Red Hat Fuse 7.8

Deploying into JBoss EAP

Deploying application packages into the JBoss Enterprise Application Platform (EAP) container

Last Updated: 2023-07-24

Red Hat Fuse 7.8 Deploying into JBoss EAP

Deploying application packages into the JBoss Enterprise Application Platform (EAP) container

Legal Notice

Abstract

The guide describes the options for deploying applications into a JBoss EAP container.

Table of Contents

CHAPTER 1. OVERVIEW OF JBOSS FUSE ON JBOSS EAP DEPLOYMENT

You can deploy Fuse applications on Red Hat JBoss Enterprise Application Platform (JBoss EAP), after installing the Fuse on EAP package into the JBoss EAP container.

This part describes the deployment model using the Camel on EAP subsystem. Apache Camel in Fuse enables you to select the container to run an integrated application.

NOTE

Red Hat JBoss EAP features a range of application deployment and configuration options to cater both administrators and developers. For more information about JBOSS EAP configuration and the deployment process, refer Red Hat JBoss EAP Configuration Guide.

1.1. SUPPORTED PRODUCT VERSIONS

To see the latest version of JBoss EAP that supports Fuse 7.8, refer to the Supported Configurations page.

1.2. CAMEL ON EAP SUBSYSTEM

The Camel on EAP subsystem integrates Apache Camel directly into the JBoss EAP container. This subsystem is available after you install the Fuse on EAP package into the JBoss EAP container. It offers many advantages for Camel deployment, including simplified deployment of Camel components and tighter integration with the underlying JBoss EAP container.

Red Hat recommends you to use the Camel on EAP Subsystem deployment model for deployment of Apache Camel applications on JBoss EAP.

CHAPTER 2. BUILDING YOUR APPLICATION ON JBOSS EAP

2.1. OVERVIEW

The following example demonstrates the use of **camel-cdi** component with Red Hat Fuse on EAP to integrate CDI beans with Camel routes.

In this example, a Camel route takes a message payload from a servlet **HTTP GET** request and passes it on to a direct endpoint. It then passes the payload onto a Camel CDI bean invocation to produce a message response and displays the output on the web browser page.

2.2. RUNNING THE PROJECT

Before running the project, ensure that your set up includes maven and the application server with Red Hat Fuse. Perform the following steps to run your project:

1. Start the application server in standalone mode:
   - For Linux: **${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml**
   - For Windows: **%JBOSS_HOME%\bin\standalone.bat -c standalone-full.xml**
2. Build and deploy the project: **mvn install -Pdeploy**
3. Now, browse to **http://localhost:8080/example-camel-cdi/?name=World** location. The following message **Hello World from 127.0.0.1** appears as an output on the web page. Also, you can view the Camel Route under the **MyRouteBuilder.java** class as:

```
from("direct:start").bean("helloBean");
```

The **bean** DSL makes Camel look for a bean named **helloBean** in the bean registry. Also, the bean is available to Camel due to the **SomeBean** class. By using the **@Named** annotation, the **camel-cdi** adds the bean to the Camel bean registry.

```
@Named("helloBean")

public class SomeBean {

    public String someMethod(String name) throws Exception {

        return String.format("Hello %s from %s", name, InetAddress.getLocalHost().getHostAddress());

    }

}
```

For more information, see **$ EAP_HOME/quickstarts/camel/camel-cdi** directory.

2.3. BOM FILE FOR JBOSS EAP

The purpose of a Maven Bill of Materials (BOM) file is to provide a curated set of Maven dependency versions that work well together, saving you from having to define versions individually for every Maven artifact.

The Fuse BOM for JBoss EAP offers the following advantages:

- Defines versions for Maven dependencies, so that you do not need to specify the version when you add a dependency to your POM.
- Defines a set of curated dependencies that are fully tested and supported for a specific version of Fuse.
- Simplifies upgrades of Fuse.

IMPORTANT

Only the set of dependencies defined by a Fuse BOM are supported by Red Hat.

To incorporate a BOM file into your Maven project, specify a **dependencyManagement** element in your project's **pom.xml** file (or, possibly, in a parent POM file), as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<project ...>
  ...
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <!-- configure the versions you want to use here -->
    <fuse.version>7.8.0.fuse-sb2-780038-redhat-00001</fuse.version>

  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.redhat-fuse</groupId>
        <artifactId>fuse-eap-bom</artifactId>
        <version>${fuse.version}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  ...
</project>
```

After specifying the BOM using the dependency management mechanism, it becomes possible to add Maven dependencies to your POM *without* specifying the version of the artifact. For example, to add a dependency for the **camel-velocity** component, you would add the following XML fragment to the **dependencies** element in your POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-velocity</artifactId>
  <scope>provided</scope>
</dependency>
```

Note how the **version** element is omitted from this dependency definition.

CHAPTER 3. FEATURES

This chapter provides the necessary information about Camel on EAP features.

Camel Context Definitions

Camel Contexts can be configured in standalone-camel.xml and domain.xml as part of the subsystem definition like this

```
<subsystem xmlns="urn:jboss:domain:camel:1.0">
  <camelContext id="system-context-1">
    <![CDATA[
    <route>
      <from uri="direct:start"/>
      <transform>
        <simple>Hello #{body}</simple>
      </transform>
    </route>
    ]]>
  </camelContext>
</subsystem>
```

Camel Context Deployments

You can deploy camel contexts to JBoss EAP with a **-camel-context.xml** suffix as:

- a standalone XML file
- a part of another supported deployment

A deployment may contain multiple **-camel-context.xml** files.

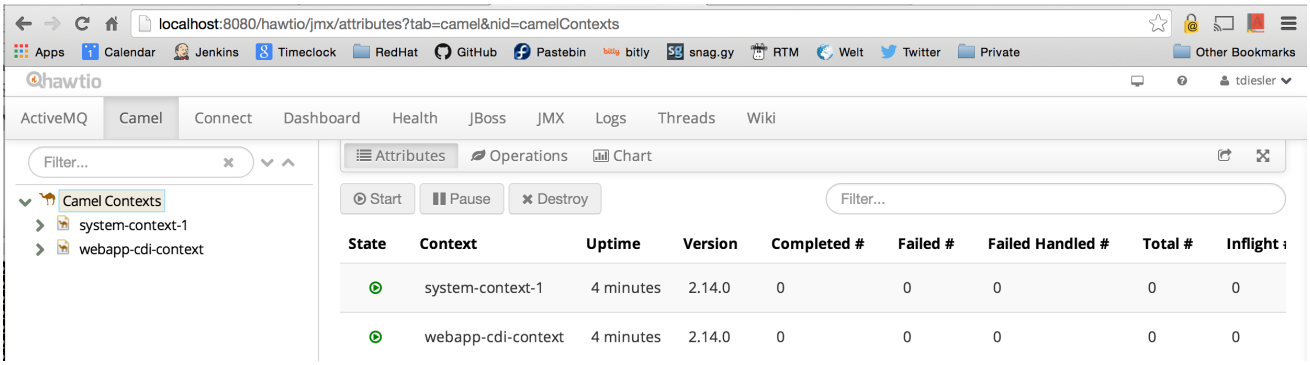
A deployed Camel context is CDI injectable like this

```
@Resource(lookup = "java:jboss/camel/context/mycontext")
CamelContext camelContext;
[discrete]
### Management Console
```

By default, access to management consoles is secured. Therefore, you need to setup a Management User first.

```
$ bin/add-user.sh

What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
```

The Hawt.io console should show the camel context from subsystem configuration.



Arquillian Test Support

The Camel on EAP test suite uses the WildFly Arquillian managed container. This can connect to an already running JBoss EAP instance or alternatively start up a standalone server instance when needed.

A number of test enrichers have been implemented that allow you to have these Camel on EAP specific types injected into your Arquillian test cases.

```
@ArquillianResource
CamelContextFactory contextFactory;

@ArquillianResource
CamelContextRegistry contextRegistry;
```

CHAPTER 4. CONFIGURATION

This chapter provides the necessary information about the Camel Subsystem and Deployment Configuration.

Camel Subsystem Configuration

The Camel Subsystem Configuration may contain static system routes. However, the system starts the route automatically.

```
<subsystem xmlns="urn:jboss:domain:camel:1.0">
  <camelContext id="system-context-1">
    <![CDATA[
    <route>
      <from uri="direct:start"/>
      <transform>
        <simple>Hello #{body}</simple>
      </transform>
    </route>
    ]]>
  </camelContext>
</subsystem>
```

Camel Deployment Configuration

If you want to modify the default configuration of your Camel deployment, you can edit either the **WEB-INF/jboss-all.xml** or **META-INF/jboss-all.xml** configuration file in your deployment.

Use a **<jboss-camel>** XML element within the **jboss-all.xml** file to control the camel configuration.

Disabling the Camel Subsystem

If you do not want to add the camel subsystem into your deployment, set the **enabled="false"** attribute on the **jboss-camel** XML element.

Example **jboss-all.xml** file:

```
<jboss umlns="urn:jboss:1.0">
  <jboss-camel xmlns="urn:jboss:jboss-camel:1.0" enabled="false"/>
</jboss>
```

Selecting Components

If you add nested **<component>** or **<component-module>** XML elements, then instead of adding the default list of Camel components to your deployment, only the specified components will be added to your deployment.

Example **jboss-all.xml** file:

```
<jboss umlns="urn:jboss:1.0">
  <jboss-camel xmlns="urn:jboss:jboss-camel:1.0">
    <component name="camel-ftp"/>
    <component-module name="org.apache.camel.component.rss"/>
  </jboss-camel>
</jboss>
```

CHAPTER 5. JAVAEE INTEGRATION

This chapter provides the necessary information about the integration points with JavaEE.

5.1. CDI

The Camel CDI component provides an auto-configuration for Apache Camel, using CDI as dependency injection framework. However, it is based on convention-over-configuration. It implements the standard camel bean integration so that you can use the Camel annotations easily in CDI beans.

For more information about CDI, refer to the cdi documentation.

The following example describes how you can consume and assosciate the Camel Context with a route.

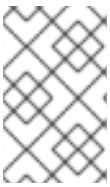
```
@Startup
@ApplicationScoped
@ContextName("cdi-context")
public class MyRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
     from("direct:start").transform(body().prepend("Hi"));
    }
}
```

```
@Inject
@ContextName("cdi-context")
private CamelContext camelctx;
```

5.1.1. Importing XML DSL configuration

Camel CDI integration enables you to import existing XML DSL files via the **@ImportResource** annotation:

```
@ImportResource("camel-context.xml")
class MyBean {
}
```



NOTE

The location of the imported file must be present on the deployment classpath. Placing the file into locations such as **WEB-INF** will not work. However, **WEB-INF/classes** will work fine.

5.2. EJB

Management support is provided through the ejb component which integrates with the EJB3 subsystem.

```
CamelContext camelctx = new DefaultCamelContext();
    camelctx.addRoutes(new RouteBuilder() {
        @Override
```

```
        public void configure() throws Exception {
            from("direct:start").to("ejb:java:module/HelloBean");
        }
    });
```

5.3. JAXB

JAXB support is provided through the Camel JAXB data format .

Camel supports unmarshalling XML data to JAXB annotated classes and marshalling from classes to XML. The following demonstrates a simple Camel route for marshalling and unmarshalling with the Camel JAXB data format class.

5.3.1. JAXB Annotated class

```
@XmlRootElement(name = "customer")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer implements Serializable {

    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

5.3.2. JAXB Class XML representation

```
<customer xmlns="http://org/wildfly/test/jaxb/model/Customer">
    <firstName>John</firstName>
    <lastName>Doe</lastName>
</customer>
```

5.3.3. Camel JAXB Unmarshalling

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext(getClass().getClassLoader());

final JaxbDataFormat jaxb = new JaxbDataFormat();
jaxb.setContextPath("org.wildfly.camel.test.jaxb.model");

camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .unmarshal(jaxb);
    }
});
camelctx.start();

ProducerTemplate producer = camelctx.createProducerTemplate();

// Send an XML representation of the customer to the direct:start endpoint
Customer customer = producer.requestBody("direct:start", readCustomerXml(), Customer.class);
Assert.assertEquals("John", customer.getFirstName());
Assert.assertEquals("Doe", customer.getLastName());
```

5.3.4. Camel JAXB Marshalling

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext();

final JaxbDataFormat jaxb = new JaxbDataFormat();
jaxb.setContextPath("org.wildfly.camel.test.jaxb.model");

camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .marshal(jaxb);
    }
});
camelctx.start();

ProducerTemplate producer = camelctx.createProducerTemplate();
Customer customer = new Customer("John", "Doe");
String customerXML = producer.requestBody("direct:start", customer, String.class);
Assert.assertEquals(readCustomerXml(), customerXML);
```

5.4. JAX-RS

JAX-RS support is provided by Camel CXF-RS.

5.4.1. CXF-RS Producer

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

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

    <cxf:rsClient id="cxfProducer"
                  address="http://localhost:8080/rest"
                  serviceClass="org.wildfly.camel.examples.cxf.jaxrs.GreetingService" />

    <camelContext id="cxfrs-camel-context" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:start" />
            <setHeader headerName="operationName">
                <simple>greet</simple>
            </setHeader>
            <setHeader headerName="CamelCxfRsUsingHttpAPI">
                <constant>false</constant>
            </setHeader>
            <to uri="cxfrs:bean:cxfProducer" />
        </route>
    </camelContext>
</beans>
```

5.4.2. CXF-RS Consumer

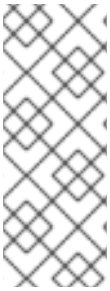
```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd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

    <cxf:rsServer id="cxfConsumer"
                  address="http://localhost:8080/rest"
                  serviceClass="org.wildfly.camel.examples.cxf.jaxrs.GreetingService" />

    <camelContext id="cxfrs-camel-context" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxfrs:bean:cxfConsumer" />
            <setBody>
                <constant>Hello world</constant>
            </setBody>
        </route>
    </camelContext>
</beans>
```

5.4.3. JAX-RS Consumer with the Camel REST DSL

The Camel REST DSL gives the capability to write Camel routes that act as JAX-RS consumers. The following RouteBuilder class demonstrates this.

```
@Startup
@ApplicationScoped
@ContextName("rest-camel-context")
public class RestConsumerRouteBuilder extends RouteBuilder {
  @Override
  public void configure() throws Exception {

    // Use the camel-undertow component to provide REST integration
    restConfiguration().component("undertow")
      .contextPath("/rest").port(8080).bindingMode(RestBindingMode.json);

    rest("/customer")
      // GET /rest/customer
      .get()
        .produces(MediaType.APPLICATION_JSON)
        .to("direct:getCustomers")
      // GET /rest/customer/1
      .get("/{id}")
        .produces(MediaType.APPLICATION_JSON)
        .to("direct:getCustomer")
      // POST /rest/customer
      .post()
        .type(Customer.class)
        .to("direct:createCustomer");
      // PUT /rest/customer
      .put()
        .type(Customer.class)
        .to("direct:updateCustomer");
      // DELETE /rest/customer/1
      .delete("/{id}")
        .to("direct:deleteCustomer");
  }
}
```

By setting the binding mode, Camel can marshal and unmarshal JSON data either by specifying a 'produces()' or 'type()' configuration step.



NOTE

- The REST DSL configuration starts with **restConfiguration().component("undertow")**.
- The Camel on EAP Subsystem only supports the camel-servlet and camel-undertow components for use with the REST DSL. However, it does not work if you configure the other components.

5.4.4. Security

Refer to the JAX-RS security section.

5.4.5. Quickstart examples in Fuse on EAP

A quickstart example is available in your Fuse on EAP installation at **quickstarts/camel/camel-cxf-jaxrs** directory.

5.5. JAX-WS

WebService support is provided through the CXF component which integrates with the JBoss EAP WebServices subsystem that also uses Apache CXF.

5.5.1. JAX-WS CXF Producer

The following code example uses CXF to consume a web service which has been deployed by the WildFly web services subsystem.

5.5.1.1. JAX-WS web service

The following simple web service has a simple 'greet' method which will concatenate two string arguments together and return them.

When the JBoss EAP web service subsystem detects classes containing JAX-WS annotations, it bootstraps a CXF endpoint. In this example the service endpoint will be located at http://hostname:port/context-root/greeting.

```
// Service interface
@WebService(name = "greeting")
public interface GreetingService {
    @WebMethod(operationName = "greet", action = "urn:greet")
    String greet(@WebParam(name = "message") String message, @WebParam(name = "name")
String name);
}

// Service implementation
public class GreetingServiceImpl implements GreetingService{
    public String greet(String message, String name) {
        return message + " " + name ;
    }
}
```

5.5.1.2. Camel route configuration

This RouteBuilder configures a CXF producer endpoint which will consume the 'greeting' web service defined above. CDI in conjunction with the camel-cdi component is used to bootstrap the RouteBuilder and CamelContext.

```
@Startup
@ApplicationScoped
@ContextName("cxf-camel-context")
public class CxfRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("direct:start")
        .to("cxf://http://localhost:8080/example-camel-cxf/greeting?serviceClass=" +
```

```
GreetingService.class.getName());
    }
}
```

The greeting web service 'greet' requires two parameters. These can be supplied to the above route by way of a **ProducerTemplate**. The web service method argument values are configured by constructing an object array which is passed as the exchange body.

```
String message = "Hello"
String name = "Kermit"

ProducerTemplate producer = camelContext.createProducerTemplate();
Object[] serviceParams = new Object[] {message, name};
String result = producer.requestBody("direct:start", serviceParams, String.class);
```

5.5.2. Camel CXF JAX-WS Consumer

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

    <cxf:cxfEndpoint id="cxfConsumer"
                     address="http://localhost:8080/webservices/greeting"
                     serviceClass="org.wildfly.camel.examples.cxf.jaxws.GreetingService" />

    <camelContext id="cxfws-camel-context" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="cxf:bean:cxfConsumer" />
            <to uri="log:ws" />
        </route>
    </camelContext>

</beans>
```

5.5.3. Security

Refer to the JAX-WS security section .

5.5.4. Quickstart examples in Fuse on EAP

A quickstart example is available in your Fuse on EAP installation at **quickstarts/camel/camel-cxf-jaxws** directory.

5.6. JMS

Messaging support is provided through the JMS component which integrates with the JBoss EAP Messaging (ActiveMQ Artemis) subsystem.

Integration with other JMS implementations is possible through configuration of vendor specific resource adapters, or if not available, by using the JBoss Generic JMS resource adapter.

5.6.1. JBoss EAP JMS configuration

You can configure the JBoss EAP messaging subsystem through the standard JBoss EAP XML configuration files. For example, standalone.xml or domain.xml.

For the examples, that follow you use the embedded ActiveMQ Artemis in memory instance. You first configure a new JMS queue on the messaging subsystem by adding the following XML configuration to the jms-destinations section.

```
<jms-queue name="WildFlyCamelQueue">
  <entry name="java:/jms/queue/WildFlyCamelQueue"/>
</jms-queue>
```

Alternatively you could use a CLI script to add the queue.

```
$ jms-queue add --queue-address=WildFlyCamelQueue --
entries=queue/WildFlyCamelQueue,java:/jms/queue/WildFlyCamelQueue
```

Also, you can create a **messaging-deployment** configuration within a custom jms.xml deployment descriptor. See section 'Deployment of -jms.xml files' within the JBoss EAP messaging subsystem documentation for more information.

5.6.2. Camel route configuration

The following JMS producer and consumer examples make use of JBoss EAP's embedded ActiveMQ Artemis server to publish and consume messages to and from destinations.

The examples also use CDI in conjunction with the camel-cdi component. JMS ConnectionFactory instances are injected into the Camel RouteBuilder through JNDI lookups.

5.6.2.1. JMS Producer

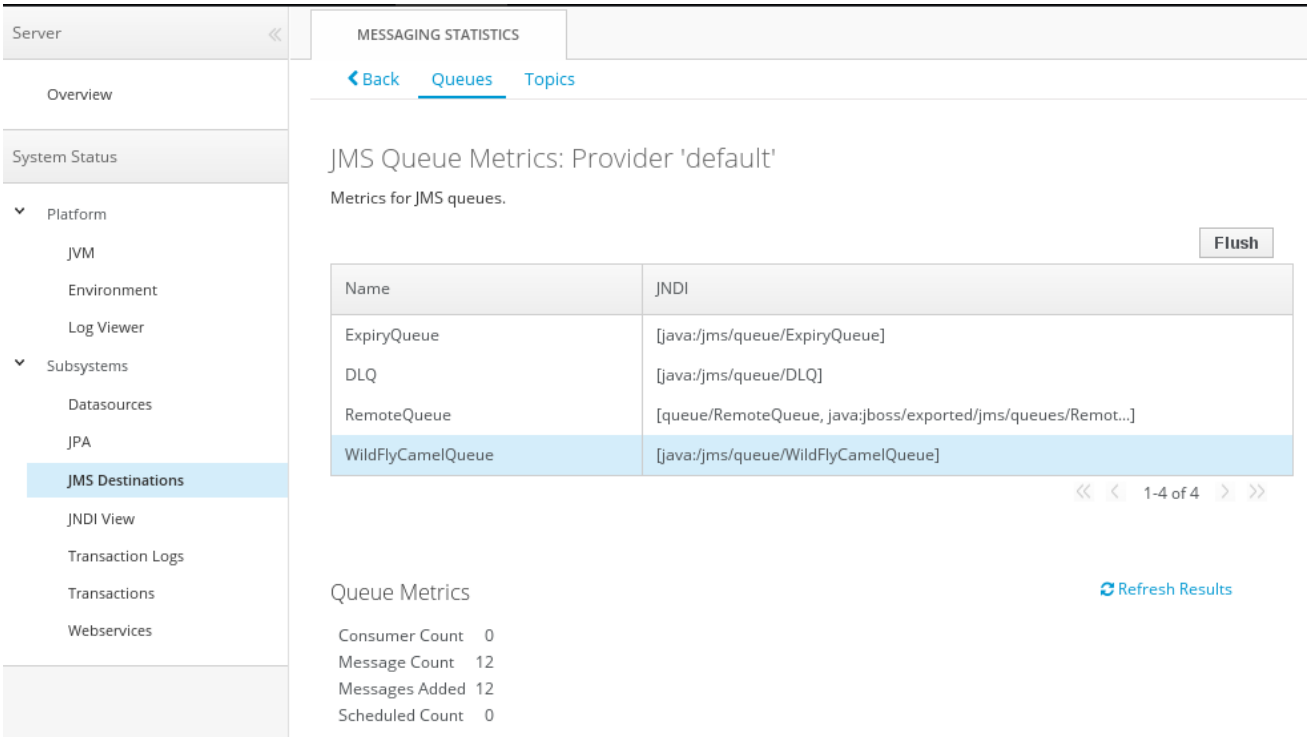
The **DefaultJMSConnectionFactory** connection factory is injected into the RouteBuilder from JNDI. Under the JBoss EAP XML configuration, you can find the connection factory, within the messaging subsystem.

Next a timer endpoint runs every 10 seconds to send an XML payload to the WildFlyCamelQueue destination that has been configured earlier.

```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JmsRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:jboss/DefaultJMSConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Override
  public void configure() throws Exception {
    JmsComponent component = new JmsComponent();
    component.setConnectionFactory(connectionFactory);
```

```
    getContext().addComponent("jms", component);

    from("timer://sendJMSMessage?fixedRate=true&period=10000")
    .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting>
</message>"))
    .to("jms:queue:WildFlyCamelQueue")
    .log("JMS Message sent");
  }
}
```

A log message will be output to the console each time a JMS message is added to the WildFlyCamelQueue destination. To verify that the messages really are being placed onto the queue, you can use the JBoss EAP administration console.



5.6.2.2. JMS Consumer

To consume JMS messages the Camel RouteBuilder implementation is similar to the producer example.

As before, the connection factory is discovered from JNDI, injected and set on the JMSComponent instance.

When the JMS endpoint consumes messages from the WildFlyCamelQueue destination, the content is logged to the console.

```
@Override
public void configure() throws Exception {
  JmsComponent component = new JmsComponent();
  component.setConnectionFactory(connectionFactory);

  getContext().addComponent("jms", component);
```

```
    from("jms:queue:WildFlyCamelQueue")
    .to("log:jms?showAll=true");
  }
```

5.6.2.3. JMS Transactions

To enable Camel JMS routes to participate in JMS transactions, some additional configuration is required. Since camel-jms is built around spring-jms, you need to configure some Spring classes to enable them to work with JBoss EAP's transaction manager and connection factory. The following code example demonstrates how to use CDI to configure a transactional JMS Camel route.

The camel-jms component requires a transaction manager of type **org.springframework.transaction.PlatformTransactionManager**. Therefore, you start by creating a bean extending **JtaTransactionManager**. Note that the bean is annotated with **@Named** to allow the bean to be registered within the Camel bean registry. Also note that the JBoss EAP transaction manager and user transaction instances are injected using CDI.

```
@Named("transactionManager")
public class CdiTransactionManager extends JtaTransactionManager {

  @Resource(mappedName = "java:/TransactionManager")
  private TransactionManager transactionManager;

  @Resource
  private UserTransaction userTransaction;

  @PostConstruct
  public void initTransactionManager() {
    setTransactionManager(transactionManager);
    setUserTransaction(userTransaction);
  }
}
```

Next, you need to declare the transaction policy that you want to use. Again, use the **@Named** annotation to make the bean available to Camel. The transaction manager is also injected so that a TransactionTemplate can be created with the desired transaction policy. PROPAGATION_REQUIRED in this instance.

```
@Named("PROPAGATION_REQUIRED")
public class CdiRequiredPolicy extends SpringTransactionPolicy {
  @Inject
  public CdiRequiredPolicy(CdiTransactionManager cdiTransactionManager) {
    super(new TransactionTemplate(cdiTransactionManager,
      new DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_REQUIRED)));
  }
}
```

Now you can configure our Camel RouteBuilder class and inject the dependencies needed for the Camel JMS component. The JBoss EAP XA connection factory is injected together with the transaction manager that has been configured earlier.

In this example RouteBuilder, whenever any messages are consumed from queue1, they are routed to another JMS queue named queue2. Messages consumed from queue2 result in JMS transaction being rolled back using the rollback() DSL method. This results in the original message being placed onto the dead letter queue(DLQ).

```
@Startup
@ApplicationScoped
@ContextName("jms-camel-context")
public class JMSRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:/JmsXA")
  private ConnectionFactory connectionFactory;

  @Inject
  CdiTransactionManager transactionManager;

  @Override
  public void configure() throws Exception {
    // Creates a JMS component which supports transactions
    JmsComponent jmsComponent = JmsComponent.jmsComponentTransacted(connectionFactory,
transactionManager);
    getContext().addComponent("jms", jmsComponent);

    from("jms:queue:queue1")
      .transacted("PROPAGATION_REQUIRED")
      .to("jms:queue:queue2");

    // Force the transaction to roll back. The message will end up on the Wildfly 'DLQ' message queue
    from("jms:queue:queue2")
      .to("log:end")
      .rollback();
  }
```

5.6.2.4. Remote JMS destinations

It is possible for one JBoss EAP instance to send messages to ActiveMQ Artemis destinations configured on another JBoss EAP instance through remote JNDI.

Some additional JBoss EAP configuration is required to achieve this. First an exported JMS queue is configured.

Only JNDI names bound in the **java:jboss/exported** namespace are considered as candidates for remote clients, so the queue is named appropriately.

NOTE

You must configure the queue on the JBoss EAP client application server *and* JBoss EAP remote server.

```
<jms-queue name="RemoteQueue">
  <entry name="java:jboss/exported/jms/queues/RemoteQueue"/>
</jms-queue>
```

Before the client can connect to the remote server, user access credentials need to be configured. On the remote server run the add user utility to create a new application user within the 'guest' group. This example has a user with the name 'admin' and a password of 'secret'.

The RouteBuilder implementation is different to the previous examples. Instead of injecting the connection factory, you need to configure an InitialContext and retrieve it from JNDI ourselves.

The **configureInitialContext** method creates this InitialContext. Notice that you need to set a provider URL which should reference your remote JBoss EAP instance host name and port number. This example uses the JBoss EAP JMS http-connector, but there are alternatives documented here.

Finally the route is configured to send an XML payload every 10 seconds to the remote destination configured earlier - 'RemoteQueue'.

```
@Override
public void configure() throws Exception {
  Context initialContext = configureInitialContext();
  ConnectionFactory connectionFactory = (ConnectionFactory)
initialContext.lookup("java:jms/RemoteConnectionFactory");

  JmsComponent component = new JmsComponent();
  component.setConnectionFactory(connectionFactory);

  getContext().addComponent("jms", component);

  from("timer://foo?fixedRate=true&period=10000")
  .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting>
</message>"))
  .to("jms:queue:RemoteQueue?username=admin&password=secret")
  .to("log:jms?showAll=true");
}

private Context configureInitialContext() throws NamingException {
  final Properties env = new Properties();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jboss.naming.remote.client.InitialContextFactory");
  env.put(Context.PROVIDER_URL, System.getProperty(Context.PROVIDER_URL, "http-
remoting://my-remote-host:8080"));
  env.put(Context.SECURITY_PRINCIPAL, System.getProperty("username", "admin"));
  env.put(Context.SECURITY_CREDENTIALS, System.getProperty("password", "secret"));
  return new InitialContext(env);
}
```

5.6.3. Security

Refer to the JMS security section.

5.6.4. Quickstart examples in Fuse on EAP

A quickstart example is available in your Fuse on EAP installation at **quickstarts/camel/camel-jms** directory.

5.7. JMX

You can provide management support through the JMX component which integrates with the JBoss EAP JMX subsystem.

```
CamelContext camelctx = contextFactory.createWildflyCamelContext(getClass().getClassLoader());
camelctx.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
```

```
        String host = InetAddress.getLocalHost().getHostName();
        from("jmx:platform?format=raw&objectDomain=org.apache.camel&key.context=" + host +
"/system-context-1&key.type=routes&key.name=\"route1\"" +
        "&monitorType=counter&observedAttribute=ExchangesTotal&granularityPeriod=500").
        to("direct:end");
    }
});
camelctx.start();

ConsumerTemplate consumer = camelctx.createConsumerTemplate();
MonitorNotification notifcation = consumer.receiveBody("direct:end", MonitorNotification.class);
Assert.assertEquals("ExchangesTotal", notifcation.getObservedAttribute());
```

5.8. JNDI

JNDI integration is provided by a JBoss EAP specific CamelContext as shown below:

```
InitialContext inictx = new InitialContext();
CamelContextFactory factory = inictx.lookup("java:jboss/camel/CamelContextFactory");
WildFlyCamelContext camelctx = factory.createCamelContext();
```

From a **WildFlyCamelContext** you can obtain a preconfigured Naming Context

```
Context context = camelctx.getNamingContext();
context.bind("helloBean", new HelloBean());
```

which can then be referenced from Camel routes.

```
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start").beanRef("helloBean");
    }
});
camelctx.start();
```

5.9. JPA

JPA integration is provided by the Camel JPA component. You can develop Camel JPA applications by providing a persistence.xml configuration file together with some JPA annotated classes.

5.9.1. Example persistence.xml

In the following example, you can use the JBoss EAP in-memory ExampleDS datasource which is configured within the JBoss EAP standalone.xml configuration file.

```
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

    <persistence-unit name="camel">
```

```
        <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-source>
        <class>org.wildfly.camel.test.jpa.model.Customer</class>
        <properties>
            <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>

</persistence>
```

5.9.2. Example JPA entitiy

```
@Entity
@Table(name = "customer")
public class Customer implements Serializable {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;

    public Customer() {
    }

    public Customer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Long getId() {
        return id;
    }

    public void setId(final Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

5.9.3. Camel JPA endpoint / route configuration

Having configured JPA, you can make use of CDI to inject an EntityManager and UserTransaction instance into your RouteBuilder class or test case:

```
@PersistenceContext
EntityManager em;

@Inject
UserTransaction userTransaction;
```

Now to configure the Camel routes and JPA endpoint:

```
WildFlyCamelContext camelctx = contextFactory.createCamelContext(getClass().getClassLoader());

EntityManagerFactory entityManagerFactory = em.getEntityManagerFactory();

// Configure a transaction manager
JtaTransactionManager transactionManager = new JtaTransactionManager();
transactionManager.setUserTransaction(userTransaction);
transactionManager.afterPropertiesSet();

// Configure the JPA endpoint to use the correct EntityManagerFactory and JtaTransactionManager
final JpaEndpoint jpaEndpoint = new JpaEndpoint();
jpaEndpoint.setCamelContext(camelctx);
jpaEndpoint.setEntityType(Customer.class);
jpaEndpoint.setEntityManagerFactory(entityManagerFactory);
jpaEndpoint.setTransactionManager(transactionManager);

camelctx.addRoutes(new RouteBuilder() {
@Override
public void configure() throws Exception {
    from("direct:start")
    .to(jpaEndpoint);
}
});

camelctx.start();
```

Finally, you can send a 'Customer' entity to the 'direct:start' endpoint and then query the ExampleDS datasource to verify that a record was saved.

```
Customer customer = new Customer("John", "Doe");
ProducerTemplate producer = camelctx.createProducerTemplate();
producer.sendBody("direct:start", customer);

// Query the in memory database customer table to verify that a record was saved
CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
CriteriaQuery<Long> query = criteriaBuilder.createQuery(Long.class);
query.select(criteriaBuilder.count(query.from(Customer.class)));

long recordCount = em.createQuery(query).getSingleResult();

Assert.assertEquals(1L, recordCount);
```

CHAPTER 6. CAMEL COMPONENTS

This chapter details information about supported camel components

6.1. CAMEL-ACTIVEMQ

Camel ActiveMQ integration is provided by the activemq component.

The component can be configured to work with an embedded or external broker. For Wildfly / EAP container managed connection pools and XA-Transaction support, the ActiveMQ Resource Adapter can be configured into the container configuration file.

6.1.1. JBoss EAP ActiveMQ resource adapter configuration

Download the ActiveMQ resource adapter rar file . The following steps outline how to configure the ActiveMQ resource adapter.

1. Stop your JBoss EAP instance.
2. Download the resource adapter and copy to the relevant JBoss EAP deployment directory. For standalone mode:

```
cp activemq-rar-5.11.1.rar ${JBOSS_HOME}/standalone/deployments/activemq-rar.rar
```

3. Configure the JBoss EAP resource adapters subsystem for the ActiveMQ adapter.

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:2.0">
    <resource-adapters>
        <resource-adapter id="activemq-rar.rar">
            <archive>
                activemq-rar.rar
            </archive>
            <transaction-support>XATransaction</transaction-support>
            <config-property name="UseInboundSession">
                false
            </config-property>
            <config-property name="Password">
                defaultPassword
            </config-property>
            <config-property name="UserName">
                defaultUser
            </config-property>
            <config-property name="ServerUrl">
                tcp://localhost:61616?jms.rmIdFromConnectionId=true
            </config-property>
            <connection-definitions>
                <connection-definition class-
name="org.apache.activemq.ra.ActiveMQManagedConnectionFactory" jndi-
name="java:/ActiveMQConnectionFactory" enabled="true" pool-name="ConnectionFactory">
                    <xa-pool>
                        <min-pool-size>1</min-pool-size>
                        <max-pool-size>20</max-pool-size>
                        <prefill>false</prefill>
                        <is-same-rm-override>false</is-same-rm-override>
```

```
            </xa-pool>
          </connection-definition>
        </connection-definitions>
        <admin-objects>
          <admin-object class-name="org.apache.activemq.command.ActiveMQQueue" jndi-
name="java:/queue/HELLOWORLDMDBQueue" use-java-context="true" pool-
name="HELLOWORLDMDBQueue">
            <config-property name="PhysicalName">
              HELLOWORLDMDBQueue
            </config-property>
          </admin-object>
          <admin-object class-name="org.apache.activemq.command.ActiveMQTopic" jndi-
name="java:/topic/HELLOWORLDMDBTopic" use-java-context="true" pool-
name="HELLOWORLDMDBTopic">
            <config-property name="PhysicalName">
              HELLOWORLDMDBTopic
            </config-property>
          </admin-object>
        </admin-objects>
      </resource-adapter>
    </resource-adapters>
  </subsystem>
```

If your resource adapter archive filename differs from activemq-rar.rar, you must change the content of the archive element in the preceding configuration to match the name of your archive file.

The values of the UserName and Password configuration properties must be chosen to match the credentials of a valid user in the external broker.

You might need to change the value of the ServerUrl configuration property to match the actual hostname and port exposed by the external broker.

4) Start JBoss EAP. If everything is configured correctly, you should see a message within the JBoss EAP server.log like.

```
13:16:08,412 INFO  [org.jboss.as.connector.deployment] (MSC service thread 1-5) JBAS010406:
Registered connection factory java:/AMQConnectionFactory`
```

6.1.2. Camel route configuration

The following ActiveMQ producer and consumer examples make use of the ActiveMQ embedded broker and the 'vm' transport (thus avoiding the need for an external ActiveMQ broker).

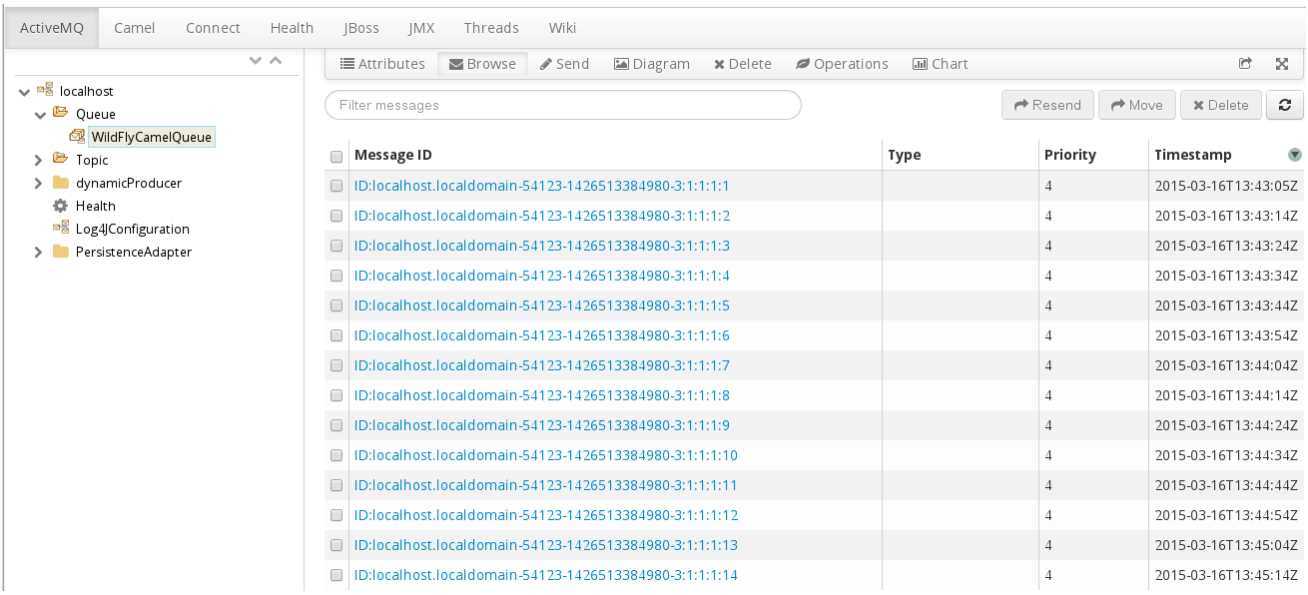
The examples use CDI in conjunction with the camel-cdi component. JMS ConnectionFactory instances are injected into the Camel RouteBuilder through JNDI lookups.

6.1.2.1. ActiveMQ Producer

```
@Startup
@ApplicationScoped
@ContextName("activemq-camel-context")
public class ActiveMQRouteBuilder extends RouteBuilder {

  @Override
  public void configure() throws Exception {
```

```
    from("timer://sendJMSMessage?fixedRate=true&period=10000")
    .transform(constant("<?xml version='1.0><message><greeting>hello world</greeting>
</message>"))
    .to("activemq:queue:WildFlyCamelQueue?brokerURL=vm://localhost")
    .log("JMS Message sent");
  }
}
```

A log message will be output to the console each time a message is added to the WildFlyCamelQueue destination. To verify that the messages really are being placed onto the queue, you can use the ../features/hawtio.md[Hawtio console,window=_blank] provided by the Camel on EAP subsystem.



6.1.2.2. ActiveMQ Consumer

To consume ActiveMQ messages the Camel RouteBuilder implementation is similar to the producer example.

When the ActiveMQ endpoint consumes messages from the WildFlyCamelQueue destination, the content is logged to the console.

```
@Override
public void configure() throws Exception {
  from("activemq:queue:WildFlyCamelQueue?brokerURL=vm://localhost")
  .to("log:jms?showAll=true");
}
```

6.1.2.3. ActiveMQ Transactions

6.1.2.3.1. ActiveMQ Resource Adapter Configuration

The ActiveMQ resource adapter is required to leverage XA transaction support, connection pooling etc.

The XML snippet below shows how the resource adapter is configured within the JBoss EAP server XML configuration. Notice that the **ServerURL** is set to use an embedded broker. The connection factory is bound to the JNDI name **java:/ActiveMQConnectionFactory**. This will be looked up in the RouteBuilder example that follows.

Finally, two queues are configured named 'queue1' and 'queue2'.

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:2.0">
  <resource-adapters>
    <resource-adapter id="activemq-rar.rar">
      ...
      <admin-objects>
        <admin-object class-name="org.apache.activemq.command.ActiveMQQueue" jndi-
name="java:/queue/queue1" use-java-context="true" pool-name="queue1pool">
          <config-property name="PhysicalName">queue1</config-property>
        </admin-object>
        <admin-object class-name="org.apache.activemq.command.ActiveMQQueue" jndi-
name="java:/queue/queue2" use-java-context="true" pool-name="queue2pool">
          <config-property name="PhysicalName">queue2</config-property>
        </admin-object>
      </admin-objects>
    </resource-adapter>
  </resource-adapters>
</subsystem>
```

6.1.2.4. Transaction Manager

The camel-activemq component requires a transaction manager of type **org.springframework.transaction.PlatformTransactionManager**. Therefore, you can start by creating a bean extending **JtaTransactionManager** which fulfills this requirement. Note that the bean is annotated with **@Named** to allow the bean to be registered within the Camel bean registry. Also note that the JBoss EAP transaction manager and user transaction instances are injected using CDI.

```
@Named("transactionManager")
public class CdiTransactionManager extends JtaTransactionManager {

  @Resource(mappedName = "java:/TransactionManager")
  private TransactionManager transactionManager;

  @Resource
  private UserTransaction userTransaction;

  @PostConstruct
  public void initTransactionManager() {
    setTransactionManager(transactionManager);
    setUserTransaction(userTransaction);
  }
}
```

6.1.2.5. Transaction Policy

Next you need to declare the transaction policy that you want to use. Again, use the **@Named** annotation to make the bean available to Camel. The transaction manager is also injected so that a **TransactionTemplate** can be created with the desired transaction policy. **PROPAGATION_REQUIRED** in this instance.

```
@Named("PROPAGATION_REQUIRED")
public class CdiRequiredPolicy extends SpringTransactionPolicy {
  @Inject
```

```
  public CdiRequiredPolicy(CdiTransactionManager cdiTransactionManager) {
    super(new TransactionTemplate(cdiTransactionManager,
      new DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_REQUIRED)));
  }
}
```

6.1.2.6. Route Builder

Now you can configure the Camel RouteBuilder class and inject the dependencies needed for the Camel ActiveMQ component. The ActiveMQ connection factory that you configured on the resource adapter configuration is injected together with the transaction manager you configured earlier.

In this example RouteBuilder, whenever any messages are consumed from queue1, they are routed to another JMS queue named queue2. Messages consumed from queue2 result in JMS transaction being rolled back using the rollback() DSL method. This results in the original message being placed onto the dead letter queue(DLQ).

```
@Startup
@ApplicationScoped
@ContextName("activemq-camel-context")
public class ActiveMQRouteBuilder extends RouteBuilder {

  @Resource(mappedName = "java:/ActiveMQConnectionFactory")
  private ConnectionFactory connectionFactory;

  @Inject
  private CdiTransactionManager transactionManager;

  @Override
  public void configure() throws Exception {
    ActiveMQComponent activeMQComponent = ActiveMQComponent.activeMQComponent();
    activeMQComponent.setTransacted(false);
    activeMQComponent.setConnectionFactory(connectionFactory);
    activeMQComponent.setTransactionManager(transactionManager);

    getContext().addComponent("activemq", activeMQComponent);

      errorHandler(deadLetterChannel("activemq:queue:ActiveMQ.DLQ")
      .useOriginalMessage()
      .maximumRedeliveries(0)
      .redeliveryDelay(1000));

    from("activemq:queue:queue1F")
      .transacted("PROPAGATION_REQUIRED")
      .to("activemq:queue:queue2");

    from("activemq:queue:queue2")
      .to("log:end")
      .rollback();
  }
}
```

6.1.3. Security

Refer to the JMS security section.

6.1.4. Code examples on GitHub

An example camel-activemq application is available on GitHub.

6.2. CAMEL-JMS

There are two supported ways of connecting camel-jms, camel-sjms and camel-sjms2 endpoints to a remote AMQ 7 broker.

1. Configuring a remote-connector with a pooled-connection-factory as described in the section called Configuring the Artemis Resource Adapter to Connect to Red Hat JBoss AMQ 7 in the JBoss EAP Configuring Messaging guide.

2. Configuring a remote-connector with connection-factory as described in Configure a remote-connector with connection-factory

The first option is the preferred method, because it provides connection pooling and XA transaction support.

For messaging scenarios that use durable subscribers, pooled-connection-factory is not supported by Fuse 7.8 on JBoss EAP due to constraints imposed by the JavaEE 7 specification. In these scenarios configuring a standard unpooled connection-factory is preferred.

Configure a remote-connector with connection-factory

1. Create an outbound-socket-binding pointing to the remote messaging server:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=messaging-remote-throughput:add(host=localhost, port=61616)
```

2. Create a remote-connector referencing the outbound-socket-binding created in step 1.

```
/subsystem=messaging-activemq/server=default/remote-connector=netty-remote-throughput:add(socket-binding=messaging-remote-throughput)
```

3. Create a connection-factory referencing the remote-connector created in step 2.

```
/subsystem=messaging-activemq/server=default/connection-factory=simple-remote-artemis-connection-factory:add(entries=[java:/jms/RemoteJms],connectors=[netty-remote-throughput])
```

6.2.1. Messaging brokers and clients

Abstract

Fuse 7.8 does not ship with a default internal messaging broker, but it is designed to interface with external JMS brokers.

Fuse 7.8 on JBoss EAP uses the resource adapters detailed in Configuring Messaging on JBoss EAP to access external messaging brokers.

See Supported Configurations for more information about the external brokers, JCA adapters and Camel component combinations that are available for messaging on Fuse 7.8 on JBoss EAP.

For more information about connecting to external brokers using Fuse on JBoss EAP using JMS, see Section 6.2, “camel-jms”.

camel-jms quickstart

A quickstart is provided to demonstrate the use of the camel-jms component with Fuse on JBoss EAP to produce and consume JMS messages.

In this quickstart a Camel route consumes files from **EAP_HOME/standalone/data/orders** and places their contents onto an in-memory ActiveMQ Artemis queue named **OrdersQueue**. Another Camel route then consumes the contents of **OrdersQueue** and sorts the orders into individual country directories within **EAP_HOME/standalone/data/orders/processed**.

CLI commands create and delete **OrdersQueue** CLI scripts take care of creating and removing the JMS **OrdersQueue** for you when the application is deployed and undeployed. These scripts are located within the **EAP_HOME/quickstarts/camel-jms/src/main/resources/cli** directory.

Prerequisites

To run this quickstart you must have a working version of Fuse 7.8

You must also follow the instructions in Using JBoss AMQ for remote JMS Communication to connect to an external AMQ 7 broker. You can then inject the connection factory as you would with the default connection factory.

```
@Resource(mappedName = "java:jboss/RemoteJmsXA")
ConnectionFactory connectionFactory;
```

Setup the quickstart

1. Start JBOSS EAP in standalone mode.
2. Navigate to **EAP_HOME/quickstarts/camel/camel-jms**
3. Enter **mvn clean install -Pdeploy`** to build and deploy the quickstart.
4. Browse to http://localhost:8080/example-camel-jms

You should see a page titled 'Orders Received'. As we send orders to the example application, a list of orders per country will be listed on this page.

Run the quickstart

There are some example order XML files within the **EAP_HOME/quickstarts/camel/camel-jms/src/main/resources** directory. Camel will choose a file at random every 5 seconds and will copy it into **EAP_HOME/standalone/data/orders** for processing.

The console will output messages detailing what happened to each of the orders. The output will look something like this.

```
JmsConsumer[OrdersQueue]) Sending order to the UK
JmsConsumer[OrdersQueue]) Sending order to another country
JmsConsumer[OrdersQueue]) Sending order to the US
```

When the files have been consumed, you can return to http://localhost:8080/example-camel-jms/orders. The count of received orders for each country should have been increased by 1.

All the processed orders will be split into the following destinations:

```
EAP_HOME/standalone/data/orders/processed/uk
EAP_HOME/standalone/data/orders/processed/us
EAP_HOME/standalone/data/orders/processed/other
```

Undeploy

To undeploy the example, navigate to **EAP_HOME/quickstarts/camel/camel-jms** run **mvn clean -Pdeploy**.

6.3. CAMEL-MAIL

Interaction with email is provided by the mail component.

By default, Camel will create its own mail session and use this to interact with your mail server. Since JBoss EAP already provides a mail subsystem with all of the relevant support for secure connections, username and password encryption etc, therefore, it is recommended to configure your mail sessions within the JBoss EAP configuration and use JNDI to wire them into your Camel endpoints.

6.3.1. JBoss EAP configuration

First you configure the JBoss EAP mail subsystem for the Mail server. The following example adds configuration for Google Mail IMAP and SMTP .

An additional mail-session is configured after the 'default' session.

```
<subsystem xmlns="urn:jboss:domain:mail:2.0">
    <mail-session name="default" jndi-name="java:jboss/mail/Default">
      <smtp-server outbound-socket-binding-ref="mail-smtp"/>
    </mail-session>

    <mail-session debug="true" name="gmail" jndi-name="java:jboss/mail/gmail">
      <smtp-server outbound-socket-binding-ref="mail-gmail-smtp" ssl="true" username="your-username-here" password="your-password-here"/>
      <imap-server outbound-socket-binding-ref="mail-gmail-imap" ssl="true" username="your-username-here" password="your-password-here"/>
    </mail-session>
</subsystem>
```

NOTE

You can configure **outbound-socket-binding-ref** values of 'mail-gmail-smtp' and 'mail-gmail-imap'.

The next step is to configure these socket bindings. You can add additional bindings to the **socket-binding-group** configuration as per the following.

```
<outbound-socket-binding name="mail-gmail-smtp">
  <remote-destination host="smtp.gmail.com" port="465"/>
</outbound-socket-binding>
```

```
<outbound-socket-binding name="mail-gmail-imap">
  <remote-destination host="imap.gmail.com" port="993"/>
</outbound-socket-binding>
```

This configures the mail session to connect to host smtp.gmail.com on port 465 and imap.gmail.com on port 993. If you're using a different mail host, then this detail will be different.

6.3.2. POP3 Configuration

If you need to configure POP3 sessions, the principles are the same as defined in the examples above.

```
<!-- Server configuration -->
<pop3-server outbound-socket-binding-ref="mail-pop3" ssl="true" username="your-username-here"
password="your-password-here"/>

<!-- Socket binding configuration -->
<outbound-socket-binding name="mail-gmail-imap">
  <remote-destination host="pop3.gmail.com" port="993"/>
</outbound-socket-binding>
```

6.3.3. Camel route configuration

6.3.3.1. Mail producer

This example uses the SMTPS protocol, together with CDI in conjunction with the camel-cdi component. The Java mail session that you configured within the JBoss EAP configuration is injected into a Camel RouteBuilder through JNDI.

6.3.3.1.1. Route builder SMTPS example

The GMail mail session is injected into a Producer class using the **@Resource** annotation with a reference to the **jndi-name** attribute that you previously configured. This allows you to reference the mail session on the camel-mail endpoint configuration.

```
public class MailSessionProducer {
    @Resource(lookup = "java:jboss/mail/greenmail")
    private Session mailSession;

    @Produces
    @Named
    public Session getMailSession() {
        return mailSession;
    }
}
```

```
public class MailRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .to("smtps://smtp.gmail.com?session=#mailSession");
    }
}
```

To send an email, you can create a ProducerTemplate and send an appropriate body together with the necessary email headers.

```
Map<String, Object> headers = new HashMap<String, Object>();
headers.put("To", "destination@test.com");
headers.put("From", "sender@example.com");
headers.put("Subject", "Camel on Wildfly rocks");

String body = "Hi,\n\nCamel on Wildfly rocks!.";

ProducerTemplate template = camelContext.createProducerTemplate();
template.sendBodyAndHeaders("direct:start", body, headers);
```

6.3.3.2. Mail consumer

To receive email you can use an IMAP MailEndpoint. The Camel route configuration looks like the following.

```
public void configure() throws Exception {
  from("imaps://imap.gmail.com?session=#mailSession")
  .to("log:email");
}
```

6.3.4. Security

6.3.4.1. SSL configuration

JBoss EAP can be configured to manage Java mail sessions and their associated transports using SSL / TLS. When configuring mail sessions you can configure SSL or TLS on server types:

- smtp-server
- imap-server
- pop-server

By setting attributes **ssl="true"** or **tls="true"**.

6.3.4.2. Securing passwords

It is recommended to not use clear text for passwords within configuration files. You can mask sensitive data using the WildFly Vault.

6.3.4.3. Camel security

Camel endpoint security documentation can be found on the mail component guide. Camel also has a security summary page.

6.3.5. Code examples on GitHub

An example camel-mail application is available on GitHub for you to try out sending / receiving email.

6.4. CAMEL-REST

The rest component allows you to define REST endpoints using the Rest DSL and plugin to other Camel components as the REST transport.

NOTE

The Camel on EAP Subsystem only supports the camel-servlet and camel-undertow components for use with the REST DSL. However, the subsystem does not work, If you attempt to configure other components.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        restConfiguration().component("servlet").contextPath("camel/rest").port(8080);
        rest("/hello").get("/{name}").to("direct:hello");
        from("direct:hello").transform(simple("Hello ${header.name}"));
    }
});
```

6.5. CAMEL-REST-SWAGGER

The rest-swagger component can configure REST producers from a Swagger document and delegate to a component implementing the **RestProducerFactory** interface such as:

- camel-http4
- camel-undertow

6.6. CAMEL-SQL

The SQL component allows you to work with databases using JDBC queries. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("sql:select name from information_schema.users?
dataSource=java:jboss/datasources/ExampleDS")
        .to("direct:end");
    }
});
```

NOTE

The JNDI datasource lookup shown above works only when configuring a **DefaultCamelContext**. See below for **CdiCamelContext** and **SpringCamelContext** examples.

When used in conjunction with the camel-cdi component, Java EE annotations can make a datasource available to Camel. This example uses the **@Named** annotation so that Camel can discover the desired datasource.

```
public class DatasourceProducer {
    @Resource(lookup = "java:jboss/datasources/ExampleDS")
    DataSource dataSource;

    @Produces
    @Named("wildFlyExampleDS")
    public DataSource getDataSource() {
        return dataSource;
    }
}
```

Now the datasource can be referenced through the **dataSource** parameter on the camel-sql endpoint configuration.

```
@ApplicationScoped
@ContextName("camel-sql-cdi-context")
@Startup
public class CdiRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("sql:select name from information_schema.users?dataSource=wildFlyExampleDS")
        .to("direct:end");
    }
}
```

When using camel-spring the route configuration would look like:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       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
       http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-
jee.xsd">

    <jee:jndi-lookup id="wildFlyExampleDS" jndi-name="java:jboss/datasources/ExampleDS"/>

    <camelContext id="sql-spring-context" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="sql:select name from information_schema.users?dataSource=#wildFlyExampleDS"
/>
            <to uri="direct:end" />
        </route>
    </camelContext>

</beans>
```

6.6.1. Spring JDBC XML namespace support

Support for the following Spring JDBC XML configurations is supported

jdbc:embedded-database

```
<jdbc:embedded-database id="datasource" type="H2">
  <jdbc:script location="db-schema.sql"/>
</jdbc:embedded-database>
```

NOTE

Only H2 databases are supported by default as JBoss EAP has native support for this. If you want to use other embedded database providers, you will need to install the appropriate database driver.

jdbc:initialize-database

```
<jdbc:initialize-database data-source="datasource">
  <jdbc:script location="classpath:db-init.sql"/>
</jdbc:initialize-database>
```

6.7. CAMEL-SOAP-REST-BRIDGE

A simple Camel route can bridge REST invocation to a legacy SOAP service. A quickstart example is provided to demonstrate the use of the **camel-soap-rest-bridge** component with Camel's REST DSL to expose a backend SOAP API service.

In this quickstart, security is involved for both REST endpoint and SOAP endpoint, both backed by RH SSO. The frontend REST API is protected via OAuth and OpenID Connect and the client will fetch a JWT (JSON Web Token) access token from RH SSO using "Resource Owner Password Credentials" OAuth2 mode. The client will use this token to access the REST endpoint.

In the bridge Camel route, the client identity is propagated from SecurityContext and when **camel-cxf producer** talks to the backend WS-SECURITY protected SOAP service, it will initially use this client identity to fetch a SAML2 token issued by the CXF STS service (which is backed by RH SSO as Identity Provider). The SAML2 token is signed and added to the WS-SECURITY header and the backend WS-SECURITY protected SOAP service will validate this SAML2 token.

The SOAP invocation also includes XSD Schema validation. If the token validation is successful, the backend SOAP service returns a response to the REST client which initiated the request.

Prerequisites

1. You have installed JBoss EAP 7.3 or later version.
2. You have installed Apache Maven 3.3.x or later later version.
3. You have installed and configured RH SSO 7.4 - follow the installation instructions at https://access.redhat.com/documentation/en-us/red_hat_single_sign-on/7.4/html/getting_started_guide/installing-standalone_#installing-server-product
4. You have installed RH SSO EAP adapter - follow the installation instructions at https://access.redhat.com/documentation/en-us/red_hat_single_sign-on/7.4/html/getting_started_guide/securing-sample-app_#installing-client-adapter

Set up the quickstart

1. Start JBOSS EAP in standalone mode.
2. Navigate to **EAP_HOME/quickstarts/camel/camel-soap-rest-bridge**
3. Enter **mvn clean install -Pdeploy** to build and deploy the quickstart.
4. Configure RH SSO
    a. Login RH SSO Admin Console from http://localhost:8180/auth with admin/admin as username/password
    b. Click **Add realm**
    c. Click **Select file**
    d. Select ./src/main/resources/keycloak-config/realm-export-new.json in this example folder which will import pre-defined necessary realm/client/user/role for this example
    e. Click **Create**

Quickstart examples in Fuse on EAP

This quickstart example which contains additional information about running the quickstart and test case outcomes is available in your Fuse on EAP installation at **EAP_HOME/quickstarts/camel/camel-soap-rest-bridge** directory.

Undeploy

To undeploy the example, navigate to the **EAP_HOME/quickstarts/camel/camel-soap-rest-bridge** directory and run **mvn clean -Pdeploy**.

6.8. ADDING COMPONENTS

Adding support for additional Camel Components is easy

Add your modules.xml definition

A modules.xml descriptor defines the class loading behavior for your component. It should be placed together with the component's jar in **modules/system/layers/fuse/org/apache/camel/component**. Module dependencies should be setup for direct compile time dependencies.

Here is an example for the camel-ftp component

```
<module xmlns="urn:jboss:module:1.1" name="org.apache.camel.component.ftp">
  <resources>
    <resource-root path="camel-ftp-2.14.0.jar" />
  </resources>
  <dependencies>
    <module name="com.jcraft.jsch" />
    <module name="javax.xml.bind.api" />
    <module name="org.apache.camel.core" />
    <module name="org.apache.commons.net" />
  </dependencies>
</module>
```

Please make sure you don't duplicate modules that are already available in WildFly and can be reused.

Add a reference to the component

To make this module visible by default to arbitrary JavaEE deployments add a reference to **modules/system/layers/fuse/org/apache/camel/component/main/module.xml**

```
<module xmlns="urn:jboss:module:1.3" name="org.apache.camel.component">
  <dependencies>
    ...
    <module name="org.apache.camel.component.ftp" export="true" services="export"/>
  </dependencies>
</module>
```

CHAPTER 7. SECURITY

Security in JBoss EAP is a vast topic. Both JBoss EAP and Camel have well documented, standardised methods of securing configuration, endpoints and payloads.

7.1. HAWTIO SECURITY

Securing the Hawtio console can be accomplished via the following steps.

1. Add system properties to standalone.xml

```
<system-properties>
    <property name="hawtio.authenticationEnabled" value="true" />
    <property name="hawtio.realm" value="hawtio-domain" />
</system-properties>
```

2. Add a security realm for Hawtio within the security subsystem

```
<security-domain name="hawtio-domain" cache-type="default">
    <authentication>
        <login-module code="RealmDirect" flag="required">
            <module-option name="realm" value="ManagementRealm"/>
        </login-module>
    </authentication>
</security-domain>
```

3. Configure a management user

```
$JBOSS_HOME/bin/add-user.sh -u someuser -p s3cret
```

Browse to http://localhost:8080/hawtio, and authenticate with the credentials configured for the management user.

7.2. JAX-RS SECURITY

The following topics explain how to secure JAX-RS endpoints.

- WildFly HTTP basic authentication
- Security Realms & SSL
- Securing EJBs

7.3. JAX-WS SECURITY

The following topics explain how to secure JAX-WS endpoints.

- WildFly HTTP basic authentication
- WS-Security
- CXF Security

- Security Realms & SSL
- Securing EJBs

7.4. JMS SECURITY

The following topics explain how to secure JMS endpoints.

- ActiveMQ Artemis security documentation
- Security settings for ActiveMQ Artemis addresses and JMS destinations
- ActiveMQ Artemis security domain configuration
- ActiveMQ Security

Additionally, you can use Camel's notion of Route Policies to integrate with the JBoss EAP security system.

7.5. ROUTE POLICY

Camel supports the notion of RoutePolicies, which can be used to integrate with the JBoss EAP security system. There are currently two supported scenarios for security integration.

7.5.1. Camel calls into JavaEE

When a camel route calls into a secured JavaEE component, it acts as a client and must provide appropriate credentials associated with the call.

You can decorate the route with a **ClientAuthorizationPolicy** as follows:

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
   @Override
   public void configure() throws Exception {
      from("direct:start")
      .policy(new ClientAuthorizationPolicy())
      .to("ejb:java:module/AnnotatedSLSB?method=doSelected");
   }
});
```

This does not do any authentication and authorization, as a part of the camel message processing. Instead, it associates the credentials that come with the Camel Exchange with the call into the EJB3 layer.

The client that calls the message consumer must provide appropriate credentials in the AUTHENTICATION header like this:

```
ProducerTemplate producer = camelctx.createProducerTemplate();
Subject subject = new Subject();
subject.getPrincipals().add(new DomainPrincipal(domain));
subject.getPrincipals().add(new EncodedUsernamePasswordPrincipal(username, password));
producer.requestBodyAndHeader("direct:start", "Kermit", Exchange.AUTHENTICATION, subject,
String.class);
```

Authentication and authorization will happen in the JavaEE layer.

7.5.2. Securing a Camel Route

In order to secure a Camel Route, you can associate a **DomainAuthorizationPolicy** with the route. This policy requires a successful authentication against the given security domain and authorization for "Role2".

```
CamelContext camelctx = new DefaultCamelContext();
camelctx.addRoutes(new RouteBuilder() {
    @Override
    public void configure() throws Exception {
        from("direct:start")
        .policy(new DomainAuthorizationPolicy().roles("Role2"))
        .transform(body().prepend("Hello "));
    }
});
camelctx.start();
```

Again, the client that calls the message consumer must provide appropriate credentials in the AUTHENTICATION header like this:

```
ProducerTemplate producer = camelctx.createProducerTemplate();
Subject subject = new Subject();
subject.getPrincipals().add(new DomainPrincipal(domain));
subject.getPrincipals().add(new EncodedUsernamePasswordPrincipal(username, password));
producer.requestBodyAndHeader("direct:start", "Kermit", Exchange.AUTHENTICATION, subject,
String.class);
```

7.6. DEPLOYING CXF JAX-WS QUICKSTART

This example demonstrates using the camel-cxf component with Red Hat Fuse on EAP to produce and consume JAX-WS web services secured by an Elytron Security Domain. Elytron is a new security framework available since EAP 7.1. In this quickstart, a Camel route takes a message payload from a direct endpoint and passes it on to a CXF producer endpoint. The producer uses the payload to pass arguments to a CXF JAX-WS web service that is secured by BASIC HTTP authentication.

Prerequisites

- Ensure that Maven installed and configured.
- Ensure that an application server with Red Hat Fuse is installed and configured.

Procedure

1. Set the **JBOSS_HOME** environment variable to point at the root directory of your application server installation.
   - For Linux

     ```
     export JBOSS_HOME=...
     ```

   - For Windows:

     ```
     ```

```
set JBOSS_HOME=...
```

2. Use the **add-user** script to create a new server application user and group.
   - For Linux

     ```
     ${JBOSS_HOME}/bin/add-user.sh -a -u testUser -p testPassword1+ -g testRole
     ```

   - For Windows:

     ```
     %JBOSS_HOME%\bin\add-user.bat -a -u testUser -p testPassword1+ -g testRole
     ```

3. Start the application server in the standalone mode.
   - For Linux

     ```
     ${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
     ```

   - For Windows:

     ```
     %JBOSS_HOME%\bin\standalone.bat -c standalone-full.xml
     ```

     The **jboss-web-xml** and **web.xml** files in the **webapp/WEB-INF** directory of this project set the application security domain, security roles and constraints.

4. Build and deploy the project.

   ```
   mvn install -Pdeploy
   ```

   This command also invokes the CLI script **configure-basic-security.cli** that creates the security domain and a few other management objects.

5. Browse to **http://localhost:8080/example-camel-cxf-jaxws-secure/**.
   A page titled **Send A Greeting** is displayed. This UI enables you to interact with the test **greeting** web service which has already started. The service WSDL is available at **http://localhost:8080/webservices/greeting-security-basic?wsdl**.

   There is a single service operation named **greet** which takes two String parameters named **message** and **name**. Invoking the web service will return a response where these values have been linked together.

Testing the Camel Secure CXF JAX-WS quickstart

1. Browse to http://localhost:8080/example-camel-cxf-jaxws-secure/.

2. On the **Send A Greeting** web form, enter a **message** and **name** into the text fields and then press the **send** button.
   The information that you have entered is displayed as a greeting on the UI. The **CamelCxfWsServlet** handles the POST request from the web UI. It retrieves the message and name from the parameter values and constructs an object array. This object array is the message payload that is sent to the **direct:start** endpoint. A **ProducerTemplate** sends the message payload to Camel. **The direct:start** endpoint passes the object array to a **cxf:bean**

web service producer. The web service response is used by **CamelCxfWsServlet** to display the greeting on the web UI. You can see the full Camel route in **src/main/webapp/WEB-INF/cxfws-security-camel-context.xml** file.

Undeploying the quickstart

1. Run the following command to undeploy the quickstart.

```
mvn clean -Pdeploy
```