Messaging

Table of Contents

Introduction

Messaging is a communication mechanism used to send and receive information in a manner which allows the senders and receivers to be unaware of the activities or status of the other parties involved in the exchange. This loose coupling is achieved through the use of an intermediary queue or topic managed by a messaging provider. The messaging provider is responsible for delivering messages sent by message producers to message consumers. Messaging in Fedora is implemented using the Java Messaging Service (JMS) which is a specification used by many messaging providers that allows messages to be sent or received from a queue or topic in a generic fashion.

Messaging in Fedora

The goal of messaging in Fedora is to provide updates about the activities of the repository as they occur. This allows external applications to monitor and perform actions based on those activities. In order to provide the capability for multiple independent clients to receive identical update messages simultaneously, an asynchronous publish-and-subscribe model was chosen as the default for Fedora's messaging capability. The messages sent using this model indicate when functions of API-M have been exercised, thus providing information about every update made to digital objects within the repository.

Fedora uses Apache ActiveMQ as its default messaging provider. While the use of JMS suggests that any messaging provider supporting JMS can replace ActiveMQ, no other providers have been tested. If you do use Fedora with another JMS-compliant messaging provider, please let us know your results.

Configuring Messaging

Messaging in Fedora is configured primarily through the messaging module within the fedora.fcfg file. The following parameters, specified as part of the messaging module, are required in order to establish a JMS Connection:

- **enabled**
  - Default: false
  - Determines whether the messaging module should be initialized in order to send messages. If you want messages to be sent, this must be set to true.
- **java.naming.factory.initial**
  - Default: org.apache.activemq.jndi.ActiveMQInitialContextFactory
  - Specifies the JNDI initial context with which the connection factory and destination administered objects will be looked up. As indicated by the default value, Fedora uses ActiveMQ as its default JNDI provider as well as its default messaging provider. The messaging provider and JNDI provider do not need to be the same.
- **java.naming.provider.url**
  - Default: vm:(broker:(tcp://localhost:61616))
  - Specifies the address at which a connection can be made to the messaging provider. Depending on the provider, this address may indicate any of several protocols and may include additional parameters. The default URL uses ActiveMQ specific syntax to allow for both internal JVM transport and transport over TCP connections via port 61616. More information on ActiveMQ broker URIs can be found on their website.
  - Fedora will attempt to connect to the messaging provider at this address on startup, so make sure that your provider is running and available.
- **connection.factory.name**
  - Default: ConnectionFactory
  - Specifies the JNDI name of the ConnectionFactory object needed to create a connection to the JMS provider. ActiveMQ creates a connection factory on startup and stores it under the name ConnectionFactory in its included JNDI provider. If you are using a different JMS or JNDI provider, you will need to create a connection factory in JNDI and specify the name under which it is stored as the value of this parameter.

Once a connection is established to the JMS provider, Fedora needs to know where to publish messages. Two topics are currently available for this purpose:

- **fedora.apim.update** - API-M methods which cause an update to occur within the repository. This includes all ingest, add, modify, set, and purge activities.
- **fedora.apim.access** - API-M methods which access the repository but do not cause an update to occur. This includes all methods not considered updates.

The names of these topics may be changed by specifying new values for the name parameter of the apimUpdateMessages and apimAccessMessages datastores within the fedora.fcfg file. Changing the names will not alter the messages being sent but will send those messages to different destinations.

If a point-to-point messaging model is preferred, the type parameters of the datastores mentioned above can be changed to "queue", which will result in the messages being placed in a queue of the name specified in the datastore. Queues allow only a single entity to retrieve and process each message, but they do remove timing dependencies inherent with the publish-and-subscribe model (subscribers must register their interest prior to a message being published in order to receive that message.)

Messaging Client

In order to receive the messages that are being sent by Fedora, you will need to create a message consumer to listen for Fedora's notification messages. To aid in this effort, the Messaging Client was created. To build the Messaging Client, run the messaging-client Ant target from the source distribution. After the build completes, look in the dist directory for fedora-messaging-client.zip, which includes all of the jars necessary to use the Fedora Messaging Client. If you are using a messaging provider other than ActiveMQ, you will need to replace the activemq-all jar with the appropriate jars from your messaging provider of choice.

The Messaging Client was designed to provide a simple Java interface for receiving messages from Fedora. Some configuration parameters are necessary in order for the client to create a connection and listen to the appropriate topic or queue. The parameter names here are the same as those listed above for messaging, and the values should be the same as those in your fedora.fcfg file. The topic or queue name(s) on which to listen are also included as parameters and the value(s) should match those in the fedora.fcfg datastores. If you are using ActiveMQ as your JMS and JNDI providers each topic or queue will be created for you, otherwise you will need to create destination object(s) in JNDI to match the property values you specify here.

- **java.naming.factory.initial** - this String value can be found in javax.naming.Context.INITIAL_CONTEXT_FACTORY
- **java.naming.provider.url** - this String value can be found in javax.naming.Context.PROVIDER_URL
- **connection.factory.name** - this String value can be found in fedora.server.messaging.JMSManager.CONNECTION_FACTORY_NAME
- **topic.{name} OR queue.{name}** - topics are specified using the prefix "topic." followed by a topic name. Queues are specified using the prefix "queue." followed by a queue name.

The code below is a simple example of how to use the Messaging Client. The JmsMessagingClient constructor includes three required parameters and two optional parameters. The required parameters include the client ID, the MessagingListener instance, and the connection properties mentioned above. The optional parameters include a message selector and flag which determines whether durable subscribers should be used to listen over the topics listed in the properties. More information about each of the available parameters can be found in the JmsMessagingClient javadocs.

```
public class Example implements MessagingListener {
    MessagingClient messagingClient;
    public void start() throws MessagingException {
        Properties properties = new Properties();
        properties.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                               "org.apache.activemq.jndi.ActiveMQInitialContextFactory");
        properties.setProperty(Context.PROVIDER_URL, "tcp://localhost:61616");
        properties.setProperty(JMSManager.CONNECTION_FACTORY_NAME, "ConnectionFactory");
        properties.setProperty("topic.fedora", "fedora.apim.*");
        messagingClient = new JmsMessagingClient("example1", this, properties, false);
        messagingClient.start();
    }
    public void stop() throws MessagingException {
        messagingClient.stop(false);
    }
    public void onMessage(String clientId, Message message) {
        String messageText = "";
        try {
            messageText = ((TextMessage)message).getText();
        } catch(JMSException e) {
            System.err.println("Error retrieving message text " + e.getMessage());
        }
        System.out.println("Message received: " + messageText + " from client " + clientId);
    }
}
```

A new feature in Fedora version 3.1 is the option to start the JmsMessagingClient asynchronously. If you are starting a messaging client only to listen for messages on a topic, there is likely no need to wait for that connection to be made before continuing with other processing. Once the connection is made, messages will be passed to your onMessage() method as usual. To take advantage of this, simply call jmsMessagingClient.start(false) rather than messagingClient.start() in the example above. Note that if you intend to publish messages, you will still need to wait for the connection to complete (i.e. use messagingClient.start()) prior to adding a message to a topic or queue.

Messages

The content of messages sent by Fedora takes the form of feed entries based on the Atom Syndication Format. These messages correspond to API-M method calls, indicating the name of the method, its parameters, return value, and other information about the method. Each message will be similar to the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<entry xmlns="http://www.w3.org/2005/Atom"
       xmlns:fedora-types="http://www.fedora.info/definitions/1/0/types/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <id>urn:uuid:3773e144-1b63-4dde-8786-464243af9186</id>
  <updated>2008-04-14T22:35:13.953Z</updated>
  <author>
    <name>fedoraAdmin</name>
    <uri>http://localhost:8080/fedora</uri>
  </author>
  <title type="text">purgeObject</title>
  <category term="demo:5" scheme="fedora-types:pid" label="xsd:string"></category>
  <category term="purge message" scheme="fedora-types:logMessage" label="xsd:string"></category>
  <category term="false" scheme="fedora-types:force" label="xsd:boolean"></category>
  <summary type="text">demo:5</summary>
  <content type="text">2008-04-14T22:35:13.953Z</content>
</entry>
```

The Atom tags in each message will have the following values:

- <id> uniquely identifies each entry
- <updated> indicates the date and time at which the call occurred
- <author> identifies the initiation point of the API-M method call
    - <name> specifies the name of the user making the call
    - <uri> corresponds to the baseURL of the Fedora repository from which the call originated
- <title> specifies the method name
- Each <category> corresponds to a method's argument:
    - The term indicates the argument value. However, null values are indicated as "null", and non-null xsd:base64Binary values are indicated as "[OMITTED]".
    - The scheme indicates the argument name
    - The label indicates the argument datatype
- <summary> corresponds to the PID of the object operated on by the method, if applicable.
- <content> corresponds to the textual representation of the method's return value, noting the following:
    - Null values are represented as "null".
    - fedora-types:ArrayOfString values are represented as a comma-separated list, e.g. "value1, value2, value3".
    - Non-null xsd:base64Binary values are not returned, and only indicated as "[OMITTED]".
    - Non-null fedora-types:Datastream values are not returned, and only indicated as "[OMITTED]".
    - fedora-types:RelationshipTuple values are represented in Notation3 (N3).

Two properties are included as part of each JMS message produced by Fedora, primarily for use by message selectors. These two properties are: **methodName** and **pid**. A message selector, which can be specified using the Messaging Client, is used to limit the messages that will be delivered based on selection criteria. An example of a message selector is: "methodName LIKE 'purge%'". This selector would limit the messages received by the client to only those messages for which the method is a purge (purgeObject and purgeDatastream.)

Configuring Fedora with an external ActiveMQ broker for higher availability

By default, Fedora will shut down if it cannot contact your configured ActiveMQ broker upon startup. On the other hand, if your external broker shuts down after Fedora has successfully established a connection to it, no recovery or shutdown is performed, meaning that you could lose messages. The following steps show how to configure a simple store-and-forward ActiveMQ message broker bridge between Fedora (producer broker) and a remote ESB (message consumer broker) set up as a failover. This means that Fedora will:

1. upon startup, load the ActiveMQ embedded broker that ships with Fedora;
2. configure the embedded broker to:
    a. forward Fedora messages to a remote broker, if the remote broker is available;
    b. if the remote broker is not available, store the messages until the remote broker becomes available, at which point it will forward the stored messages to it
3. use the ActiveMQ “failover” transport to attempt periodic reconnections to the remote broker when it is unavailable

Best Practice

The following configuration is recommended as a **best practice** in production environments where Fedora messages are a critical part of the repository workflow.

Step 1: Install a remote broker

Install a remote broker, and make sure it can receive messages via TCP. Start it up, make sure that it runs, binds to the correct address and port. Maybe even run a few test, to verify that messages arrive and are processed. In the example configurations below, the remote broker is bound to 0.0.0.0 (all interfaces), port 61617.

Step 2: Configure Fedora

1. You will need to make a few jars available to Fedora so it can load and process the ActiveMQ configuration file. Drop the jars `xbean-spring-3.4.3.jar` and `spring-context-2.5.6.jar` files into the Fedora webapp WEB-INF/lib directory. Most spring applications provide these jars; if you are using Apache Servicemix as the container for your remote broker, they can be found in the `SERVICEMIX_HOME/lib` directory.
2. Create the Fedora `activemq.xml` file, put it in `FEDORA_HOME/server/config`.

```
<beans xmlns:amq="http://activemq.apache.org/schema/core">

  <!-- ActiveMQ JMS Broker configuration -->
  <amq:broker id="broker" useShutdownHook="false">

    <amq:managementContext>
      <amq:managementContext connectorPort="1093" createConnector="false"/>
    </amq:managementContext>

    <!-- Your remote broker, configured with failover -->
    <amq:networkConnectors>
      <amq:networkConnector uri="static:(failover:(tcp://0.0.0.0:61617))"/>
    </amq:networkConnectors>

    <!-- The directory where Fedora will store the ActiveMQ data -->
    <amq:persistenceAdapter>
      <amq:amqPersistenceAdapter directory="file:./data/amq"/>
    </amq:persistenceAdapter>
  </amq:broker>

  <!-- Set this to prevent objects from being serialized when
       passed along to your embedded broker;  saves some overhead processing -->
  <bean id="jmsConnectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
     <property name="objectMessageSerializationDefered" value="false"/>

  </bean>

</beans>
```

This piece of the config file creates the bridge to the remote broker, configured in failover mode:

```
<amq:networkConnectors>
  <amq:networkConnector uri="static:(failover:(tcp://0.0.0.0:61617))"/>
</amq:networkConnectors>
```

3. Configure Fedora to read the `activemq.xml` file upon startup, create the embedded broker.
Make sure messaging is enabled in `fedora.fcfg`:

```
<module role="org.fcrepo.server.messaging.Messaging" class="org.fcrepo.server.messaging.MessagingModule">
  <comment>Fedora's Java Messaging Service (JMS) Module</comment>
  <param name="enabled" value="true"/>
[...]
```

Also change the `java.naming.provider.url` parameter:

```
<param name="java.naming.provider.url" value="vm://localhost?brokerConfig=xbean:file:/path/to/fedora_home
/server/config/activemq.xml"/>
```

4. Restart Fedora, and start up your remote broker (or the other way around: it no longer matters).
The ActiveMQ failover transport documentation contains a list of useful parameters that you can use to configure connection management to the remote broker (max reconnect attempts, period between reconnects, etc.)

Useful Documentation

Configuring an ActiveMQ store-and-forward network of brokers: http://activemq.apache.org/how-do-distributed-queues-work.html

- With failover: http://bsnyderblog.blogspot.com/2010/01/how-to-use-automatic-failover-in.html
- Failover documentation: http://activemq.apache.org/failover-transport-reference.html

Using the `xbean:file` URI to load the ActiveMQ broker configuration from a Spring bean config file: http://activemq.apache.org/broker-xbean-uri.html