Setup Camel Message Integrations

The Fedora Repository makes it possible to design custom event-driven application workflows. For instance, a common use case is sending content to an external search engine or triplestore. Other repositories may wish to generate derivative content such as creating a set of smaller images from a high-resolution master.

Because Fedora publishes modification events on a JMS topic using a local ActiveMQ broker, one can write custom listener applications to handle these various workflows. By default, the repository's JMS broker supports both the OpenWire and STOMP protocols, which means that it is possible to write client listeners or consumers in a wide variety of languages, including PHP, Python, Ruby and JAVA, among others.

For simple message-consuming applications, writing special-purpose applications may be an excellent choice. In contrast, once a repository begins making use of more complex message-based workflows or when there are multiple listener applications to manage, many repositories use systems such as Apache Camel to simplify the handling of these messages.

Camel makes use of "components" to integrate various services using a terse, domain specific language (DSL) that can be expressed in JAVA, XML, Scala or Groovy. There exists an fcrepo-camel component designed to work specifically with a Fedora4 repository. This makes it possible to model Solr indexing in only a few lines of code like so:

Camel Route using the JAVA DSL

```
XPathBuilder xpath = new XPathBuilder("/rdf:RDF/rdf:Description/rdf:type[@rdf:resource='http://fedora.info
/definitions/v4/indexing#Indexable']")
xpath.namespace("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#")

from("activemq:topic:fedora")
  .to("fcrepo:localhost:8080/fedora/rest")
  .filter(xpath)
    .to("fcrepo:localhost:8080/fedora/rest?transform=default")
    .to("http4:localhost:8080/solr/core/update");
```

In this specific case, the XPath filtering predicate is just an example; you can, of course, use many different Predicate languages, including XQuery, SQL or various Scripting Languages.

This same logic can also be expressed using the Spring XML extensions:

Camel Route using the Spring DSL

```
<route>
  <from uri="activemq:topic:fedora"/>
  <to uri="fcrepo:localhost:8080/fedora/rest"/>
  <filter>
    <xpath>/rdf:RDF/rdf:Description/rdf:type[@rdf:resource='http://fedora.info/definitions/v4
/indexing#Indexable']</xpath>
    <to uri="fcrepo:localhost:8080/fedora/rest?transform=default"/>
    <to uri="http4:localhost:8080/solr/core/update"/>
  </filter>
</route>
```

Or, in Scala:

Camel Route using the Scala DSL

```
val xpath = new XPathBuilder("/rdf:RDF/rdf:Description/rdf:type[@rdf:resource='http://fedora.info/definitions/v4
/indexing#Indexable']")
xpath.namespace("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#")

"activemq:topic:fedora" ==> {
    to("fcrepo:localhost:8080/fedora/rest")
    filter(xpath) {
        to("fcrepo:localhost:8080/fedora/rest?transform=default")
        to("http4:localhost:8080/solr/core/update")
    }
}
```

Please note that the hostnames used for Fedora and Solr in the snippets above are arbitrary. It is quite likely that these systems will be deployed on separate hosts and that the Camel routes will be deployed on yet another host. Camel makes it easy to distribute applications and replicate data asynchronously across an arbitrarily large number of independent systems.

Message Headers

By default, Fedora publishes events to a `topic` on a local broker. This topic is named "fedora". Each message will contain an empty body and up to five different header values. Those header values are namespaced so they look like this:

- org.fcrepo.jms.identifier
- org.fcrepo.jms.eventType
- org.fcrepo.jms.properties
- org.fcrepo.jms.timestamp
- org.fcrepo.jms.baseURL

Both `eventType` and `properties` are comma-delimited lists of events or properties. The eventTypes follow the JCR 2.0 specification and include:

- http://fedora.info/definitions/v4/repository#NODE_ADDED
- http://fedora.info/definitions/v4/repository#NODE_REMOVED
- http://fedora.info/definitions/v4/repository#PROPERTY_ADDED
- http://fedora.info/definitions/v4/repository#PROPERTY_CHANGED
- http://fedora.info/definitions/v4/repository#PROPERTY_REMOVED

The `properties` field will list the RDF properties that changed with that event. `NODE_REMOVED` events contain no properties. The fcrepo component for Camel is configured to recognize these headers and act appropriately.

Examples

LDPath Transformations

If an `fcr:transform` program has been installed as `mytransform`, you can generate a JSON representation of a container and send it to a low-latency, highly available document store, such as Riak. The following route determines if a container has been removed or simply added/updated. It then routes the message appropriately to a load-balancer sitting in front of the Riak HTTP endpoint.

Camel route to populate a Riak store, using the Scala DSL

```
val riakKeyProcessor = (exchange: Exchange) => {
        exchange.getIn.setHeader(
                Exchange.HTTP_PATH,
                "/buckets/fcrepo/keys/" + URLEncoder.encode(exchange.getIn.getHeader("org.fcrepo.jms.
identifier", classOf[String]))
        )
}

"activemq:topic:fedora" ==> {
        choice() {
                when(_.in("org.fcrepo.jms.eventType") == "http://fedora.info/definitions/v4
/repository#NODE_REMOVED") {
                        setHeader(Exchange.HTTP_METHOD, constant("DELETE"))
                        process(riakKeyProcessor)
                        to("http4:localhost:8098")
                }
                otherwise() {
                    to("fcrepo:localhost:8080/fedora/rest")
                        filter(xpathFilter) {
                to("fcrepo:localhost:8080/fedora/rest?transform=mytransform")
                          setHeader(Exchange.HTTP_METHOD, constant("PUT"))
                        process(riakKeyProcessor)
                to("http4:localhost:8098")
                        }
                }
        }
}
```

External Triplestore

Some additional processing must be done to transform an `application/n-triples` response into a valid `application/sparql-update` payload before sending to an external triplestore such as Fuseki or Sesame. The fcrepo component contains some processors in `org.fcrepo.camel.processor` to handle this case. The examples below assume that messages have already been routed based on `eventType` (see below) and passed to the appropriate queue.

Populate an external triplestore

```
from("direct:delete")
  .process(new SparqlDescribeProcessor())
  .to("http4:localhost:3030/db/query")
  .process(new SparqlDeleteProcessor())
  .to("http4:localhost:3030/db/update");

from("direct:new")
 .to("fcrepo:localhost:8080/rest")
 .process(new SparqlInsertProcessor())
 .to("http4:localhost:3030/db/update");

from("direct:update")
  .to("fcrepo:localhost:8080/rest")
  .process(new SparqlUpdateProcessor())
  .to("http4:localhost:3030/db/update");
```

When using these `Sparql*` processor classes, it is also possible to apply these operations to named graphs. While Fedora does not support named graphs, it is possible to assign nodes to certain named graphs in an external triplestore. The `CamelFcrepoNamedGraph` header can be used to apply a `sparql-update` operation to a particular named graph. For instance, to route all operations to a named graph (in this case the graph URI is defined dynamically as a property placeholder:

Named Graph

```
from("direct:update")
  .to("fcrepo:localhost:8080/rest")
  .setHeader(FcrepoHeaders.FCREPO_NAMED_GRAPH)
    .simple("{{named.graph}}")
  .process(new SparqlUpdateProcessor())
  .to("http4:localhost:3030/ds/update");
```

Or, to partition the graph based on an existing RDF property:

Multiple Named Graphs

```
from("direct:update")
  .to("fcrepo:localhost:8080/rest")
  .setHeader(FcrepoHeaders.FCREPO_NAMED_GRAPH)
    .xpath("/rdf:RDF/rdf:Description/ex:namedGraph/text()", String.class, ns)
  .process(new SparqlUpdateProcessor())
  .to("http4:localhost:3030/ds/update");
```

Or, you may want (possibly overlapping) "public" and "private" named graphs, defined as constants:

Constant Named Graphs

```
from("direct:update")
  .to("fcrepo:localhost:8080/rest")
  .multicast("direct:public", "direct:private");

from("direct:public")
  .filter(publicPredicate)
    .setHeader(FcrepoHeaders.FCREPO_NAMED_GRAPH)
      .constant("http://site/graph/public")
    .process(new SparqlUpdateProcessor())
    .to("http4:localhost:3030/ds/update");

from("direct:private")
  .filter(privatePredicate)
    .setHeader(FcrepoHeaders.FCREPO_NAMED_GRAPH)
      .constant("http://site/graph/private")
    .process(new SparqlUpdateProcessor())
    .to("http4:localhost:3030/ds/update");
```

In each case the Sparql update operation will include a `GRAPH <uri> { ... }` clause, where `uri` is the value of the `FCREPO_NAMED_GRAPH` header (in the Spring DSL, this header can be accessed as `CamelFcrepoNamedGraph`). It is important that the value of the named graph header is a properly formed URI.

Event-based Routing

It is often helpful to route messages to different queues based on the `eventType` value. This example splits messages on `eventType` values and routes the messages to appropriate queues. Following this example, it would be prudent to aggregate the messages based on `org.fcrepo.jms.identifier` value after retrieving the messages from the downstream queues.

Content-based Routing

```
<route id="fcrepo-event-splitter">
    <description>
        Retrieve messages from the fedora topic. Event types are comma-delimited, so split them into separate
messages before routing them.
    </description>
    <from uri="activemq:topic:fedora"/>
    <setBody>
        <simple>${header.org.fcrepo.jms.eventType}</simple>
    </setBody>
    <split>
        <tokenize token=","/>
        <setHeader headerName="org.fcrepo.jms.eventType">
            <simple>${body}</simple>
        </setHeader>
        <setBody>
            <simple>null</simple>
        </setBody>
        <to uri="seda:fcrepo-event-router"/>
    </split>
</route>

<route id="fcrepo-event-router">
    <description>
        Route messages based on the eventType.
    </description>
    <from uri="seda:fcrepo-event-router"/>
    <choice>
        <when>
            <simple>${header.org.fcrepo.jms.eventType} == "http://fedora.info/definitions/v4
/repository#NODE_REMOVED"</simple>
            <to uri="activemq:queue:fcrepo.delete"/>
        </when>
        <when>
            <simple>${header.org.fcrepo.jms.eventType} == "http://fedora.info/definitions/v4
/repository#NODE_ADDED"</simple>
            <to uri="activemq:queue:fcrepo.add"/>
        </when>
        <when>
            <simple>${header.org.fcrepo.jms.eventType} == "http://fedora.info/definitions/v4
/repository#PROPERTY_ADDED"</simple>
            <to uri="activemq:queue:fcrepo.update"/>
        </when>
        <when>
            <simple>${header.org.fcrepo.jms.eventType} == "http://fedora.info/definitions/v4
/repository#PROPERTY_CHANGED"</simple>
            <to uri="activemq:queue:fcrepo.update"/>
        </when>
        <when>
            <simple>${header.org.fcrepo.jms.eventType} == "http://fedora.info/definitions/v4
/repository#PROPERTY_REMOVED"</simple>
            <to uri="activemq:queue:fcrepo.update"/>
        </when>
        <otherwise>
            <log message="No router for ${header.org.fcrepo.jms.eventType}"/>
        </otherwise>
    </choice>
</route>
```

Supporting Queues

The default configuration is fine for locally-deployed listeners, but it can be problematic in a distributed context. For instance, if the listener is restarted while a message is sent to the topic, that message may be missed. Furthermore, if there is a networking hiccup between Fedora's local broker and the remote listener, that too can result in lost messages. Instead, in this case, a queue may be better suited.

ActiveMQ supports “virtual destinations”, allowing your broker to automatically forward messages from one location to another. If Fedora4 is deployed in Tomcat, the ActiveMQ configuration will be located in `WEB-INF/classes/config/activemq.xml`. That file can be edited to include the following block:

activemq.xml customization: supporting a queue/fedora endpoint

```
<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
      <compositeTopic name="fedora">
        <forwardTo>
          <queue physicalName="fedora"/>
        </forwardTo>
      </compositeTopic>
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>
```

Now a consumer can pull messages from a queue without risk of losing messages.

This configuration, however, will not allow any other applications to read from the original `topic`. If it is necessary to have `/topic/fedora` available to consumers, this configuration will be useful:

activemq.xml customization: supporting topic and queue endpoints

```
<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
      <compositeTopic name="fedora" forwardOnly="false">
        <forwardTo>
          <queue physicalName="fedora"/>
        </forwardTo>
      </compositeTopic>
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>
```

Now, both `/topic/fedora` and `/queue/fedora` will be available to consumers.

Distributed Brokers

The above example will allow you to distribute the message consumers across multiple machines without missing messages, but it can also be useful to distribute the message broker across multiple machines. This can be especially useful if you want to further decouple the message producers and consumers. It can also be useful for high-availability and failover support.

ActiveMQ supports a variety of distributed broker topologies. To push messages from both the message queue and topic to a remote broker, this configuration can be used:

activemq.xml customization: distributed brokers

```
<networkConnectors>
  <networkConnector name="fedora_bridge" dynamicOnly="true" uri="static:(tcp://remote-host:61616)">
    <dynamicallyIncludedDestinations>
      <topic physicalName="fedora"/>
      <queue physicalName="fedora"/>
    </dynamicallyIncludedDestinations>
  </networkConnector>
</networkConnectors>
```

Protocol Support

ActiveMQ brokers support a wide variety of protocols. If Fedora's internal broker is bridged to an external broker, please remember to enable the proper protocols on the remote broker. This can be done like so:

activemq.xml customization: protocol support

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
  <transportConnector name="stomp" uri="stomp://0.0.0.0:61613"/>
</transportConnectors>
```

Each `transportConnector` supports many additional options that can be added to this configuration.

Deployment

Camel routes can be deployed in any JVM container. In order to deploy to Jetty or Tomcat, the route must be built as a WAR file. This command will get you started:

```
$> mvn archetype:generate \
  -DarchetypeGroupId=org.apache.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-war \
  -DarchetypeVersion=2.14.0 \
  -DgroupId=org.example.camel \
  -DartifactId=my-camel-route \
  -Dversion=1.0.0-SNAPSHOT \
  -Dpackage=org.example.camel
```

After the project has been built (`mvn install`), you will find the WAR file in `./target`. That file can simply be copied to the `webapps` directory of your Jetty/Tomcat server.

Another popular deployment option is Karaf, which is a light-weight OSGi-based JVM container. Karaf has the advantage of supporting hot code swapping, which allows you to make sure that your routes are always running. It also allows you to deploy XML-based routes (Spring or Blueprint) by simply copying the files into a `$KARAF_HOME/deploy` directory. If deploying camel routes to Karaf, Blueprint-based routes have some advantages over the Spring-based DSL, particularly in terms of being able to use property placeholders within your routes.

Karaf can be set up by:

1. downloading Karaf from an apache.org mirror
2. running ./bin/karaf to enter the shell
3. installing required bundles (n.b. the following commands correspond to Karaf 3.x. For Karaf 2.x installations, please refer to the Karaf 2.x documentation):

Karaf console

```
$> feature:repo-add camel 2.14.0
$> feature:repo-add activemq 5.10.0
$> feature:install camel
$> feature:install activemq-camel

# display available camel features
$> feature:list | grep camel

# install camel features, as needed
$> feature:install camel-http4

# install fcrepo-camel (as of v4.1.0)
$> feature:repo-add mvn:org.fcrepo.camel/fcrepo-camel/4.1.0/xml/features
$> feature:install fcrepo-camel
```

4. setting up a service wrapper (so that karaf runs as a system-level service)

Karaf console

```
$> feature:install wrapper
$> wrapper:install
```

5. following the directions provided by this command

Now, routes can be deployed (and re-deployed) by simply copying JAR files or XML documents to `$KARAF_HOME/deploy`.

Monitoring Your Camel Routes

It is often useful to keep runtime statistics for your camel routes. Hawtio is a web console for monitoring your messaging infrastructure, and it can be deployed in any JVM container, including Karaf, Tomcat or Jetty.

In Karaf, hawtio can be installed like so:

Karaf console

```
$> feature:repo-add hawtio 1.4.29
$> feature:install hawtio
```

Once deployed, hawtio is available at http://localhost:8181/hawtio/

With Tomcat or Jetty, deploying hawtio is simply a matter of installing a WAR file. Please see the hawtio website for more information.