SOA Made Easy with Open Source Apache Camel

Over the last several years, integration technology has been growing by leaps and bounds. The XML/REST/Web Services/SOA revolution has driven engineers and software firms to create an abundance of protocols, adaptors, transports, containers, standards, best practices...you name it.

The bits and bytes that are now available are undeniably sophisticated, diverse, and capable of almost anything, but many of the packages are built from the technology up and leave the job of how to use the capabilities effectively as an exercise for the reader.

Today, many readers have completed many such exercises. There is a wealth of experience and thousands of successful projects out there that have led to the definition of many infrastructure design patterns that help developers cut to the chase when it comes to integration. One set of design patterns that has gained traction in the industry is Hohpe and Woolf's Enterprise Integration Patterns. These patterns include a technology-agnostic vocabulary for describing large-scale integration solutions. Rather than focusing on the low-level programming, they take a top-down approach to developing an asynchronous, message-based architecture.

A consistent vocabulary is nice, but an easy-to-use framework for actually building the infrastructure would be even better.
That was exactly the thinking behind the open source Camel project at Apache. Now that a tried-and-true set of patterns is available, the obvious next step is to create an engine that can implement the patterns in the simplest way possible.

Camel is a code-first tool that allows developers to perform sophisticated large-scale integration without having to learn any vendor-specific or complex underlying technology. Camel is a POJO-based implementation of the Enterprise Integration Patterns using a declarative Java Domain Specific Language to connect to messaging systems and configure routing and mediation rules. The result is a framework that lets Java developers design and build a Service Oriented Architecture (SOA) without having to read pages and pages of specifications for technologies like JMS or JBI or deal with the lower-level details of Spring.

Apache Camel grew organically from code and ideas that were generated from other Apache projects particularly Apache ActiveMQ and Apache ServiceMix. Project members found that people wanted to create and use patterns from the Enterprise Integration Patterns book in many different scenarios. The Camel team set about to build such a framework for exactly this purpose.

Camel Overview
The first step in building Camel was to decouple the implementation of the patterns from the underlying plumbing. Some people want to use the patterns inside an enterprise service bus (ESB), some people want to use them inside a message broker, and other people want to use these patterns inside an application itself or to talk between messaging providers. Still other people want to use them inside a Web Services framework or some other communication platform. Rather than tie this routing code to a particular message broker or ESB, Camel extracts this code to be a standalone framework that can be

used in any project. Camel has a small footprint and can be reused anywhere, whether in a servlet, in the Web Services stack, inside a full ESB, or in a messaging application.

The primary advantage of Camel is that the development team doesn't have to work with containers just to connect systems. Many might consider working with containers to be a right of passage or a test of one's mettle, but to a growing number of teams these hurdles are an unnecessary barrier to entry. With Apache Camel, developers can get the job done with a minimum of extraneous tasks. Camel can, however, be deployed within a JBI container if other requirements warrant that, but it's not necessary.

To simplify the programming, Camel supports a domain-specific language in both Java and XML for the Enterprise Integration Patterns to be used in any Java IDE or from within spring XML (see Figure 1). This higher level of abstraction makes problem solving more efficient.

Camel reuses many Spring 2 features, such as declarative transactions, inversion of control configuration, and various utility classes for working with such things as JMS and JDBC and Java Persistence API (JPA). This raises the abstraction level to make things very simple, reducing the amount of XML one has to write, but still exposing the wire-level access if anyone needs to roll his sleeves up and get down and dirty.

Camel Examples
We're going explain different ways of configuring Apache Camel, first using the Java DSL (Domain Specific Language) and then using Spring XML configuration.

Java DSL Configuration
This example demonstrates a use case in which you want to archive messages from a JMS Queue into files in a directory structure. The first thing to do is to create a CamelContext object:

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

There's 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");
     // set up a listener on the file component
     from("file://test").process(new Processor() {

       public void process(Exchange e) {
         System.out.println("Received exchange: " + e.getIn());
       }
     });
   }
});
```

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

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

Next you must start the Camel context. If you're using Spring to configure the Camel context this is done automatically for you; although if you're 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.

In normal use, an external system would be firing messages or events directly into Camel through one if its components but we're going to use the CamelTemplate, which is a really easy way to test your configuration:

```
   CamelTemplate template = new CamelTemplate(context);
```

We can now send some test messages over JMS using the CamelTemplate:

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

From the CamelTemplate 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 file FileComponent will take messages from 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.

Spring XML Configuration
This example will use Spring XML configuration to transform files from a directory using XQuery and send the results to a JMS queue. It parsers some files from a directory, transforms them using XQuery then sends them to a message queue. To make it easy to look at the generated files, we also have another route that consumes from the JMS queue and writes them to an output directory.

Running the Example
To run the example we use the Camel Maven Plugin. For example, from the source or binary distribution the following should work:

```
cd examples/camel-example-spring-xquery
mvn camel:run
```

You should now see the generated files in the target/outputFiles directory, which are the transformed messages read from the JMS queue.

Code Walkthrough
What this does is boot up the Spring ApplicationContext defined in the file META-INF/spring /camelContext.xml on the classpath. This is a regular Spring XML document that uses the Camel XML configuration to configure a CamelContext.

Note that at the end of this XML example file we explicitly configure the ActiveMQ component with details on how to connect to the broker.

The main part of the Spring XML file is here:

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

  <!-- lets parse files, transform them with XQuery and send them to JMS -->
  <route>
    <from uri="file:src/data?noop=true"/>
    <to uri="xquery:myTransform.xquery"/>
    <to uri="jms:MyQueue"/>
  </route>

  <!-- now lets write messages from the queue to a directory -->
  <route>
    <from uri="jms:MyQueue"/>
    <to uri="file:target/outputFiles"/>
  </route>

</camelContext>

<!-- lets configure the default ActiveMQ broker URL -->
<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">
  <property name="connectionFactory">
    <bean class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="vm://localhost?broker.persistent=false"/>
    </bean>
  </property>
</bean>
```

This hopefully has given you a flavor of how easy it is to do enterprise integration using Apache Camel. For more information see the Web site at http://activemq.apache.org/camel/.

Resources
www.enterpriseintegrationpatterns.com/