CamelTestSupport** but has no Spring or Guice dependency injection configuration but instead overrides the **createRouteBuilder()** method.

Spring Test with XML Config Example

Here is the Spring Testing example using XML Config.

```

@ContextConfiguration
public class FilterTest extends AbstractJUnit38SpringContextTests {

    @EndpointInject(uri = "mock:result")
    protected MockEndpoint resultEndpoint;

    @Produce(uri = "direct:start")
    protected ProducerTemplate template;

    @DirtiesContext

```

```

public void testSendMatchingMessage() throws Exception {
    String expectedBody = "<matched/>";

    resultEndpoint.expectedBodiesReceived(expectedBody);

    template.sendBodyAndHeader(expectedBody, "foo", "bar");

    resultEndpoint.assertIsSatisfied();
}

@DirtiesContext
public void testSendNotMatchingMessage() throws Exception {
    resultEndpoint.expectedMessageCount(0);

    template.sendBodyAndHeader("<notMatched/>", "foo", "notMatchedHeaderValue");

    resultEndpoint.assertIsSatisfied();
}
}

```

Notice that we use **@DirtiesContext** on the test methods to force Spring Testing to automatically reload the CamelContext after each test method - this ensures that the tests don't clash with each other (e.g. one test method sending to an endpoint that is then reused in another test method).

Also notice the use of **@ContextConfiguration** to indicate that by default we should look for the FilterTest-context.xml on the classpath to configure the test case which looks like this

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/
camel-spring.xsd
">

    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:start"/>
            <filter>
                <xpath>$foo = 'bar'</xpath>
                <to uri="mock:result"/>
            </filter>
        </route>
    </camelContext>

</beans>

```

Spring Test with Java Config Example

Here is the Spring Testing example using Java Config. For more information see Spring Java Config.

```
@ContextConfiguration(
    locations =
"org.apache.camel.spring.javaconfig.patterns.FilterTest$ContextConfig",
    loader = JavaConfigContextLoader.class)
public class FilterTest extends AbstractJUnit4SpringContextTests {

    @EndpointInject(uri = "mock:result")
    protected MockEndpoint resultEndpoint;

    @Produce(uri = "direct:start")
    protected ProducerTemplate template;

    @DirtiesContext
    @Test
    public void testSendMatchingMessage() throws Exception {
        String expectedBody = "<matched/>";

        resultEndpoint.expectedBodiesReceived(expectedBody);

        template.sendBodyAndHeader(expectedBody, "foo", "bar");

        resultEndpoint.assertIsSatisfied();
    }

    @DirtiesContext
    @Test
    public void testSendNotMatchingMessage() throws Exception {
        resultEndpoint.expectedMessageCount(0);

        template.sendBodyAndHeader("<notMatched/>", "foo", "notMatchedHeaderValue");

        resultEndpoint.assertIsSatisfied();
    }

    @Configuration
    public static class ContextConfig extends SingleRouteCamelConfiguration {
        @Bean
        public RouteBuilder route() {
            return new RouteBuilder() {
                public void configure() {
                    from("direct:start").filter(header("foo").isEqualTo("bar")).to("mock:result");
                }
            };
        }
    }
}
```

This is similar to the XML Config example above except that there is no XML file and instead the nested **ContextConfig** class does all of the configuration; so your entire test case is contained in a single Java class. We currently have to reference by class name this class in the **@ContextConfiguration** which is a bit ugly. Please vote for SJC-238 to address this and make Spring Test work more cleanly with Spring JavaConfig.

Its totally optional but for the ContextConfig implementation we derive from **SingleRouteCamelConfiguration** which is a helper Spring Java Config class which will configure the CamelContext for us and then register the RouteBuilder we create.

Blueprint Test

Here is the Blueprint Testing example using XML Config.

```
// to use camel-test-blueprint, then extend the CamelBlueprintTestSupport class,
// and add your unit tests methods as shown below.
public class DebugBlueprintTest extends CamelBlueprintTestSupport {

    // override this method, and return the location of our Blueprint XML file to be
    // used for testing
    @Override
    protected String getBlueprintDescriptor() {
        return "org/apache/camel/test/blueprint/camelContext.xml";
    }

    // here we have regular Junit @Test method
    @Test
    public void testRoute() throws Exception {
        // set mock expectations
        getMockEndpoint("mock:a").expectedMessageCount(1);

        // send a message
        template.sendBody("direct:start", "World");

        // assert mocks
        assertMockEndpointsSatisfied();
    }
}
```

Also notice the use of **getBlueprintDescriptors** to indicate that by default we should look for the camelContext.xml in the package to configure the test case which looks like this

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="
```

```

    http://www.osgi.org/xmlns/blueprint/v1.0.0 http://www.osgi.org/xmlns/
blueprint/v1.0.0/blueprint.xsd">

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">

        <route>
            <from uri="direct:start"/>
            <transform>
                <simple>Hello ${body}</simple>
            </transform>
            <to uri="mock:a"/>
        </route>

    </camelContext>

</blueprint>

```

Testing endpoints

Camel provides a number of endpoints which can make testing easier.

Name	Description
DataSet	For load & soak testing this endpoint provides a way to create huge numbers of messages for sending to Components and asserting that they are consumed correctly
Mock	For testing routes and mediation rules using mocks and allowing assertions to be added to an endpoint
Test	Creates a Mock endpoint which expects to receive all the message bodies that could be polled from the given underlying endpoint

The main endpoint is the Mock endpoint which allows expectations to be added to different endpoints; you can then run your tests and assert that your expectations are met at the end.

Stubbing out physical transport technologies

If you wish to test out a route but want to avoid actually using a real physical transport (for example to unit test a transformation route rather than performing a full integration test) then the following endpoints can be useful.

Name	Description
Direct	Direct invocation of the consumer from the producer so that single threaded (non-SEDA) in VM invocation is performed which can be useful to mock out physical transports

SEDA Delivers messages asynchronously to consumers via a `java.util.concurrent.BlockingQueue` which is good for testing asynchronous transports

Testing existing routes

Camel provides some features to aid during testing of existing routes where you cannot or will not use Mock etc. For example you may have a production ready route which you want to test with some 3rd party API which sends messages into this route.

Name	Description
NotifyBuilder	Allows you to be notified when a certain condition has occurred. For example when the route has completed 5 messages. You can build complex expressions to match your criteria when to be notified.
AdviceWith	Allows you to advise or enhance an existing route using a RouteBuilder style. For example you can add interceptors to intercept sending outgoing messages to assert those messages are as expected.

CAMEL TEST

As a simple alternative to using Spring Testing or Guice the **camel-test** module was introduced into the Camel 2.0 trunk so you can perform powerful Testing of your Enterprise Integration Patterns easily.

Adding to your pom.xml

To get started using Camel Test you will need to add an entry to your pom.xml

JUnit

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




The camel-test JAR is using JUnit. There is an alternative camel-testng JAR (Camel 2.8 onwards) using the TestNG test framework.

TestNG

Available as of Camel 2.8

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

You might also want to add slf4j and log4j to ensure nice logging messages (and maybe adding a log4j.properties file into your src/test/resources directory).

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <scope>test</scope>
</dependency>
```

Writing your test

You firstly need to derive from the class **CamelTestSupport** and typically you will need to override the **createRouteBuilder()** method to create routes to be tested.

Here is an example.

```
public class FilterTest extends CamelTestSupport {

    @EndpointInject(uri = "mock:result")
    protected MockEndpoint resultEndpoint;

    @Produce(uri = "direct:start")
    protected ProducerTemplate template;

    @Test
```

```

public void testSendMatchingMessage() throws Exception {
    String expectedBody = "<matched/>";

    resultEndpoint.expectedBodiesReceived(expectedBody);

    template.sendBodyAndHeader(expectedBody, "foo", "bar");

    resultEndpoint.assertIsSatisfied();
}

@Test
public void testSendNotMatchingMessage() throws Exception {
    resultEndpoint.expectedMessageCount(0);

    template.sendBodyAndHeader("<notMatched/>", "foo", "notMatchedHeaderValue");

    resultEndpoint.assertIsSatisfied();
}

@Override
protected RouteBuilder createRouteBuilder() {
    return new RouteBuilder() {
        public void configure() {
            from("direct:start").filter(header("foo").isEqualTo("bar")).to("mock:result");
        }
    };
}
}

```

Notice how you can use the various Camel binding and injection annotations to inject individual Endpoint objects - particularly the Mock endpoints which are very useful for Testing. Also you can inject producer objects such as ProducerTemplate or some application code interface for sending messages or invoking services.

JNDI

Camel uses a Registry to allow you to configure Component or Endpoint instances or Beans used in your routes. If you are not using Spring or OSGi then JNDI is used as the default registry implementation.

So you will also need to create a **jndi.properties** file in your **src/test/resources** directory so that there is a default registry available to initialise the CamelContext.

Here is an example jndi.properties file

```
java.naming.factory.initial = org.apache.camel.util.jndi.CamelInitialContextFactory
```

Dynamically assigning ports

Available as of Camel 2.7

Tests that use port numbers will fail if that port is already on use. `AvailablePortFinder` provides methods for finding unused port numbers at runtime.

```
// Get the next available port number starting from the default starting port of 1024
int port1 = AvailablePortFinder.getNextAvailable();
/*
 * Get another port. Note that just getting a port number does not reserve it so
 * we look starting one past the last port number we got.
 */
int port2 = AvailablePortFinder.getNextAvailable(port1 + 1);
```

Setup CamelContext once per class, or per every test method

Available as of Camel 2.8

The Camel Test kit will by default setup and shutdown `CamelContext` per every test method in your test class. So for example if you have 3 test methods, then `CamelContext` is started and shutdown after each test, that is 3 times.

You may want to do this once, to share the `CamelContext` between test methods, to speedup unit testing. This requires to use JUnit 4! In your unit test method you have to extend the `org.apache.camel.test.junit4.CamelTestSupport` or the `org.apache.camel.test.junit4.CamelSpringTestSupport` test class and override the `isCreateCamelContextPerClass` method and return true as shown in the following example:

Listing 1. Setup CamelContext once per class

```
public class FilterCreateCamelContextPerClassTest extends CamelTestSupport {

    @Override
    public boolean isCreateCamelContextPerClass() {
        // we override this method and return true, to tell Camel test-kit that
        // it should only create CamelContext once (per class), so we will
        // re-use the CamelContext between each test method in this class
        return true;
    }

    @Test
    public void testSendMatchingMessage() throws Exception {
        String expectedBody = "<matched/>";

        getMockEndpoint("mock:result").expectedBodiesReceived(expectedBody);
    }
}
```



TestNG

This feature is also supported in camel-testng



Beware

When using this the CamelContext will keep state between tests, so have that in mind. So if your unit tests start to fail for no apparent reason, it could be due this fact. So use this feature with a bit of care.

```
        template.sendBodyAndHeader("direct:start", expectedBody, "foo", "bar");

        assertMockEndpointsSatisfied();
    }

    @Test
    public void testSendNotMatchingMessage() throws Exception {
        getMockEndpoint("mock:result").expectedMessageCount(0);

        template.sendBodyAndHeader("direct:start", "<notMatched/>", "foo",
"notMatchedHeaderValue");

        assertMockEndpointsSatisfied();
    }

    @Override
    protected RouteBuilder createRouteBuilder() {
        return new RouteBuilder() {
            public void configure() {
                from("direct:start").filter(header("foo").isEqualTo("bar")).to("mock:result");
            }
        };
    }
}
```

See Also

- Testing
- Mock
- Test

SPRING TESTING

Testing is a crucial part of any development or integration work. The Spring Framework offers a number of features that makes it easy to test while using Spring for Inversion of Control which works with JUnit 3.x, JUnit 4.x or TestNG.

We can reuse Spring for IoC and the Camel Mock and Test endpoints to create sophisticated integration tests that are easy to run and debug inside your IDE.

For example here is a simple unit test

```
import org.apache.camel.CamelContext;
import org.apache.camel.component.mock.MockEndpoint;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.AbstractJUnit38SpringContextTests;

@ContextConfiguration
public class MyCamelTest extends AbstractJUnit38SpringContextTests {

    @Autowired
    protected CamelContext camelContext;

    public void testMocksAreValid() throws Exception {
        MockEndpoint.assertIsSatisfied(camelContext);
    }
}
```

This test will load a Spring XML configuration file called **MyCamelTest-context.xml** from the classpath in the same package structure as the MyCamelTest class and initialize it along with any Camel routes we define inside it, then inject the CamelContext instance into our test case.

For instance, like this maven folder layout:

```
src/main/java/com/mycompany/MyCamelTest.class
src/main/resources/com/mycompany/MyCamelTest-context.xml
```

Spring Test with Java Config Example

You can completely avoid using an XML configuration file by using Spring Java Config.

Here is an example using Java Config.

```
@ContextConfiguration(
    locations =
    "org.apache.camel.spring.javaconfig.patterns.FilterTest$ContextConfig",
    loader = JavaConfigContextLoader.class)
```

```

public class FilterTest extends AbstractJUnit4SpringContextTests {

    @EndpointInject(uri = "mock:result")
    protected MockEndpoint resultEndpoint;

    @Produce(uri = "direct:start")
    protected ProducerTemplate template;

    @DirtiesContext
    @Test
    public void testSendMatchingMessage() throws Exception {
        String expectedBody = "<matched/>";

        resultEndpoint.expectedBodiesReceived(expectedBody);

        template.sendBodyAndHeader(expectedBody, "foo", "bar");

        resultEndpoint.assertIsSatisfied();
    }

    @DirtiesContext
    @Test
    public void testSendNotMatchingMessage() throws Exception {
        resultEndpoint.expectedMessageCount(0);

        template.sendBodyAndHeader("<notMatched/>", "foo", "notMatchedHeaderValue");

        resultEndpoint.assertIsSatisfied();
    }

    @Configuration
    public static class ContextConfig extends SingleRouteCamelConfiguration {
        @Bean
        public RouteBuilder route() {
            return new RouteBuilder() {
                public void configure() {
                    from("direct:start").filter(header("foo").isEqualTo("bar")).to("mock:result");
                }
            };
        }
    }
}

```

This is similar to the XML Config example above except that there is no XML file and instead the nested **ContextConfig** class does all of the configuration; so your entire test case is contained in a single Java class. We currently have to reference by class name this class in the **@ContextConfiguration** which is a bit ugly. Please vote for [SJC-238](#) to address this and make Spring Test work more cleanly with Spring JavaConfig.

Adding more Mock expectations

If you wish to programmatically add any new assertions to your test you can easily do so with the following. Notice how we use `@EndpointInject` to inject a Camel endpoint into our code then the Mock API to add an expectation on a specific message.

```
@ContextConfiguration
public class MyCamelTest extends AbstractJUnit38SpringContextTests {

    @Autowired
    protected CamelContext camelContext;

    @EndpointInject(uri = "mock:foo")
    protected MockEndpoint foo;

    public void testMocksAreValid() throws Exception {
        // lets add more expectations
        foo.message(0).header("bar").isEqualTo("ABC");

        MockEndpoint.assertIsSatisfied(camelContext);
    }
}
```

Further processing the received messages

Sometimes once a Mock endpoint has received some messages you want to then process them further to add further assertions that your test case worked as you expect.

So you can then process the received message exchanges if you like...

```
@ContextConfiguration
public class MyCamelTest extends AbstractJUnit38SpringContextTests {

    @Autowired
    protected CamelContext camelContext;

    @EndpointInject(uri = "mock:foo")
    protected MockEndpoint foo;

    public void testMocksAreValid() throws Exception {
        // lets add more expectations...

        MockEndpoint.assertIsSatisfied(camelContext);

        // now lets do some further assertions
        List<Exchange> list = foo.getReceivedExchanges();
        for (Exchange exchange : list) {
            Message in = exchange.getIn();
            ...
        }
    }
}
```

```
}  
}  
}
```

Sending and receiving messages

It might be that the Enterprise Integration Patterns you have defined in either Spring XML or using the Java DSL do all of the sending and receiving and you might just work with the Mock endpoints as described above. However sometimes in a test case its useful to explicitly send or receive messages directly.

To send or receive messages you should use the Bean Integration mechanism. For example to send messages inject a `ProducerTemplate` using the `@EndpointInject` annotation then call the various send methods on this object to send a message to an endpoint. To consume messages use the `@MessageDriven` annotation on a method to have the method invoked when a message is received.

```
public class Foo {  
    @EndpointInject(uri="activemq:foo.bar")  
    ProducerTemplate producer;  
  
    public void doSomething() {  
        // lets send a message!  
        producer.sendBody("<hello>world!</hello>");  
    }  
  
    // lets consume messages from the 'cheese' queue  
    @MessageDriven(uri="activemq:cheese")  
    public void onCheese(String name) {  
        ...  
    }  
}
```

See Also

- a real example test case using Mock and Spring along with its Spring XML
- Bean Integration
- Mock endpoint
- Test endpoint

CAMEL GUICE

As of 1.5 we now have support for Google Guice as a dependency injection framework. To use it just be dependent on **camel-guice.jar** which also depends on the following jars.

Dependency Injecting Camel with Guice

The GuiceCamelContext is designed to work nicely inside Guice. You then need to bind it using some Guice Module.

The camel-guice library comes with a number of reusable Guice Modules you can use if you wish - or you can bind the GuiceCamelContext yourself in your own module.

- CamelModule is the base module which binds the GuiceCamelContext but leaves it up to you to bind the RouteBuilder instances
- CamelModuleWithRouteTypes extends CamelModule so that in the constructor of the module you specify the RouteBuilder classes or instances to use
- CamelModuleWithMatchingRoutes extends CamelModule so that all bound RouteBuilder instances will be injected into the CamelContext or you can supply an optional Matcher to find RouteBuilder instances matching some kind of predicate.

So you can specify the exact RouteBuilder instances you want

```
Injector injector = Guice.createInjector(new
CamelModuleWithRouteTypes(MyRouteBuilder.class, AnotherRouteBuilder.class));
// if required you can lookup the CamelContext
CamelContext camelContext = injector.getInstance(CamelContext.class);
```

Or inject them all

```
Injector injector = Guice.createInjector(new CamelModuleWithRouteTypes());
// if required you can lookup the CamelContext
CamelContext camelContext = injector.getInstance(CamelContext.class);
```

You can then use Guice in the usual way to inject the route instances or any other dependent objects.

Bootstrapping with JNDI

A common pattern used in J2EE is to bootstrap your application or root objects by looking them up in JNDI. This has long been the approach when

working with JMS for example - looking up the JMS ConnectionFactory in JNDI for example.

You can follow a similar pattern with Guice using the GuiceyFruit JNDI Provider which lets you bootstrap Guice from a **jndi.properties** file which can include the Guice Modules to create along with environment specific properties you can inject into your modules and objects.

If the **jndi.properties** is conflict with other component, you can specify the jndi properties file name in the Guice Main with option -j or -jndiProperties with the properties file location to let Guice Main to load right jndi properties file.

Configuring Component, Endpoint or RouteBuilder instances

You can use Guice to dependency inject whatever objects you need to create, be it an Endpoint, Component, RouteBuilder or arbitrary bean used within a route.

The easiest way to do this is to create your own Guice Module class which extends one of the above module classes and add a provider method for each object you wish to create. A provider method is annotated with **@Provides** as follows

```
public class MyModule extends CamelModuleWithMatchingRoutes {  
  
    @Provides  
    @JndiBind("jms")  
    JmsComponent jms(@Named("activemq.brokerURL") String brokerUrl) {  
        return JmsComponent.jmsComponent(new ActiveMQConnectionFactory(brokerUrl));  
    }  
}
```

You can optionally annotate the method with **@JndiBind** to bind the object to JNDI at some name if the object is a component, endpoint or bean you wish to refer to by name in your routes.

You can inject any environment specific properties (such as URLs, machine names, usernames/passwords and so forth) from the jndi.properties file easily using the **@Named** annotation as shown above. This allows most of your configuration to be in Java code which is typesafe and easily refactorable - then leaving some properties to be environment specific (the jndi.properties file) which you can then change based on development, testing, production etc.

Creating multiple RouteBuilder instances per type

It is sometimes useful to create multiple instances of a particular RouteBuilder with different configurations.

To do this just create multiple provider methods for each configuration; or create a single provider method that returns a collection of RouteBuilder instances.

For example

```
import org.apache.camel.guice.CamelModuleWithMatchingRoutes;
import com.google.common.collect.Lists;

public class MyModule extends CamelModuleWithMatchingRoutes {

    @Provides
    @JndiBind("foo")
    Collection<RouteBuilder> foo(@Named("fooUrl") String fooUrl) {
        return Lists.newArrayList(new MyRouteBuilder(fooUrl), new
MyRouteBuilder("activemq:CheeseQueue"));
    }
}
```

See Also

- there are a number of Examples you can look at to see Guice and Camel being used such as Guice JMS Example
- Guice Maven Plugin for running your Guice based routes via Maven

TEMPLATING

When you are testing distributed systems its a very common requirement to have to stub out certain external systems with some stub so that you can test other parts of the system until a specific system is available or written etc.

A great way to do this is using some kind of Template system to generate responses to requests generating a dynamic message using a mostly-static body.

There are a number of templating components included in the Camel distribution you could use

- FreeMarker
- StringTemplate
- Velocity
- XQuery
- XSLT

or the following external Camel components

- Scalate

Example

Here's a simple example showing how we can respond to InOut requests on the **My.Queue** queue on ActiveMQ with a template generated response. The reply would be sent back to the JMSReplyTo Destination.

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm");
```

If you want to use InOnly and consume the message and send it to another destination you could use

```
from("activemq:My.Queue").
  to("velocity:com/acme/MyResponse.vm").
  to("activemq:Another.Queue");
```

See Also

- Mock for details of mock endpoint testing (as opposed to template based stubs).

DATABASE

Camel can work with databases in a number of different ways. This document tries to outline the most common approaches.

Database endpoints

Camel provides a number of different endpoints for working with databases

- JPA for working with hibernate, openjpa or toplink. When consuming from the endpoints entity beans are read (and deleted/updated to mark as processed) then when producing to the endpoints they are written to the database (via insert/update).
- iBATIS similar to the above but using Apache iBATIS
- JDBC similar though using explicit SQL

Database pattern implementations

Various patterns can work with databases as follows

- Idempotent Consumer

- Aggregator
- BAM for business activity monitoring

PARALLEL PROCESSING AND ORDERING

It is a common requirement to want to use parallel processing of messages for throughput and load balancing, while at the same time process certain kinds of messages in order.

How to achieve parallel processing

You can send messages to a number of Camel Components to achieve parallel processing and load balancing such as

- SEDA for in-JVM load balancing across a thread pool
- ActiveMQ or JMS for distributed load balancing and parallel processing
- JPA for using the database as a poor mans message broker

When processing messages concurrently, you should consider ordering and concurrency issues. These are described below

Concurrency issues

Note that there is no concurrency or locking issue when using ActiveMQ, JMS or SEDA by design; they are designed for highly concurrent use. However there are possible concurrency issues in the Processor of the messages i.e. what the processor does with the message?

For example if a processor of a message transfers money from one account to another account; you probably want to use a database with pessimistic locking to ensure that operation takes place atomically.

Ordering issues

As soon as you send multiple messages to different threads or processes you will end up with an unknown ordering across the entire message stream as each thread is going to process messages concurrently.

For many use cases the order of messages is not too important. However for some applications this can be crucial. e.g. if a customer submits a purchase order version 1, then amends it and sends version 2; you don't want to process the first version last (so that you loose the update). Your Processor might be clever enough to ignore old messages. If not you need to preserve order.

Recommendations

This topic is large and diverse with lots of different requirements; but from a high level here are our recommendations on parallel processing, ordering and concurrency

- for distributed locking, use a database by default, they are very good at it 😊
- to preserve ordering across a JMS queue consider using Exclusive Consumers in the ActiveMQ component
- even better are Message Groups which allows you to preserve ordering across messages while still offering parallelisation via the **JMSXGroupID** header to determine what can be parallelized
- if you receive messages out of order you could use the Resequencer to put them back together again

A good rule of thumb to help reduce ordering problems is to make sure each single can be processed as an atomic unit in parallel (either without concurrency issues or using say, database locking); or if it can't, use a Message Group to relate the messages together which need to be processed in order by a single thread.

Using Message Groups with Camel

To use a Message Group with Camel you just need to add a header to the output JMS message based on some kind of Correlation Identifier to correlate messages which should be processed in order by a single thread - so that things which don't correlate together can be processed concurrently.

For example the following code shows how to create a message group using an XPath expression taking an invoice's product code as the Correlation Identifier

```
from("activemq:a").setHeader("JMSXGroupID", xpath("/invoice/  
productCode")).to("activemq:b");
```

You can of course use the Xml Configuration if you prefer

ASYNCHRONOUS PROCESSING

Overview

Camel supports a more complex asynchronous processing model. The asynchronous processors implement the AsyncProcessor interface which is derived from the more synchronous Processor interface. There are



Supported versions

The information on this page applies for the Camel 1.x and Camel 2.4 onwards. In Camel 1.x the asynchronous processing is only implemented for JBI where as in Camel 2.4 onwards we have implemented it in many other areas. See more at Asynchronous Routing Engine.

advantages and disadvantages when using asynchronous processing when compared to using the standard synchronous processing model.

Advantages:

- Processing routes that are composed fully of asynchronous processors do not use up threads waiting for processors to complete on blocking calls. This can increase the scalability of your system by reducing the number of threads needed to process the same workload.
- Processing routes can be broken up into SEDA processing stages where different thread pools can process the different stages. This means that your routes can be processed concurrently.

Disadvantages:

- Implementing asynchronous processors is more complex than implementing the synchronous versions.

When to Use

We recommend that processors and components be implemented the more simple synchronous APIs unless you identify a performance or scalability requirement that dictates otherwise. A Processor whose process() method blocks for a long time would be good candidates for being converted into an asynchronous processor.

Interface Details

```
public interface AsyncProcessor extends Processor {
    boolean process(Exchange exchange, AsyncCallback callback);
}
```

The AsyncProcessor defines a single process() method which is very similar to its synchronous Processor.process() brethren. Here are the differences:

- A non-null AsyncCallback **MUST** be supplied which will be notified when the exchange processing is completed.

- It **MUST** not throw any exceptions that occurred while processing the exchange. Any such exceptions must be stored on the exchange's Exception property.
- It **MUST** know if it will complete the processing synchronously or asynchronously. The method will return true if it does complete synchronously, otherwise it returns false.
- When the processor has completed processing the exchange, it must call the callback.done(boolean sync) method. The sync parameter **MUST** match the value returned by the process() method.

Implementing Processors that Use the AsyncProcessor API

All processors, even synchronous processors that do not implement the AsyncProcessor interface, can be coerced to implement the AsyncProcessor interface. This is usually done when you are implementing a Camel component consumer that supports asynchronous completion of the exchanges that it is pushing through the Camel routes. Consumers are provided a Processor object when created. All Processor object can be coerced to a AsyncProcessor using the following API:

```
Processor processor = ...
AsyncProcessor asyncProcessor = AsyncProcessorTypeConverter.convert(processor);
```

For a route to be fully asynchronous and reap the benefits to lower Thread usage, it must start with the consumer implementation making use of the asynchronous processing API. If it called the synchronous process() method instead, the consumer's thread would be forced to be blocked and in use for the duration that it takes to process the exchange.

It is important to take note that just because you call the asynchronous API, it does not mean that the processing will take place asynchronously. It only allows the possibility that it can be done without tying up the caller's thread. If the processing happens asynchronously is dependent on the configuration of the Camel route.

Normally, the the process call is passed in an inline inner AsyncCallback class instance which can reference the exchange object that was declared final. This allows it to finish up any post processing that is needed when the called processor is done processing the exchange. See below for an example.

```
final Exchange exchange = ...
AsyncProcessor asyncProcessor = ...
asyncProcessor.process(exchange, new AsyncCallback() {
    public void done(boolean sync) {
```



```

    if (exchange.isFailed()) {
        ... // do failure processing.. perhaps rollback etc.
    } else {
        ... // processing completed successfully, finish up
            // perhaps commit etc.
    }
}
});

```

Asynchronous Route Sequence Scenarios

Now that we have understood the interface contract of the AsyncProcessor, and have seen how to make use of it when calling processors, lets look at what the thread model/sequence scenarios will look like for some sample routes.

The Jetty component's consumers support async processing by using continuations. Suffice to say it can take a http request and pass it to a camel route for async processing. If the processing is indeed async, it uses Jetty continuation so that the http request is 'parked' and the thread is released. Once the camel route finishes processing the request, the jetty component uses the AsyncCallback to tell Jetty to 'un-park' the request. Jetty un-parks the request, the http response returned using the result of the exchange processing.

Notice that the jetty continuations feature is only used "If the processing is indeed async". This is why AsyncProcessor.process() implementations MUST accurately report if request is completed synchronously or not.

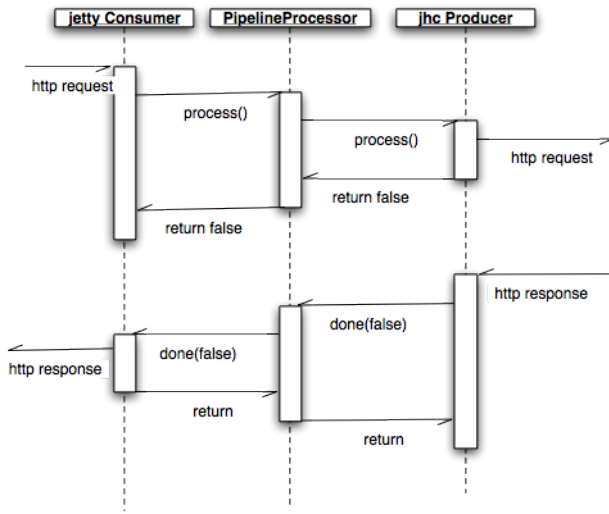
The jhc component's producer allows you to make HTTP requests and implement the AsyncProcessor interface. A route that uses both the jetty asynchronous consumer and the jhc asynchronous producer will be a fully asynchronous route and has some nice attributes that can be seen if we take a look at a sequence diagram of the processing route. For the route:

```

from("jetty:http://localhost:8080/service").to("jhc:http://localhost/service-impl");

```

The sequence diagram would look something like this:



The diagram simplifies things by making it look like processors implement the AsyncCallback interface when in reality the AsyncCallback interfaces are inline inner classes, but it illustrates the processing flow and shows how 2 separate threads are used to complete the processing of the original http request. The first thread is synchronous up until processing hits the jhc producer which issues the http request. It then reports that the exchange processing will complete async since it will use a NIO to complete getting the response back. Once the jhc component has received a full response it uses AsyncCallback.done() method to notify the caller. These callback notifications continue up until it reaches the original jetty consumer which then un-parks the http request and completes it by providing the response.

Mixing Synchronous and Asynchronous Processors

It is totally possible and reasonable to mix the use of synchronous and asynchronous processors/components. The pipeline processor is the backbone of a Camel processing route. It glues all the processing steps together. It is implemented as an AsyncProcessor and supports interleaving synchronous and asynchronous processors as the processing steps in the pipeline.

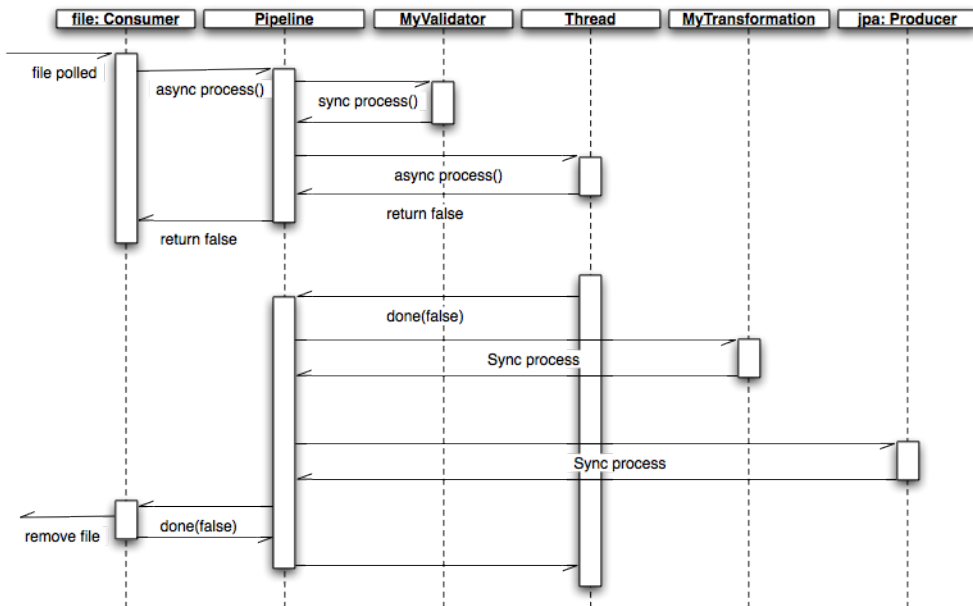
Lets say we have 2 custom processors, MyValidator and MyTransformation, both of which are synchronous processors. Lets say we want to load file from the data/in directory validate them with the MyValidator() processor, Transform them into JPA java objects using MyTransformation and then insert them into the database using the JPA component. Lets say that the transformation process takes quite a bit of time and we want to allocate 20 threads to do parallel transformations of the input files. The solution is to

make use of the thread processor. The thread is AsyncProcessor that forces subsequent processing in asynchronous thread from a thread pool.

The route might look like:

```
from("file:data/in").process(new MyValidator()).threads(20).process(new MyTransformation()).to("jpa:PurchaseOrder");
```

The sequence diagram would look something like this:



You would actually have multiple threads executing the 2nd part of the thread sequence.

Staying synchronous in an AsyncProcessor

Generally speaking you get better throughput processing when you process things synchronously. This is due to the fact that starting up an asynchronous thread and doing a context switch to it adds a little bit of overhead. So it is generally encouraged that AsyncProcessors do as much work as they can synchronously. When they get to a step that would block for a long time, at that point they should return from the process call and let the caller know that it will be completing the call asynchronously.

IMPLEMENTING VIRTUAL TOPICS ON OTHER JMS PROVIDERS

ActiveMQ supports Virtual Topics since durable topic subscriptions kinda suck (see this page for more detail) mostly since they don't support Competing Consumers.

Most folks want Queue semantics when consuming messages; so that you can support Competing Consumers for load balancing along with things like Message Groups and Exclusive Consumers to preserve ordering or partition the queue across consumers.

However if you are using another JMS provider you can implement Virtual Topics by switching to ActiveMQ 😊 or you can use the following Camel pattern.

First here's the ActiveMQ approach.

- send to **activemq:topic:VirtualTopic.Orders**
- for consumer A consume from **activemq:Consumer.A.VirtualTopic.Orders**

When using another message broker use the following pattern

- send to **jms:Orders**
- add this route with a `to()` for each logical durable topic subscriber

```
from("jms:Orders").to("jms:Consumer.A", "jms:Consumer.B", ...);
```

- for consumer A consume from **jms:Consumer.A**

WHAT'S THE CAMEL TRANSPORT FOR CXF

In CXF you offer or consume a webservice by defining it's address. The first part of the address specifies the protocol to use. For example `address="http://localhost:90000"` in an endpoint configuration means your service will be offered using the http protocol on port 9000 of localhost. When you integrate Camel Transport into CXF you get a new transport "camel". So you can specify `address="camel://direct:MyEndpointName"` to bind the CXF service address to a camel direct endpoint.

Technically speaking Camel transport for CXF is a component which implements the CXF transport API with the Camel core library. This allows you to use camel's routing engine and integration patterns support smoothly together with your CXF services.

INTEGRATE CAMEL INTO CXF TRANSPORT LAYER

To include the Camel Transport into your CXF bus you use the CamelTransportFactory. You can do this in Java as well as in Spring.

Setting up the Camel Transport in Spring

You can use the following snippet in your applicationcontext if you want to configure anything special. If you only want to activate the camel transport you do not have to do anything in your application context. As soon as you include the camel-cxf jar in your app cxf will scan the jar and load a CamelTransportFactory for you.

```
<bean class="org.apache.camel.component.cxf.transport.CamelTransportFactory">
  <property name="bus" ref="cxf" />
  <property name="camelContext" ref="camelContext" />
  <!-- checkException new added in Camel 2.1 and Camel 1.6.2 -->
  <!-- If checkException is true , CamelDestination will check the outMessage's
    exception and set it into camel exchange. You can also override this value
    in CamelDestination's configuration. The default value is false.
    This option should be set true when you want to leverage the camel's error
    handler to deal with fault message -->
  <property name="checkException" value="true" />
  <property name="transportIds">
    <list>
      <value>http://cxf.apache.org/transports/camel</value>
    </list>
  </property>
</bean>
```

Integrating the Camel Transport in a programmatic way

Camel transport provides a setContext method that you could use to set the Camel context into the transport factory. If you want this factory take effect, you need to register the factory into the CXF bus. Here is a full example for you.

```
import org.apache.cxf.Bus;
import org.apache.cxf.BusFactory;
import org.apache.cxf.transport.ConduitInitiatorManager;
import org.apache.cxf.transport.DestinationFactoryManager;
...

BusFactory bf = BusFactory.newInstance();
Bus bus = bf.createBus();
CamelTransportFactory camelTransportFactory = new CamelTransportFactory();
camelTransportFactory.setCamelContext(context)
// register the conduit initiator
```

```

ConduitInitiatorManager cim = bus.getExtension(ConduitInitiatorManager.class);
cim.registerConduitInitiator(CamelTransportFactory.TRANSPORT_ID,
camelTransportFactory);
// register the destination factory
DestinationFactoryManager dfm = bus.getExtension(DestinationFactoryManager.class);
dfm.registerDestinationFactory(CamelTransportFactory.TRANSPORT_ID,
camelTransportFactory);
// set or bus as the default bus for cxf
BusFactory.setDefaultBus(bus);

```

CONFIGURE THE DESTINATION AND CONDUIT

Namespace

The elements used to configure an Camel transport endpoint are defined in the namespace `http://cxf.apache.org/transports/camel`. It is commonly referred to using the prefix `camel`. In order to use the Camel transport configuration elements you will need to add the lines shown below to the `beans` element of your endpoint's configuration file. In addition, you will need to add the configuration elements' namespace to the `xsi:schemaLocation` attribute.

Listing 1. Adding the Configuration Namespace

```

<beans ...
  xmlns:camel="http://cxf.apache.org/transports/camel"
  ...
  xsi:schemaLocation="...
    http://cxf.apache.org/transports/camel
    http://cxf.apache.org/transports/camel.xsd
  ...>

```

The destination element

You configure an Camel transport server endpoint using the `camel:destination` element and its children. The `camel:destination` element takes a single attribute, `name`, the specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form `portQName.camel-destination`. The example below shows the `camel:destination` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort">` if the endpoint's target namespace was `http://widgets.widgetvendor.net`.

Listing 1. camel:destination Element

```

...
<camel:destination name="{http://widgets/
widgetvndor.net}widgetSOAPPort.http-destination">
  <camelContext id="context" xmlns="http://activemq.apache.org/camel/schema/spring">
    <route>
      <from uri="direct:EndpointC" />
      <to uri="direct:EndpointD" />
    </route>
  </camelContext>
</camel:destination>
...

```

The `camel:destination` element has a number of child elements that specify configuration information. They are described below.

Element	Description
<code>camel-spring:camelContext</code>	You can specify the camel context in the camel destination
<code>camel:camelContextRef</code>	The camel context id which you want inject into the camel destination

The conduit element

You configure an Camel transport client using the `camel:conduit` element and its children. The `camel:conduit` element takes a single attribute, `name`, that specifies the WSDL port element that corresponds to the endpoint. The value for the `name` attribute takes the form `portQName.camel-conduit`. For example, the code below shows the `camel:conduit` element that would be used to add configuration for an endpoint that was specified by the WSDL fragment `<port binding="widgetSOAPBinding" name="widgetSOAPPort">` if the endpoint's target namespace was `http://widgets.widgetvndor.net`.

Listing 1. http-conf:conduit Element

```

...
<camelContext id="conduit_context" xmlns="http://activemq.apache.org/camel/schema/
spring">
  <route>
    <from uri="direct:EndpointA" />
    <to uri="direct:EndpointB" />
  </route>
</camelContext>

<camel:conduit name="{http://widgets/widgetvndor.net}widgetSOAPPort.camel-conduit">
  <camel:camelContextRef>conduit_context</camel:camelContextRef>
</camel:conduit>

```

```

<camel:conduit name="*.camel-conduit">
  <!-- you can also using the wild card to specify the camel-conduit that you want to
configure -->
  ...
  </camel:conduit>
  ...

```

The `camel:conduit` element has a number of child elements that specify configuration information. They are described below.

Element	Description
<code>camel-spring:camelContext</code>	You can specify the camel context in the camel conduit
<code>camel:camelContextRef</code>	The camel context id which you want inject into the camel conduit

EXAMPLE USING CAMEL AS A LOAD BALANCER FOR CXF

This example show how to use the camel load balance feature in CXF, and you need load the configuration file in CXF and publish the endpoints on the address "camel://direct:EndpointA" and "camel://direct:EndpointB"

```

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:camel="http://cxf.apache.org/transports/camel"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://cxf.apache.org/transports/camel http://cxf.apache.org/transports/
camel.xsd
    http://camel.apache.org/schema/cxf http://camel.apache.org/schema/cxf/
cxfEndpoint.xsd
    http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/
camel-spring.xsd
">

  <bean id = "roundRobinRef"
class="org.apache.camel.processor.loadbalancer.RoundRobinLoadBalancer" />

  <camelContext id="dest_context" xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="jetty:http://localhost:9091/GreeterContext/GreeterPort"/>
      <loadBalance ref="roundRobinRef">
        <to uri="direct:EndpointA"/>
        <to uri="direct:EndpointB"/>
      </loadBalance>
    </route>
  </camelContext>

```



```

</camelContext>

<!-- Inject the camel context to the Camel transport's destination -->
<camel:destination name="{http://apache.org/
hello_world_soap_http}CamelPort.camel-destination">
  <camel:camelContextRef>dest_context</camel:camelContextRef>
</camel:destination>

</beans>

```

COMPLETE HOWTO AND EXAMPLE FOR ATTACHING CAMEL TO CXF

Better JMS Transport for CXF Webservice using Apache Camel

INTRODUCTION

When sending an Exchange to an Endpoint you can either use a Route or a ProducerTemplate. This works fine in many scenarios. However you may need to guarantee that an exchange is delivered to the same endpoint that you delivered a previous exchange on. For example in the case of delivering a batch of exchanges to a MINA socket you may need to ensure that they are all delivered through the same socket connection. Furthermore once the batch of exchanges have been delivered the protocol requirements may be such that you are responsible for closing the socket.

USING A PRODUCER

To achieve fine grained control over sending exchanges you will need to program directly to a Producer. Your code will look similar to:

```

CamelContext camelContext = ...

// Obtain an endpoint and create the producer we will be using.
Endpoint endpoint = camelContext.getEndpoint("someuri:etc");
Producer producer = endpoint.createProducer();
producer.start();

try {
  // For each message to send...
  Object requestMessage = ...
  Exchange exchangeToSend = producer.createExchange();
  exchangeToSend().setBody(requestMessage);
  producer.process(exchangeToSend);
}

```

```
...  
} finally {  
    // Tidy the producer up.  
    producer.stop();  
}
```

In the case of using Apache MINA the `producer.stop()` invocation will cause the socket to be closed.

Tutorials

There now follows the documentation on camel tutorials

We have a number of tutorials as listed below. The tutorials often comes with source code which is either available in the Camel Download or attached to the wiki page.

- OAuth Tutorial
This tutorial demonstrates how to implement OAuth for a web application with Camel's gauth component. The sample application of this tutorial is also online at <http://gauthcloud.appspot.com/>
- Tutorial for Camel on Google App Engine
This tutorial demonstrates the usage of the Camel Components for Google App Engine. The sample application of this tutorial is also online at <http://camelcloud.appspot.com/>
- Tutorial on Spring Remoting with JMS
This tutorial is focused on different techniques with Camel for Client-Server communication.
- Report Incident - This tutorial introduces Camel steadily and is based on a real life integration problem
This is a very long tutorial beginning from the start; its for entry level to Camel. Its based on a real life integration, showing how Camel can be introduced in an existing solution. We do this in baby steps. The tutorial is currently work in progress, so check it out from time to time. The tutorial explains some of the inner building blocks Camel uses under the covers. This is good knowledge to have when you start using Camel on a higher abstract level where it can do wonders in a few lines of routing DSL.
- Using Camel with ServiceMix a tutorial on using Camel inside Apache ServiceMix.
- Better JMS Transport for CXF Webservice using Apache Camel
Describes how to use the Camel Transport for CXF to attach a CXF Webservice to a JMS Queue
- Tutorial how to use good old Axis 1.4 with Camel
This tutorial shows that Camel does work with the good old frameworks such as AXIS that is/was widely used for WebService.
- Tutorial on using Camel in a Web Application
This tutorial gives an overview of how to use Camel inside Tomcat, Jetty or any other servlet engine
- Tutorial on Camel 1.4 for Integration
Another real-life scenario. The company sells widgets, with a



Notice

These tutorials listed below, is hosted at Apache. We offer the Articles page where we have a link collection for 3rd party Camel material, such as tutorials, blog posts, published articles, videos, pod casts, presentations, and so forth.

If you have written a Camel related article, then we are happy to provide a link to it. You can contact the Camel Team, for example using the Mailing Lists, (or post a tweet with the word Apache Camel).

somewhat unique business process (their customers periodically report what they've purchased in order to get billed). However every customer uses a different data format and protocol. This tutorial goes through the process of integrating (and testing!) several customers and their electronic reporting of the widgets they've bought, along with the company's response.

- Tutorial how to build a Service Oriented Architecture using Camel with OSGI - Updated 20/11/2009

The tutorial has been designed in two parts. The first part introduces basic concept to create a simple SOA solution using Camel and OSGI and deploy it in a OSGI Server like Apache Felix Karaf and Spring DM Server while the second extends the ReportIncident tutorial part 4 to show How we can separate the different layers (domain, service, ...) of an application and deploy them in separate bundles. The Web Application has also be modified in order to communicate to the OSGI bundles.

- Several of the vendors on the Commercial Camel Offerings page also offer various tutorials, webinars, examples, etc.... that may be useful.
- Examples
While not actual tutorials you might find working through the source of the various Examples useful.

TUTORIAL ON SPRING REMOTING WITH JMS

Â

PREFACE

This tutorial aims to guide the reader through the stages of creating a project which uses Camel to facilitate the routing of messages from a JMS queue to a



Thanks

This tutorial was kindly donated to Apache Camel by Martin Gilday.

Spring service. The route works in a synchronous fashion returning a response to the client.

- Tutorial on Spring Remoting with JMS
- Preface
- Prerequisites
- Distribution
- About
- Create the Camel Project
- Update the POM with Dependencies
- Writing the Server
- Create the Spring Service
- Define the Camel Routes
- Configure Spring
- Run the Server
- Writing The Clients
- Client Using The ProducerTemplate
- Client Using Spring Remoting
- Client Using Message Endpoint EIP Pattern
- Run the Clients
- Using the Camel Maven Plugin
- Using Camel JMX
- See Also

PREREQUISITES

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

DISTRIBUTION

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

ABOUT

This tutorial is a simple example that demonstrates more the fact how well Camel is seamless integrated with Spring to leverage the best of both worlds. This sample is client server solution using JMS messaging as the transport. The sample has two flavors of servers and also for clients demonstrating different techniques for easy communication.

The Server is a JMS message broker that routes incoming messages to a business service that does computations on the received message and returns a response.

The EIP patterns used in this sample are:

Pattern	Description
Message Channel	We need a channel so the Clients can communicate with the server.
Message	The information is exchanged using the Camel Message interface.
Message Translator	This is where Camel shines as the message exchange between the Server and the Clients are text based strings with numbers. However our business service uses int for numbers. So Camel can do the message translation automatically.
Message Endpoint	It should be easy to send messages to the Server from the the clients. This is archived with Camels powerful Endpoint pattern that even can be more powerful combined with Spring remoting. The tutorial have clients using each kind of technique for this.
Point to Point Channel	We using JMS queues so there are only one receive of the message exchange
Event Driven Consumer	Yes the JMS broker is of course event driven and only reacts when the client sends a message to the server.

We use the following Camel components:

Component	Description
ActiveMQ	We use Apache ActiveMQ as the JMS broker on the Server side
Bean	We use the bean binding to easily route the messages to our business service. This is a very powerful component in Camel.

File	In the AOP enabled Server we store audit trails as files.
JMS	Used for the JMS messaging

CREATE THE CAMEL PROJECT

```
mvn archetype:create -DgroupId=org.example -DartifactId=CamelWithJmsAndSpring
```


Update the POM with Dependencies

First we need to have dependencies for the core Camel jars, its spring, jms components and finally ActiveMQ as the message broker.

```
<!-- required by both client and server -->
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-core</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jms</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
</dependency>
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>activemq-camel</artifactId>
</dependency>
```

As we use spring xml configuration for the ActiveMQ JMS broker we need this dependency:

```
<!-- xbean is required for ActiveMQ broker configuration in the spring xml file -->
<dependency>
  <groupId>org.apache.xbean</groupId>
  <artifactId>xbean-spring</artifactId>
</dependency>
```

 For the purposes of the tutorial a single Maven project will be used for both the client and server. Ideally you would break your application down into the appropriate components.

WRITING THE SERVER

Create the Spring Service

For this example the Spring service (= our business service) on the server will be a simple multiplier which trebles in the received value.

```
public interface Multiplier {  
  
    /**  
     * Multiplies the given number by a pre-defined constant.  
     *  
     * @param originalNumber The number to be multiplied  
     * @return The result of the multiplication  
     */  
    int multiply(int originalNumber);  
  
}
```

And the implementation of this service is:

```
@Service(value = "multiplier")  
public class Treble implements Multiplier {  
  
    public int multiply(final int originalNumber) {  
        return originalNumber * 3;  
    }  
  
}
```

Notice that this class has been annotated with the `@Service` spring annotation. This ensures that this class is registered as a bean in the registry with the given name **multiplier**.

Define the Camel Routes

```
public class ServerRoutes extends RouteBuilder {  
  
    @Override  
    public void configure() throws Exception {
```



```

        // route from the numbers queue to our business that is a spring bean
        registered with the id=multiplier
        // Camel will introspect the multiplier bean and find the best candidate of
        the method to invoke.
        // You can add annotations etc to help Camel find the method to invoke.
        // As our multiplier bean only have one method its easy for Camel to find the
        method to use.
        from("jms:queue:numbers").to("multiplier");

        // Camel has several ways to configure the same routing, we have defined some
        of them here below

        // as above but with the bean: prefix
        //from("jms:queue:numbers").to("bean:multiplier");

        // beanRef is using explicit bean bindings to lookup the multiplier bean and
        invoke the multiply method
        //from("jms:queue:numbers").beanRef("multiplier", "multiply");

        // the same as above but expressed as a URI configuration
        //from("jms:queue:numbers").to("bean:multiplier?methodName=multiply");
    }
}

```

This defines a Camel route *from* the JMS queue named **numbers** to the Spring bean named **multiplier**. Camel will create a consumer to the JMS queue which forwards all received messages onto the the Spring bean, using the method named **multiply**.

Configure Spring

The Spring config file is placed under META-INF/spring as this is the default location used by the Camel Maven Plugin, which we will later use to run our server.

First we need to do the standard scheme declarations in the top. In the camel-server.xml we are using spring beans as the default **bean:** namespace and springs **context:**. For configuring ActiveMQ we use **broker:** and for Camel we of course have **camel:**. Notice that we don't use version numbers for the camel-spring schema. At runtime the schema is resolved in the Camel bundle. If we use a specific version number such as 1.4 then its IDE friendly as it would be able to import it and provide smart completion etc. See Xml Reference for further details.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:camel="http://camel.apache.org/schema/spring"

```

```

xmlns:broker="http://activemq.apache.org/schema/core"
xsi:schemaLocation="
  http://www.springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/
schema/context/spring-context.xsd
  http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/
camel-spring.xsd
  http://activemq.apache.org/schema/core http://activemq.apache.org/schema/
core/activemq-core-5.5.0.xsd">

```

We use Spring annotations for doing IoC dependencies and its component-scan features comes to the rescue as it scans for spring annotations in the given package name:

```

<!-- let Spring do its IoC stuff in this package -->
<context:component-scan base-package="org.apache.camel.example.server"/>

```

Camel will of course not be less than Spring in this regard so it supports a similar feature for scanning of Routes. This is configured as shown below. Notice that we also have enabled the JMXAgent so we will be able to introspect the Camel Server with a JMX Console.

```

<!-- declare a camel context that scans for classes that is RouteBuilder
in the package org.apache.camel.example.server -->
<camel:camelContext id="camel-server">
  <camel:package>org.apache.camel.example.server</camel:package>
  <!-- enable JMX connector so we can connect to the server and browse mbeans -->
  <!-- Camel will log at INFO level the service URI to use for connecting with
jconsole -->
  <camel:jmxAgent id="agent" createConnector="true"/>
</camel:camelContext>

```

The ActiveMQ JMS broker is also configured in this xml file. We set it up to listen on TCP port 61610.

```

<!-- lets configure the ActiveMQ JMS broker server to listen on TCP 61610 -->
<broker:broker useJmx="true" persistent="false" brokerName="myBroker">
  <broker:transportConnectors>
    <!-- expose a VM transport for in-JVM transport between AMQ and Camel on the
server side -->
    <broker:transportConnector name="vm" uri="vm://myBroker"/>
    <!-- expose a TCP transport for clients to use -->
    <broker:transportConnector name="tcp" uri="tcp://localhost:${tcp.port}"/>
  </broker:transportConnectors>
</broker:broker>

```

As this examples uses JMS then Camel needs a JMS component that is connected with the ActiveMQ broker. This is configured as shown below:

```
<!-- lets configure the Camel ActiveMQ to use the embedded ActiveMQ broker declared above -->
<bean id="jms" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="vm://myBroker"/>
</bean>
```

Notice: The JMS component is configured in standard Spring beans, but the gem is that the bean id can be referenced from Camel routes - meaning we can do routing using the JMS Component by just using **jms:** prefix in the route URI. What happens is that Camel will find in the Spring Registry for a bean with the id="jms". Since the bean id can have arbitrary name you could have named it id="jmsbroker" and then referenced to it in the routing as from="jmsbroker:queue:numbers).to("multiplier"); We use the vm protocol to connect to the ActiveMQ server as its embedded in this application.

component-scan	Defines the package to be scanned for Spring stereotype annotations, in this case, to load the "multiplier" bean
----------------	--

camel-context	Defines the package to be scanned for Camel routes. Will find the ServerRoutes class and create the routes contained within it
---------------	--

jms bean	Creates the Camel JMS component
----------	---------------------------------

Run the Server

The Server is started using the `org.apache.camel.spring.Main` class that can start camel-spring application out-of-the-box. The Server can be started in several flavors:

- as a standard java main application - just start the `org.apache.camel.spring.Main` class
- using maven `java:exec`
- using `camel:run`

In this sample as there are two servers (with and without AOP) we have prepared some profiles in maven to start the Server of your choice.

The server is started with:

```
mvn compile exec:java -PCamelServer
```

WRITING THE CLIENTS

This sample has three clients demonstrating different Camel techniques for communication

- CamelClient using the ProducerTemplate for Spring template style coding
- CamelRemoting using Spring Remoting
- CamelEndpoint using the Message Endpoint EIP pattern using a neutral Camel API

Client Using The ProducerTemplate

We will initially create a client by directly using ProducerTemplate. We will later create a client which uses Spring remoting to hide the fact that messaging is being used.

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

```
<camel:camelContext id="camel-client">
  <camel:template id="camelTemplate"/>
</camel:camelContext>
```

```
<!-- Camel JMSProducer to be able to send messages to a remote Active MQ server -->
<bean id="jms" class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="brokerURL" value="tcp://localhost:61610"/>
</bean>
```

The client will not use the Camel Maven Plugin so the Spring XML has been placed in *src/main/resources* to not conflict with the server configs.

camelContext	The Camel context is defined but does not contain any routes
template	The ProducerTemplate is used to place messages onto the JMS queue
jms bean	This initialises the Camel JMS component, allowing us to place messages onto the queue

And the CamelClient source code:

```

public static void main(final String[] args) throws Exception {
    System.out.println("Notice this client requires that the CamelServer is already
running!");

    ApplicationContext context = new
ClassPathXmlApplicationContext("camel-client.xml");

    // get the camel template for Spring template style sending of messages (=
producer)
    ProducerTemplate camelTemplate = context.getBean("camelTemplate",
ProducerTemplate.class);

    System.out.println("Invoking the multiply with 22");
    // as opposed to the CamelClientRemoting example we need to define the service
URI in this java code
    int response = (Integer)camelTemplate.sendBody("jms:queue:numbers",
ExchangePattern.InOut, 22);
    System.out.println("... the result is: " + response);

    System.exit(0);
}

```

The `ProducerTemplate` is retrieved from a Spring `ApplicationContext` and used to manually place a message on the "numbers" JMS queue. The `requestBody` method will use the exchange pattern `InOut`, which states that the call should be synchronous, and that the caller expects a response.

Before running the client be sure that both the ActiveMQ broker and the `CamelServer` are running.

Client Using Spring Remoting

Spring Remoting "eases the development of remote-enabled services". It does this by allowing you to invoke remote services through your regular Java interface, masking that a remote service is being called.

```

<!-- Camel proxy for a given service, in this case the JMS queue -->
<camel:proxy
  id="multiplierProxy"
  serviceInterface="org.apache.camel.example.server.Multiplier"
  serviceUrl="jms:queue:numbers"/>

```

The snippet above only illustrates the different and how Camel easily can setup and use Spring Remoting in one line configurations.

The **proxy** will create a proxy service bean for you to use to make the remote invocations. The **serviceInterface** property details which Java interface is to be implemented by the proxy. **serviceUrl** defines where messages sent to this proxy bean will be directed. Here we define the JMS endpoint with the "numbers" queue we used when working with Camel

template directly. The value of the **id** property is the name that will be the given to the bean when it is exposed through the Spring `ApplicationContext`. We will use this name to retrieve the service in our client. I have named the bean *multiplierProxy* simply to highlight that it is not the same multiplier bean as is being used by `CamelServer`. They are in completely independent contexts and have no knowledge of each other. As you are trying to mask the fact that remoting is being used in a real application you would generally not include proxy in the name.

And the Java client source code:

```
public static void main(final String[] args) {
    System.out.println("Notice this client requires that the CamelServer is already
running!");

    ApplicationContext context = new
ClassPathXmlApplicationContext("camel-client-remoting.xml");
    // just get the proxy to the service and we as the client can use the "proxy" as
it was
    // a local object we are invoking. Camel will under the covers do the remote
communication
    // to the remote ActiveMQ server and fetch the response.
    Multiplier multiplier = context.getBean("multiplierProxy", Multiplier.class);

    System.out.println("Invoking the multiply with 33");
    int response = multiplier.multiply(33);
    System.out.println("... the result is: " + response);

    System.exit(0);
}
```

Again, the client is similar to the original client, but with some important differences.

1. The Spring context is created with the new *camel-client-remoting.xml*
2. We retrieve the proxy bean instead of a `ProducerTemplate`. In a non-trivial example you would have the bean injected as in the standard Spring manner.
3. The multiply method is then called directly. In the client we are now working to an interface. There is no mention of Camel or JMS inside our Java code.

Client Using Message Endpoint EIP Pattern

This client uses the Message Endpoint EIP pattern to hide the complexity to communicate to the Server. The Client uses the same simple API to get hold of the endpoint, create an exchange that holds the message, set the payload and create a producer that does the send and receive. All done using the same neutral Camel API for **all** the components in Camel. So if the

communication was socket TCP based you just get hold of a different endpoint and all the java code stays the same. That is really powerful.

Okay enough talk, show me the code!

```
public static void main(final String[] args) throws Exception {
    System.out.println("Notice this client requires that the CamelServer is already
running!");

    ApplicationContext context = new
ClassPathXmlApplicationContext("camel-client.xml");
    CamelContext camel = context.getBean("camel-client", CamelContext.class);

    // get the endpoint from the camel context
    Endpoint endpoint = camel.getEndpoint("jms:queue:numbers");

    // create the exchange used for the communication
    // we use the in out pattern for a synchronized exchange where we expect a
response
    Exchange exchange = endpoint.createExchange(ExchangePattern.InOut);
    // set the input on the in body
    // must you correct type to match the expected type of an Integer object
exchange.getIn().setBody(11);

    // to send the exchange we need an producer to do it for us
    Producer producer = endpoint.createProducer();
    // start the producer so it can operate
    producer.start();

    // let the producer process the exchange where it does all the work in this
oneline of code
    System.out.println("Invoking the multiply with 11");
    producer.process(exchange);

    // get the response from the out body and cast it to an integer
    int response = exchange.getOut().getBody(Integer.class);
    System.out.println("... the result is: " + response);

    // stop and exit the client
    producer.stop();
    System.exit(0);
}
```

Switching to a different component is just a matter of using the correct endpoint. So if we had defined a TCP endpoint as: "mina:tcp://localhost:61610" then its just a matter of getting hold of this endpoint instead of the JMS and all the rest of the java code is exactly the same.

Run the Clients

The Clients is started using their main class respectively.

- as a standard java main application - just start their main class
- using maven `java:exec`

In this sample we start the clients using maven:

```
mvn compile exec:java -PCamelClient
```

```
mvn compile exec:java -PCamelClientRemoting
```

```
mvn compile exec:java -PCamelClientEndpoint
```

Also see the Maven `pom.xml` file how the profiles for the clients is defined.

USING THE CAMEL MAVEN PLUGIN

The Camel Maven Plugin allows you to run your Camel routes directly from Maven. This negates the need to create a host application, as we did with Camel server, simply to start up the container. This can be very useful during development to get Camel routes running quickly.

Listing 1. `pom.xml`

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

All that is required is a new plugin definition in your Maven POM. As we have already placed our Camel config in the default location (`camel-server.xml` has been placed in `META-INF/spring/`) we do not need to tell the plugin where the route definitions are located. Simply run `mvn camel:run`.

USING CAMEL JMX

Camel has extensive support for JMX and allows us to inspect the Camel Server at runtime. As we have enabled the `JMXAgent` in our tutorial we can fire up the `jconsole` and connect to the following service URI:

`service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi/camel`. Notice that Camel will log at INFO level the JMX Connector URI:

```
...
DefaultInstrumentationAgent INFO JMX connector thread started on
service:jmx:rmi:///jndi/rmi://claus-acer:1099/jmxrmi/camel
...
```


In the screenshot below we can see the route and its performance metrics:

The screenshot shows the J2SE 5.0 Monitoring & Management Console. The title bar reads "J2SE 5.0 Monitoring & Management Console: service:jmx:rmi:///jndi/rmi://claus-acer:1099/jmxrmi/camel". The "MBeans" tab is selected, showing a tree view on the left and a table of attributes on the right. The tree view shows a hierarchy: JMImplementation > org.apache.camel > claus-acer/camel > [jms]queue%3anumbers@646039348 > route > [jms]queue%3anumbers@646039348. The table on the right displays the following data:

Name	Value
Description	EventDrivenConsumerRoute[Endpoint[jms:queue:...
EndpointUri	jms:queue:numbers
FirstExchangeCompletionTime	Wed Jun 25 07:00:56 CEST 2008
FirstExchangeFailureTime	
LastExchangeCompletionTime	Wed Jun 25 07:01:36 CEST 2008
LastExchangeFailureTime	
MaxProcessingTimeMillis	14.827862
MeanProcessingTimeMillis	7.400568
MinProcessingTimeMillis	3.573918
NumCompleted	3
NumExchanges	3
NumFailed	0
TotalProcessingTimeMillis	22.201704

A "Refresh" button is located at the bottom right of the table.

SEE ALSO

- [Spring Remoting with JMS Example on Amin Abbaspour's Weblog](#)

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.

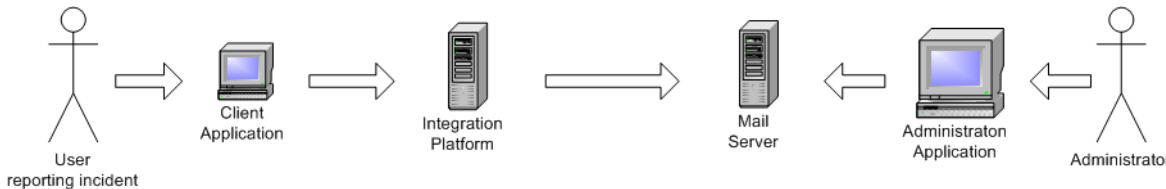
MOTIVATION FOR THIS TUTORIAL

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 USE-CASE

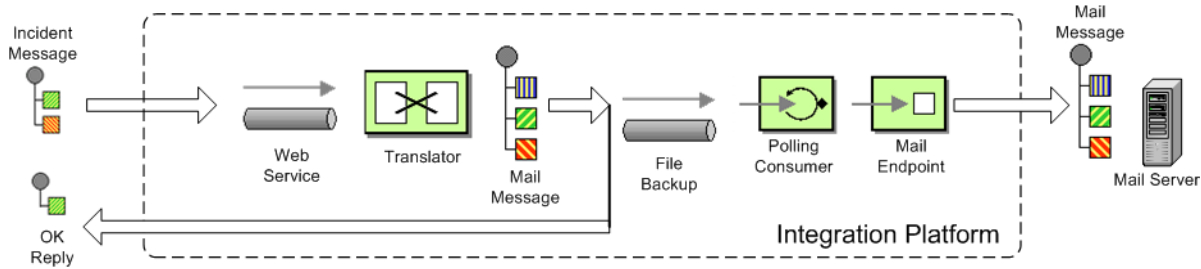
The goal is to allow staff to report incidents into a central administration. For that they use client software where they report the incident and submit it to the central administration. As this is an integration in a transition phase the administration should get these incidents by email whereas they 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

LINKS

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



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.

PART 1

PREREQUISITES

This tutorial uses the following frameworks:

- Maven 2.0.9
- Apache Camel 1.4.0
- Apache CXF 2.1.1
- Spring 2.5.5

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. doesn'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>2.1.1</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>${cxf-version}</version>
</dependency>
<dependency>
  <groupId>org.apache.cxf</groupId>
  <artifactId>cxf-rt-transport-http</artifactId>
  <version>${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 integrate 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 1.5 so we must add this to the pom.xml

```

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

```

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>
        </execution>
    </executions>
    <goals>
        <goal>wsdl2java</goal>
    </goals>
</plugin>

```

```

                </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 from 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-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 -->
  <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 in a web container, so we add jetty to our `pom.xml` so we can run `mvn jetty:run`:

```

<properties>
    ...
    <jetty-version>6.1.1</jetty-version>
</properties>

<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>
    </plugins>
</build>

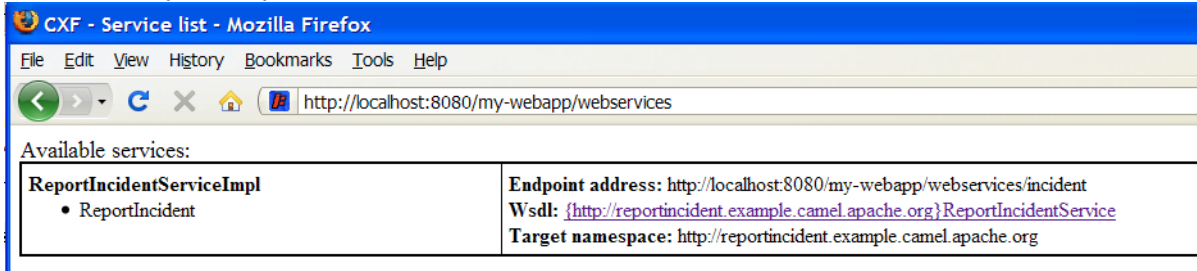
```

Notice: We use Jetty v6.1.1 as newer versions has troubles on my laptop. Feel free to try a newer version on your system, but v6.1.1 works flawless.

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 `/webservic*/` we should hit this URL to get the attention of CXF: `http://localhost:8080/`

camel-example-reportincident/webservices.

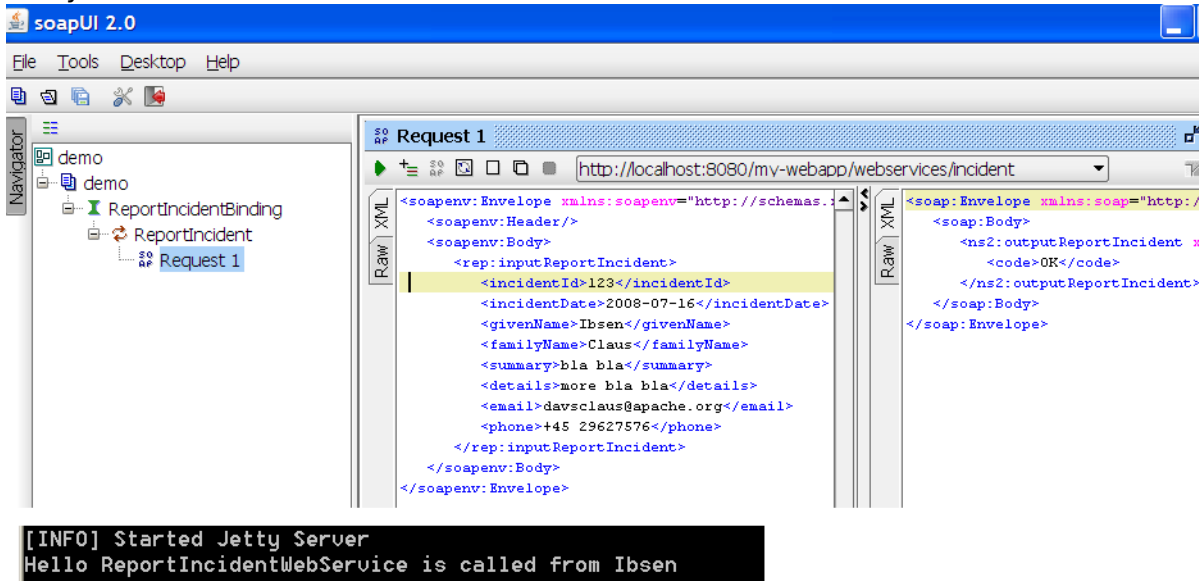


Â

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.



Â

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.

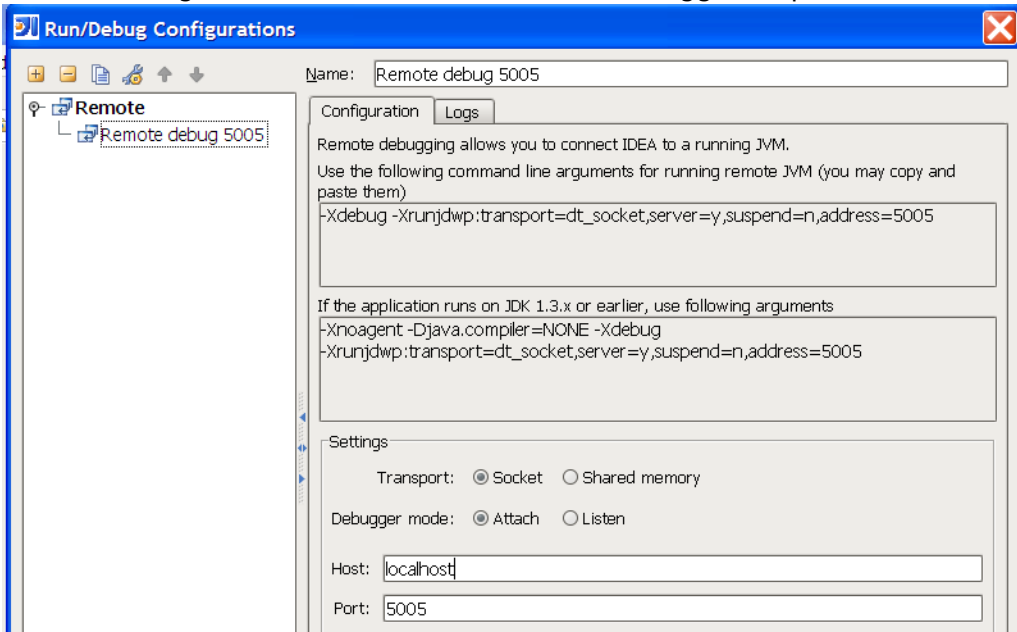
Se we set the MAVEN_OPTS environment to start in debug mode and listen on port 5005.

```
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:



À

Then we set a breakpoint in our code ReportIncidentEndpoint and hit the SoapUI once again and we are broken 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 testReportIncident() 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", "0K", 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-transport-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](#)

Name	Size	Creator	Creation Date	Comment	Â
• ZIP Archive tutorial_reportincident_part-one.zi...	14 kB	Claus Ibsen	Jul 17, 2008 23:34	Â	◦ ◦

LINKS

- [Introduction](#)
- [Part 1](#)
- [Part 2](#)
- [Part 3](#)
- [Part 4](#)
- [Part 5](#)

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.

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 grab 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();
}
```



Synchronize IDE

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

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.

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 Camel's 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 practice in Camel to
configure
        // endpoints based on URI.
default
        // com.mycompany.part2 = the log category used. Will log at INFO level as
        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
```



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.

```
        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 too 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 practice 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 automatically 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:

```

do
    // create the file endpoint, we cast to FileEndpoint because then we can
    // 100% java setter 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.

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();
}
```

i 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/camel/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.

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** file 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 **template** 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, but its 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.

RESOURCES

Name	Size	Creator	Creation Date	Comment	Â
ZIP Archive part-two.zip	17 kB	Claus Ibsen	Jul 19, 2008 00:52	Â	<ul style="list-style-type: none"> o Properties o Remove

LINKS

- Introduction
- Part 1

- Part 2
- Part 3
- Part 4
- Part 5

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. 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.



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.



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 endpoint = 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 = endpoint.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

RESOURCES

Name	Size	Creator	Creation Date	Comment	Â
ZIP Archive part-three.zip	18 kB	Claus Ibsen	Jul 20, 2008 03:34	Â	<ul style="list-style-type: none"> ◦ Properties ◦ Remove

LINKS

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

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.

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.



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! 😊

Before we jump into it, we want to state that this tutorial is about **Developers not losing control**. In my humble experience one of the key fears of developers is that they are forced into a tool/framework where they lose 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 language, 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.

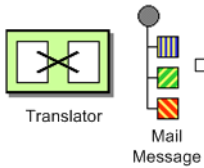
In the example above we have a very common routing, that can be distilled from pseudo verbs to actual code with:

- 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.

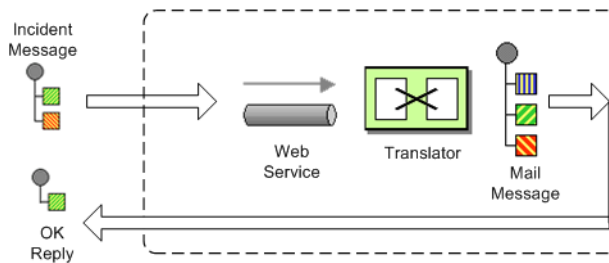
So we implement the logic in our webservice operation:

```
/**
 * 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.

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 {
```

i 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](#).

```
}
```

`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.

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);
}
```



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.

```
// 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 what 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. Phewww 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 Bean 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 Camel's 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:

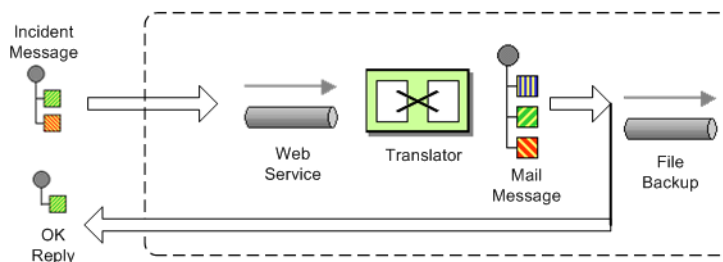
```

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");
    }
}

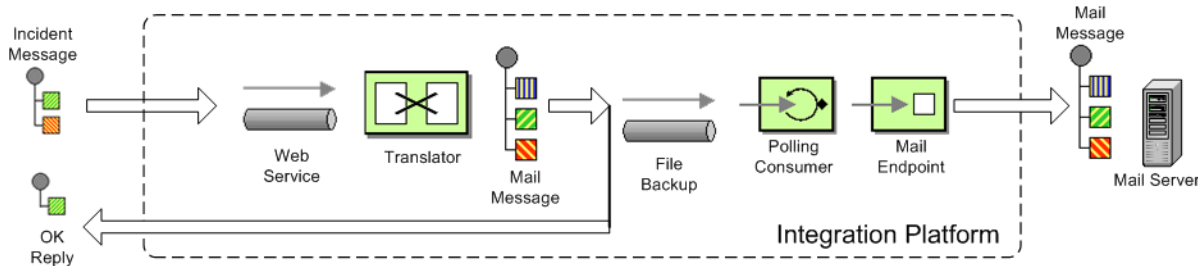
```

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.

RESOURCES

Name	Size	Creator	Creation Date	Comment	Â
ZIP Archive part-four.zip	11 kB	Claus Ibsen	Aug 25, 2008 07:24	Â	<ul style="list-style-type: none">◦ Properties◦ Remove

LINKS

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

BETTER JMS TRANSPORT FOR CXF WEBSERVICE USING APACHE CAMEL

Configuring JMS in Apache CXF before Version 2.1.3 is possible but not really easy or nice. This article shows how to use Apache Camel to provide a better JMS Transport for CXF.

Update: Since CXF 2.1.3 there is a new way of configuring JMS (Using the JMSConfigFeature). It makes JMS config for CXF as easy as with Camel. Using Camel for JMS is still a good idea if you want to use the rich feature of Camel for routing and other Integration Scenarios that CXF does not support.

You can find the original announcement for this Tutorial and some additional info on Christian Schneider's Blog

So how to connect Apache Camel and CXF

The best way to connect Camel and CXF is using the Camel transport for CXF. This is a camel module that registers with cxf as a new transport. It is quite easy to configure.

```
<bean class="org.apache.camel.component.cxf.transport.CamelTransportFactory">  
  <property name="bus" ref="cxf" />  
  <property name="camelContext" ref="camelContext" />  
  <property name="transportIds">
```

```

<list>
  <value>http://cxf.apache.org/transports/camel</value>
</list>
</property>
</bean>

```

This bean registers with CXF and provides a new transport prefix camel:// that can be used in CXF address configurations. The bean references a bean cxf which will be already present in your config. The other reference is a camel context. We will later define this bean to provide the routing config.

How is JMS configured in Camel

In camel you need two things to configure JMS. A ConnectionFactory and a JMSComponent. As ConnectionFactory you can simply set up the normal Factory your JMS provider offers or bind a JNDI ConnectionFactory. In this example we use the ConnectionFactory provided by ActiveMQ.

```

<bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

```

Then we set up the JMSComponent. It offers a new transport prefix to camel that we simply call jms. If we need several JMSComponents we can differentiate them by their name.

```

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory" ref="jmsConnectionFactory" />
  <property name="useMessageIDAsCorrelationID" value="true" />
</bean>

```

You can find more details about the JMSComponent at the Camel Wiki. For example you find the complete configuration options and a JNDI sample there.

Setting up the CXF client

We will configure a simple CXF webservice client. It will use stub code generated from a wsdl. The webservice client will be configured to use JMS directly. You can also use a direct: Endpoint and do the routing to JMS in the Camel Context.

```

<client id="CustomerService" xmlns="http://cxf.apache.org/jaxws"
xmlns:customer="http://customerservice.example.com/"

```

```
serviceName="customer:CustomerServiceService"
endpointName="customer:CustomerServiceEndpoint"
address="camel:jms:queue:CustomerService"
serviceClass="com.example.customerservice.CustomerService">
</client>
```

We explicitly configure `serviceName` and `endpointName` so they are not read from the `wsdl`. The names we use are arbitrary and have no further function but we set them to look nice. The `serviceClass` points to the service interface that was generated from the `wsdl`. Now the important thing is `address`. Here we tell `cxfr` to use the camel transport, use the `JmsComponent` who registered the prefix "jms" and use the queue "CustomerService".

Setting up the CamelContext

As we do not need additional routing an empty `CamelContext` bean will suffice.

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

Running the Example

- Download the example project here
- Follow the `readme.txt`

Conclusion

As you have seen in this example you can use Camel to connect services to JMS easily while being able to also use the rich integration features of Apache Camel.

TUTORIAL USING AXIS 1.4 WITH APACHE CAMEL

- Tutorial using Axis 1.4 with Apache Camel
- Prerequisites
- Distribution
- Introduction
- Setting up the project to run Axis
- Maven 2
- `wsdl`
- Configuring Axis



Removed from distribution

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

- Running the Example
- Integrating Spring
- Using Spring
- Integrating Camel
- CamelContext
- Store a file backup
- Running the example
- Unit Testing
- Smarter Unit Testing with Spring
- Unit Test calling Webservice
- Annotations
- The End
- See Also

Prerequisites

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

Distribution

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

Introduction

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

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

This tutorial uses the following frameworks:

- Maven 2.0.9
- Apache Camel 1.5.0

- Apache Axis 1.4
- Spring 2.5.5

Setting up the project to run Axis

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

Maven 2

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

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

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

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

<dependency>
  <groupId>axis</groupId>
  <artifactId>axis-wsd14j</artifactId>
  <version>1.5.1</version>
</dependency>

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

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

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

```

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

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

```

wsdl

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

```

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

  <!-- Type definitions for input- and output parameters for webservice -->
  <wsdl:types>
    <xs:schema targetNamespace="http://reportincident.example.camel.apache.org">
      <xs:element name="inputReportIncident">
        <xs:complexType>
          <xs:sequence>
            <xs:element type="xs:string"
name="incidentId"/>

```



```

name="incidentDate"/>
name="givenName"/>
name="familyName"/>
name="summary"/>
name="details"/>
name="email"/>
name="phone"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="outputReportIncident">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:string"
name="code"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
</wsdl:types>

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

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

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

```

```

                <soap:body parts="parameters" use="literal"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body parts="parameters" use="literal"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

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

```

Configuring Axis

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

```

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

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

```

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

```
mvn axistools:wSDL2java
```

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

```

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

```

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

```

package org.apache.camel.example.axis;

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

import java.rmi.RemoteException;

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

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

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

```

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

```

<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

```

```

<!-- global configuration -->
<globalConfiguration>
  <parameter name="sendXsiTypes" value="true"/>
  <parameter name="sendMultiRefs" value="true"/>
  <parameter name="sendXMLDeclaration" value="true"/>
  <parameter name="axis.sendMinimizedElements" value="true"/>
</globalConfiguration>
<handler name="URLMapper"
type="java:org.apache.axis.handlers.http.URLMapper"/>

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

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

```

```

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

```

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

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

```

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

```

Running the Example

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

```

mvn jetty:run

```

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

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

```

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

```

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

Integrating Spring

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

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

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

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

<listener>

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

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

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

</beans>
```

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

Using Spring

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

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

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

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

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

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

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

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

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

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

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

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

```

package org.apache.camel.example.axis;

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

import java.rmi.RemoteException;

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

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

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

```

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

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

```

public class AxisReportIncidentService extends ServletEndpointSupport implements
ReportIncidentService_PortType {

    private ReportIncidentService service;

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

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

```


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

Integrating Camel

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

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

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

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

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

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

CamelContext

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

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

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

  <camel:camelContext id="camel">
```



Camel does not require Spring

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

```
<!-- TODO: Here we can add Camel stuff -->
</camel:camelContext>

</beans>
```

Store a file backup

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

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

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

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

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

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

```

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

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

```

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

```

public class ReportIncidentService {

    private ProducerTemplate template;

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

```

And then let Spring handle the dependency inject as below:

```

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

```

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

```

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

    String data = parameters.getDetails();

    // store the data as a file
    String filename = parameters.getIncidentId() + ".txt";
    // send the data to the endpoint and the header contains what filename it
should be stored as
}

```

```

        template.sendBodyAndHeader("backup", data, "org.apache.camel.file.name",
filename);

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

```

The template in the code above uses 4 parameters:

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

Running the example

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

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

```

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

```

And there should be a file in the target subfolder.

```

dir target /b
123.txt

```

Unit Testing

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

```

<dependency>
  <groupId>junit</groupId>

```

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

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

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

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

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

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

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

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

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

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

Results :

Tests in error:
```

```
testIncident(org.apache.camel.example.axis.ReportIncidentServiceTest)
```

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

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

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

```
private CamelContext context;

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

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

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

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

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

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

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

```

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

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

```

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

```

Results :

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

```

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

```

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

```

Smarter Unit Testing with Spring

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

```

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

```

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

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

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

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

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

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

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

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

Unit Test calling WebService

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

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



```
<scope>test</scope>
</dependency>
```

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

```
public class AxisReportIncidentServiceTest extends TestCase {

    private Server server;

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

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

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

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

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

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

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

```

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

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

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

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

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

```

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

Annotations

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

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

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

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

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

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

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

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

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

</camel:camelContext>
```

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

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

To without the name:

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

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

The End

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

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

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

See Also

- Tutorials
- Examples

TUTORIAL ON USING CAMEL IN A WEB APPLICATION

Camel has been designed to work great with the Spring framework; so if you are already a Spring user you can think of Camel as just a framework for adding to your Spring XML files.

So you can follow the usual Spring approach to working with web applications; namely to add the standard Spring hook to load a **/WEB-INF/applicationContext.xml** file. In that file you can include your usual Camel XML configuration.

Step1: Edit your web.xml

To enable spring add a context loader listener to your **/WEB-INF/web.xml** file

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/
xml/ns/javaee/web-app_2_5.xsd"
  version="2.5">

  <listener>

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

</web-app>
```

This will cause Spring to boot up and look for the **/WEB-INF/applicationContext.xml** file.

Step 2: Create a /WEB-INF/applicationContext.xml file

Now you just need to create your Spring XML file and add your camel routes or configuration.

For example

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

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="seda:foo"/>
      <to uri="mock:results"/>
    </route>
  </camelContext>

</beans>
```

Then boot up your web application and you're good to go!

Hints and Tips

If you use Maven to build your application your directory tree will look like this...

```
src/main/webapp/WEB-INF
  web.xml
  applicationContext.xml
```

You should update your Maven pom.xml to enable WAR packaging/naming like this...

```
<project>
  ...
  <packaging>war</packaging>
  ...
```

```
<build>
  <finalName>[desired WAR file name]</finalName>
  ...
</build>
```

To enable more rapid development we highly recommend the `jetty:run` maven plugin.

Please refer to the help for more information on using `jetty:run` - but briefly if you add the following to your `pom.xml`

```
<build>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <configuration>
        <webAppConfig>
          <contextPath></contextPath>
        </webAppConfig>
        <scanIntervalSeconds>10</scanIntervalSeconds>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Then you can run your web application as follows

```
mvn jetty:run
```

Then Jetty will also monitor your `target/classes` directory and your `src/main/webapp` directory so that if you modify your spring XML, your `web.xml` or your java code the web application will be restarted, re-creating your Camel routes.

If your unit tests take a while to run, you could miss them out when running your web application via

```
mvn -Dtest=false jetty:run
```

TUTORIAL BUSINESS PARTNERS



Under Construction

This tutorial is a work in progress.

BACKGROUND AND INTRODUCTION

Business Background

So there's a company, which we'll call Acme. Acme sells widgets, in a fairly unusual way. Their customers are responsible for telling Acme what they purchased. The customer enters into their own systems (ERP or whatever) which widgets they bought from Acme. Then at some point, their systems emit a record of the sale which needs to go to Acme so Acme can bill them for it. Obviously, everyone wants this to be as automated as possible, so there needs to be integration between the customer's system and Acme.

Sadly, Acme's sales people are, technically speaking, doormats. They tell all their prospects, "you can send us the data in whatever format, using whatever protocols, whatever. You just can't change once it's up and running."

The result is pretty much what you'd expect. Taking a random sample of 3 customers:

- Customer 1: **XML over FTP**
- Customer 2: **CSV over HTTP**
- Customer 3: **Excel via e-mail**

Now on the Acme side, all this has to be converted to a canonical XML format and submitted to the Acme accounting system via JMS. Then the Acme accounting system does its stuff and sends an XML reply via JMS, with a summary of what it processed (e.g. 3 line items accepted, line item #2 in error, total invoice \$123.45). Finally, that data needs to be formatted into an e-mail, and sent to a contact at the customer in question ("Dear Joyce, we received an invoice on 1/2/08. We accepted 3 line items totaling \$123.45, though there was an error with line items #2 [invalid quantity ordered]. Thank you for your business. Love, Acme.").

So it turns out Camel can handle all this:

- Listen for HTTP, e-mail, and FTP files
- Grab attachments from the e-mail messages
- Convert XML, XLS, and CSV files to a canonical XML format
- read and write JMS messages
- route based on company ID
- format e-mails using Velocity templates
- send outgoing e-mail messages

Tutorial Background

This tutorial will cover all that, plus setting up tests along the way.

Before starting, you should be familiar with:

- Camel concepts including the CamelContext, Routes, Components and Endpoints, and Enterprise Integration Patterns
- Configuring Camel with the XML or Java DSL

You'll learn:

- How to set up a Maven build for a Camel project
- How to transform XML, CSV, and Excel data into a standard XML format with Camel
 - How to write POJOs (Plain Old Java Objects), Velocity templates, and XSLT stylesheets that are invoked by Camel routes for message transformation
- How to configure simple and complex Routes in Camel, using either the XML or the Java DSL format
- How to set up unit tests that load a Camel configuration and test Camel routes
- How to use Camel's Data Formats to automatically convert data between Java objects and XML, CSV files, etc.
- How to send and receive e-mail from Camel
- How to send and receive JMS messages from Camel
- How to use Enterprise Integration Patterns including Message Router and Pipes and Filters
 - How to use various languages to express content-based routing rules in Camel
- How to deal with Camel messages, headers, and attachments

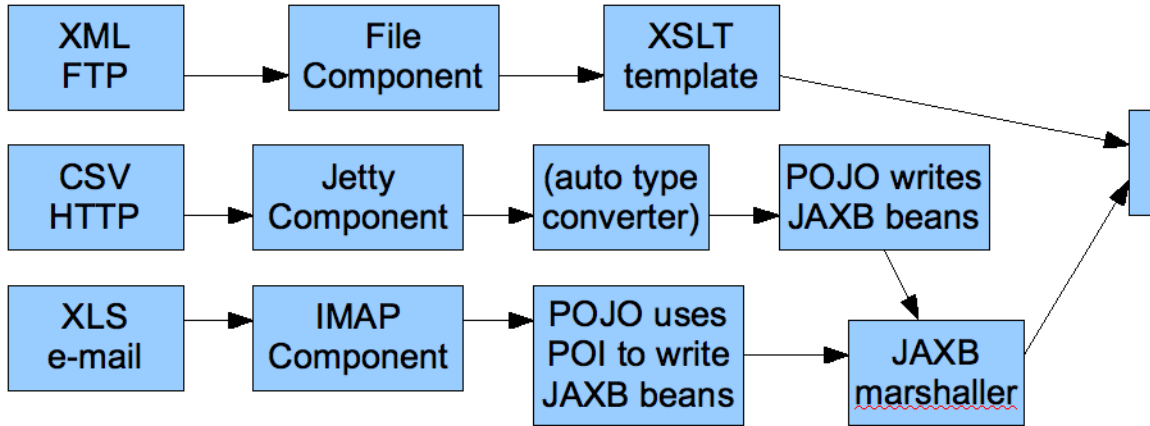
You may choose to treat this as a hands-on tutorial, and work through building the code and configuration files yourself. Each of the sections gives detailed descriptions of the steps that need to be taken to get the components and routes working in Camel, and takes you through tests to make sure they are working as expected.

But each section also links to working copies of the source and configuration files, so if you don't want the hands-on approach, you can simply review and/or download the finished files.

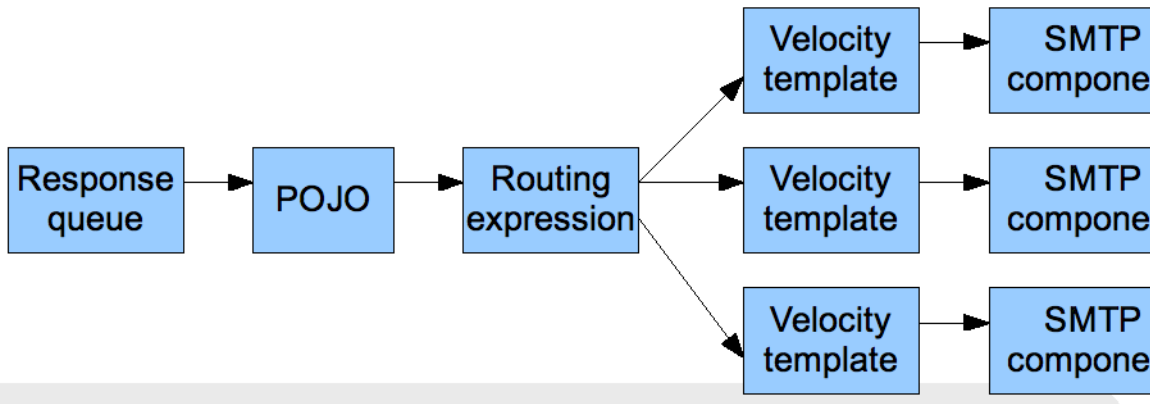
High-Level Diagram

Here's more or less what the integration process looks like.

First, the input from the customers to Acme:



And then, the output from Acme to the customers:



Tutorial Tasks

To get through this scenario, we're going to break it down into smaller pieces, implement and test those, and then try to assemble the big scenario and test that.

Here's what we'll try to accomplish:

1. Create a Maven build for the project
2. Get sample files for the customer Excel, CSV, and XML input
3. Get a sample file for the canonical XML format that Acme's accounting system uses
4. Create an XSD for the canonical XML format
5. Create JAXB POJOs corresponding to the canonical XSD

6. Create an XSLT stylesheet to convert the Customer 1 (XML over FTP) messages to the canonical format
7. Create a unit test to ensure that a simple Camel route invoking the XSLT stylesheet works
8. Create a POJO that converts a `List<List<String>>` to the above JAXB POJOs
 - Note that Camel can automatically convert CSV input to a List of Lists of Strings representing the rows and columns of the CSV, so we'll use this POJO to handle Customer 2 (CSV over HTTP)
9. Create a unit test to ensure that a simple Camel route invoking the CSV processing works
10. Create a POJO that converts a Customer 3 Excel file to the above JAXB POJOs (using POI to read Excel)
11. Create a unit test to ensure that a simple Camel route invoking the Excel processing works
12. Create a POJO that reads an input message, takes an attachment off the message, and replaces the body of the message with the attachment
 - This is assuming for Customer 3 (Excel over e-mail) that the e-mail contains a single Excel file as an attachment, and the actual e-mail body is throwaway
13. Build a set of Camel routes to handle the entire input (Customer -> Acme) side of the scenario.
14. Build unit tests for the Camel input.
15. **TODO:** Tasks for the output (Acme -> Customer) side of the scenario

LET'S GET STARTED!

Step 1: Initial Maven build

We'll use Maven for this project as there will eventually be quite a few dependencies and it's nice to have Maven handle them for us. You should have a current version of Maven (e.g. 2.0.9) installed.

You can start with a pretty empty project directory and a Maven POM file, or use a simple JAR archetype to create one.

Here's a sample POM. We've added a dependency on **camel-core**, and set the compile version to 1.5 (so we can use annotations):

Listing 1. pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
```

```

<modelVersion>4.0.0</modelVersion>
<groupId>org.apache.camel.tutorial</groupId>
<artifactId>business-partners</artifactId>
<version>1.0-SNAPSHOT</version>
<name>Camel Business Partners Tutorial</name>
<dependencies>
  <dependency>
    <artifactId>camel-core</artifactId>
    <groupId>org.apache.camel</groupId>
    <version>1.4.0</version>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>

```

Step 2: Get Sample Files

You can make up your own if you like, but here are the "off the shelf" ones. You can save yourself some time by downloading these to `src/test/resources` in your Maven project.

- Customer 1 (XML): `input-customer1.xml`
- Customer 2 (CSV): `input-customer2.csv`
- Customer 3 (Excel): `input-customer3.xls`
- Canonical Acme XML Request: `canonical-acme-request.xml`
- Canonical Acme XML Response: **TODO**

If you look at these files, you'll see that the different input formats use different field names and/or ordering, because of course the sales guys were totally OK with that. Sigh.

Step 3: XSD and JAXB Beans for the Canonical XML Format

Here's the sample of the canonical XML file:

```

<?xml version="1.0" encoding="UTF-8"?>
<invoice xmlns="http://activemq.apache.org/camel/tutorial/partners/invoice">
  <partner-id>2</partner-id>

```

```

<date-received>9/12/2008</date-received>
<line-item>
  <product-id>134</product-id>
  <description>A widget</description>
  <quantity>3</quantity>
  <item-price>10.45</item-price>
  <order-date>6/5/2008</order-date>
</line-item>
<!-- // more line-item elements here -->
<order-total>218.82</order-total>
</invoice>

```

If you're ambitious, you can write your own XSD (XML Schema) for files that look like this, and save it to `src/main/xsd`.

Solution: If not, you can download mine, and save that to save it to `src/main/xsd`.

Generating JAXB Beans

Down the road we'll want to deal with the XML as Java POJOs. We'll take a moment now to set up those XML binding POJOs. So we'll update the Maven POM to generate JAXB beans from the XSD file.

We need a dependency:

```

<dependency>
  <artifactId>camel-jaxb</artifactId>
  <groupId>org.apache.camel</groupId>
  <version>1.4.0</version>
</dependency>

```

And a plugin configured:

```

<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>jaxb2-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>xjc</goal>
      </goals>
    </execution>
  </executions>
</plugin>

```

That should do it (it automatically looks for XML Schemas in `src/main/xsd` to generate beans for). Run **mvn install** and it should emit the beans into `target/generated-sources/jaxb`. Your IDE should see them there, though

you may need to update the project to reflect the new settings in the Maven POM.

Step 4: Initial Work on Customer 1 Input (XML over FTP)

To get a start on Customer 1, we'll create an XSLT template to convert the Customer 1 sample file into the canonical XML format, write a small Camel route to test it, and build that into a unit test. If we get through this, we can be pretty sure that the XSLT template is valid and can be run safely in Camel.

Create an XSLT template

Start with the Customer 1 sample input. You want to create an XSLT template to generate XML like the canonical XML sample above – an invoice element with `line-item` elements (one per item in the original XML document). If you're especially clever, you can populate the current date and order total elements too.

Solution: My sample XSLT template isn't that smart, but it'll get you going if you don't want to write one of your own.

Create a unit test

Here's where we get to some meaty Camel work. We need to:

- Set up a unit test
- That loads a Camel configuration
- That has a route invoking our XSLT
- Where the test sends a message to the route
- And ensures that some XML comes out the end of the route

The easiest way to do this is to set up a Spring context that defines the Camel stuff, and then use a base unit test class from Spring that knows how to load a Spring context to run tests against. So, the procedure is:

Set Up a Skeletal Camel/Spring Unit Test

1. Add dependencies on Camel-Spring, and the Spring test JAR (which will automatically bring in JUnit 3.8.x) to your POM:

```
<dependency>
  <artifactId>camel-spring</artifactId>
  <groupId>org.apache.camel</groupId>
  <version>1.4.0</version>
</dependency>
```

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

2. Create a new unit test class in `src/test/java/your-package-here`, perhaps called `XMLInputTest.java`
3. Make the test extend Spring's `AbstractJUnit38SpringContextTests` class, so it can load a Spring context for the test
4. Create a Spring context configuration file in `src/test/resources`, perhaps called `XMLInputTest-context.xml`
5. In the unit test class, use the class-level `@ContextConfiguration` annotation to indicate that a Spring context should be loaded
 - By default, this looks for a Context configuration file called `TestClassName-context.xml` in a subdirectory corresponding to the package of the test class. For instance, if your test class was `org.apache.camel.tutorial.XMLInputTest`, it would look for `org/apache/camel/tutorial/XMLInputTest-context.xml`
 - To override this default, use the **locations** attribute on the `@ContextConfiguration` annotation to provide specific context file locations (starting each path with a `/` if you don't want it to be relative to the package directory). My solution does this so I can put the context file directly in `src/test/resources` instead of in a package directory under there.
6. Add a `CamelContext` instance variable to the test class, with the `@Autowired` annotation. That way Spring will automatically pull the `CamelContext` out of the Spring context and inject it into our test class.
7. Add a `ProducerTemplate` instance variable and a `setUp` method that instantiates it from the `CamelContext`. We'll use the `ProducerTemplate` later to send messages to the route.

```
protected ProducerTemplate<Exchange> template;

protected void setUp() throws Exception {
    super.setUp();
    template = camelContext.createProducerTemplate();
}
```

8. Put in an empty test method just for the moment (so when we run this we can see that "1 test succeeded")

9. Add the Spring `<beans>` element (including the Camel Namespace) with an empty `<camelContext>` element to the Spring context, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/
                           spring-beans-2.5.xsd
                           http://activemq.apache.org/camel/schema/spring
                           http://activemq.apache.org/camel/schema/
                           spring/camel-spring-1.4.0.xsd">
  <camelContext id="camel" xmlns="http://activemq.apache.org/camel/schema/
spring">
  </camelContext>
</beans>
```

Test it by running **mvn install** and make sure there are no build errors. So far it doesn't test much; just that your project and test and source files are all organized correctly, and the one empty test method completes successfully.

Solution: Your test class might look something like this:

- `src/test/java/org/apache/camel/tutorial/XMLInputTest.java`
- `src/test/resources/XMLInputTest-context.xml` (same as just above)

Flesh Out the Unit Test

So now we're going to write a Camel route that applies the XSLT to the sample Customer 1 input file, and makes sure that some XML output comes out:

1. Save the `input-customer1.xml` file to `src/test/resources`
2. Save your XSLT file (created in the previous step) to `src/main/resources`
3. Write a Camel Route, either right in the Spring XML, or using the Java DSL (in another class under `src/test/java` somewhere). This route should use the Pipes and Filters integration pattern to:
 1. Start from the endpoint `direct:start` (which lets the test conveniently pass messages into the route)
 2. Call the endpoint `xslt:YourXSLTFile.xml` (to transform the message with the specified XSLT template)
 3. Send the result to the endpoint `mock:finish` (which lets the test verify the route output)
4. Add a test method to the unit test class that:
 1. Get a reference to the Mock endpoint `mock:finish` using code like this:

```
MockEndpoint finish = MockEndpoint.resolve(camelContext,
"mock:finish");
```

2. Set the expectedMessageCount on that endpoint to 1
3. Get a reference to the Customer 1 input file, using code like this:

```
InputStream in =
XMLInputTest.class.getResourceAsStream("/input-partner1.xml");
assertNotNull(in);
```

4. Send that InputStream as a message to the direct:start endpoint, using code like this:

```
template.sendBody("direct:start", in);
```

Note that we can send the sample file body in several formats (File, InputStream, String, etc.) but in this case an InputStream is pretty convenient.

5. Ensure that the message made it through the route to the final endpoint, by testing all configured Mock endpoints like this:

```
MockEndpoint.assertIsSatisfied(camelContext);
```

6. If you like, inspect the final message body using some code like `finish.getExchanges().get(0).getIn().getBody()`.
 - If you do this, you'll need to know what format that body is – String, byte array, InputStream, etc.
5. Run your test with **mvn install** and make sure the build completes successfully.

Solution: Your finished test might look something like this:

- `src/test/java/org/apache/camel/tutorial/XMLInputTest.java`
- For XML Configuration:
 - `src/test/resources/XMLInputTest-context.xml`
- Or, for Java DSL Configuration:
 - `src/test/resources/XMLInputTest-dsl-context.xml`
 - `src/test/java/org/apache/camel/tutorial/routes/XMLInputTestRoute.java`



Test Base Class

Once your test class is working, you might want to extract things like the `@Autowired CamelContext`, the `ProducerTemplate`, and the `setUp` method to a custom base class that you extend with your other tests.

Step 5: Initial Work on Customer 2 Input (CSV over HTTP)

To get a start on Customer 2, we'll create a POJO to convert the Customer 2 sample CSV data into the JAXB POJOs representing the canonical XML format, write a small Camel route to test it, and build that into a unit test. If we get through this, we can be pretty sure that the CSV conversion and JAXB handling is valid and can be run safely in Camel.

Create a CSV-handling POJO

To begin with, CSV is a known data format in Camel. Camel can convert a CSV file to a List (representing rows in the CSV) of Lists (representing cells in the row) of Strings (the data for each cell). That means our POJO can just assume the data coming in is of type `List<List<String>>`, and we can declare a method with that as the argument.

Looking at the JAXB code in `target/generated-sources/jaxb`, it looks like an `Invoice` object represents the whole document, with a nested list of `LineItemType` objects for the line items. Therefore our POJO method will return an `Invoice` (a document in the canonical XML format).

So to implement the CSV-to-JAXB POJO, we need to do something like this:

1. Create a new class under `src/main/java`, perhaps called `CSVConverterBean`.
2. Add a method, with one argument of type `List<List<String>>` and the return type `Invoice`
 - You may annotate the argument with `@Body` to specifically designate it as the body of the incoming message
3. In the method, the logic should look roughly like this:
 1. Create a new `Invoice`, using the method on the generated `ObjectFactory` class
 2. Loop through all the rows in the incoming CSV (the outer `List`)
 3. Skip the first row, which contains headers (column names)
 4. For the other rows:
 1. Create a new `LineItemType` (using the `ObjectFactory` again)

2. Pick out all the cell values (the Strings in the inner List) and put them into the correct fields of the LineItemType
 - Not all of the values will actually go into the line item in this example
 - You may hardcode the column ordering based on the sample data file, or else try to read it dynamically from the headers in the first line
 - Note that you'll need to use a JAXB DatatypeFactory to create the XMLGregorianCalendar values that JAXB uses for the date fields in the XML – which probably means using a SimpleDateFormat to parse the date and setting that date on a GregorianCalendar
3. Add the line item to the invoice
5. Populate the partner ID, date of receipt, and order total on the Invoice
6. Throw any exceptions out of the method, so Camel knows something went wrong
7. Return the finished Invoice

Solution: Here's an example of what the CSVConverterBean might look like.

Create a unit test

Start with a simple test class and test Spring context like last time, perhaps based on the name CSVInputTest:

Listing 1. CSVInputTest.java

```
/**
 * A test class the ensure we can convert Partner 2 CSV input files to the
 * canonical XML output format, using JAXB POJOs.
 */
@ContextConfiguration(locations = "/CSVInputTest-context.xml")
public class CSVInputTest extends AbstractJUnit38SpringContextTests {
    @Autowired
    protected CamelContext camelContext;
    protected ProducerTemplate<Exchange> template;

    protected void setUp() throws Exception {
        super.setUp();
        template = camelContext.createProducerTemplate();
    }

    public void testCSVConversion() {
        // TODO
    }
}
```

```
}  
}
```

Listing 1. CSVInputTest-context.xml

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