EMBEDDED MESSAGING USING ACTIVEMQ

Embedded messaging is useful when you need localized messaging within your application and don't need (or want) an external message broker. It's a good technique for decoupling components within your application, resolving application bottlenecks, increasing the overall scalability of your application, or adding an embedded entry point for remote access to your application.

Introduction

Have you ever wanted to leverage JMS messaging in your application, but couldn't because your company doesn't have a messaging infrastructure in place? Have you ever had a simple JMS messaging need in your application, but didn't want to go through the hassles of setting up an entire messaging infrastructure for one small use case? Have you ever wanted to scale out portions of your monolithic application to resolve bottlenecks, but didn't want to deal with the complexity of threads and concurrency? If so, you might want to consider using embedded messaging.

Embedded messaging is a technique that lets you create an internal message broker within your application and leverage all of the features of JMS messaging internally without the need for an external message broker. In this article I will start with the basics of embedded messaging using ActiveMQ, and then show you a real world example of how to use embedded messaging to resolve application bottlenecks within your application. If you want more information about embedded messaging (as well as a plethora of other really cool messaging topics), you can view two new enterprise messaging videos I recently recorded from O'Reilly. The references to these videos can be found at the end of this article.

The Basics

Many of the popular open source message brokers available today support embedded messaging, including ActiveMQ, OpenMQ, and HornetQ. Embedded messaging is a technique that allows you to create an internal message broker instance within the context (JVM) of your application. Aside from possibly clustering, there is little or no difference between the broker you create within your application and an external broker started in a separate JVM.

There are a few things to consider before jumping right into using embedded messaging. When using embedded messaging to decouple components within an application, you cannot perform the intended operation in the same transactional unit of work. For example, suppose class A is invoking an operation on class B within the context of a transaction. If you decouple these components using embedded messaging, the operation invoked in class B (now a message listener) will run under a different transaction context than class A.

Another thing to be aware of when using embedded messaging is that you need to design your application in such a way as to ensure that the embedded broker is always started and available before starting any message listeners. While this may seem really obvious, it does get a bit tricky at times, particularly for complex applications.

There are two possible topologies to consider as well when designing your application to use embedded messaging. The most basic (and typical) embedded messaging topology is one where the scope of the embedded broker is limited to only the application (and JVM) the broker is started in. In other words, other applications in separate JVMs are completely unaware that the embedded message broker exists. The second topology is one where the internal broker exposes a port and IP address so that other applications can talk to it as well. Due to the limitations of this second type of topology (specifically embedded broker availability when the application is down), it is really only useful in the context of providing an embedded remote endpoint. For example, suppose application A has a service that can be remotely accessed from application B. In this case application A can start up an embedded broker with a queue to act as an end point for the service. Application B can now connect to the embedded broker in application A and communicate with that remote service.

Creating a Broker

There are numerous ways of creating an embedded broker within your application. For ActiveMQ (which I will be using for this article), you can directly instantiate the ActiveMQ `BrokerService` class, or create the `BrokerService` through an ActiveMQ `BrokerFactory` class.

You will need to add a couple of JAR files from the ActiveMQ distribution (specifically from the `ACTIVEMQ_HOME/lib directory`) to create and run an embedded broker. In most cases you will only need to add the `activemq-all-5.9.1.jar` file (or whatever version you may be using) located in the `ACTIVEMQ_HOME` root distribution directory. Depending on what broker features you are using, you may need to add one or more of the following JAR files, also found in the `ACTIVEMQ_HOME/lib` directory:

```
commons-logging.jar
log4j-1.2.17.jar
spring-beans-3.2.5.RELEASE.jar
spring-context-3.2.5.RELEASE.jar
spring-core-3.2.5.RELEASE.jar
spring-expression-3.2.5.RELEASE.jar
xbean-spring-3.15.jar
commons-logging.jar
```

There are times when the `activemq-all-5.9.1.jar` may cause some conflicts (particularly with some application servers), so if you experience this you can use the `activemq-broker-5.9.1.jar` and the `activemq-client-5.9.1.jar` in place of the `activemq-all-5.9.1.jar`.

To create a broker using the ActiveMQ BrokerService, you first instantiate a `BrokerService` object, then set the necessary configuration through Java as follows:

```
BrokerService broker = new BrokerService();
broker.addConnector("tcp://localhost:61888");
broker.setBrokerName("broker1");
broker.setPersistent(false);
//other broker properties can be set here...
broker.start();
```

Once you instantiate the `BrokerService` you will need to set some broker configuration parameters. There are literally hundreds of parameters you can set, but the ones I am showing here are the three most basic ones. The `addConnector()` operation is only needed if you intend on exposing your embedded broker to outside applications. For internal application use only, you should not set this property.

If you don't like setting configuration parameters in Java, you can use another technique in ActiveMQ that uses an XML properties file along with a `BrokerFactory` class. This technique encapsulates all of the configuration parameters in a single XML configuration file located in your classpath. While you can use any name you like, I prefer using the standard `activemq.xml`

file name. When you create a broker from the `BrokerFactory`, you pass in the name of the XML configuration file as follows:

```
BrokerService broker = BrokerFactory.createBroker(
  new URI("xbean:activemq.xml");
broker.start();
```

Your XML configuration file should be located somewhere in the application classpath (I typically use `src/main/resources`). Although there are hundreds of properties you can set, the bare minimum configuration file would look like this:

```
<broker xmlns="http://activemq.apache.org/schema/core"
  brokerName="broker1" persistent="false">

<!-- other properties can be set here... -->

<transportConnectors>
       <transportConnector
           name="openwire" uri="tcp://0.0.0.0:61888/>
   </transportConnectors>
</broker>
```

Accessing an Embedded Broker

Normally, when establishing a connection to an external ActiveMQ message broker you would create a `Connection` object through an ActiveMQ connection factory and pass it the IP address and port it is listening on as follows:

```
Connection connection = new ActiveMQConnectionFactory(
   "tcp://localhost:61616").createConnection();
```

While you can use this technique for accessing an embedded broker, there is a much better way. ActiveMQ has a special protocol for accessing an embedded broker called the `vm` protocol. When using the `vm` protocol, you use the broker name rather than an IP address and port when making a connection to the embedded broker:

```
Connection connection = new ActiveMQConnectionFactory(
   "vm://broker1?create=false").createConnection();
```

Notice I added the `?create=false parameter` to the broker address. I usually do this because if the embedded broker doesn't exist when making a connection the broker, ActiveMQ will dynamically create the broker for you. While this is a neat feature, I prefer to create the broker explicitly so I can control the default configuration settings.

Using an Embedded Broker

Now that you know the basics of setting up and accessing an embedded broker, its time to see how to use one within an application. Rather than make up a fictitious example, I will show you a real-world example I recently coded and deployed at the company I am currently working at.

In the application I am currently working on, application reference data is loaded from the database into an internal cache on application startup with the ability to reload the internal data cache on demand whenever there is a change to the reference data. Specifically, there are nine reference tables that are loaded, some of them rather large. The basic code skeleton looks like this (all of the details are removed for brevity):

```
public class ReferenceManager {

   //key is the table name
   private Map<String, ReferenceCache> cache;

   public void loadRefData() {
      loadTable1();
      loadTable2();
      ...
   }

   private void loadTable1() {
      //perform database query
      //format data and add to cache
   }
   ...
}
```

Due to the size of the reference tables and the formatting of the data prior to adding it to the cache, this process was taking around 90 seconds to complete. Since the application startup and reload is part of a larger transaction, this process was sometimes causing transaction timeouts, particularly under heavy load. By adding embedded messaging, the table loads from

the database were able to be run asynchronously and in parallel without the need for complex threading or external message broker configuration and setup. This reduced the end-to-end cache load elapsed time to just under 12 seconds.

Let's run through the changes needed to add embedded messaging to run each individual table load in parallel. The first change to the original synchronous `ReferenceManager` class is to add an embedded broker to the startup method to the `ReferenceManager`. Depending on the implementation of your class, this can be done through the constructor, in the post construct method of the bean, or a separate public method called after instantiation:

```
public ReferenceManager() {
   BrokerService broker = new BrokerService();
   broker.setBrokerName("refbroker");
   broker.setPersistent(false);
   broker.start();
}
```

Since the embedded broker is not exposed to other applications outside of the JVM, you do not need to use the `useConnector()` property (only the `setBrokerName()`). Generally you will not need message persistence, which is why you should consider setting the persistence property to false.

Once the embedded broker is defined, the next step is to move the logic to load each table into a separate message listener class (one listener for each of the nine tables or a single listener with conditional logic based on the queue name or message property). These listener classes can be a simple POJO-based JMS message listener (as is shown here), or it can be a Spring message-driven POJO or EJB message-driven bean. Usually you will find that a simple POJO-based message listener class is most practical because you will need control of when the listener starts (remember, the embedded broker must be started before the listeners can start!).

```
public class RefTable1Loader implements MessageListener {
   Session session = null;

   public RefTable1Loader(String queueName) {
      Connection conn =
         new ActiveMQConnectionFactory(
         "vm://refbroker?create=false").createConnection();
      conn.start();
      session = conn.createSession(
         false, Session.AUTO_ACKNOWLEDGE);
      Queue queue = session.createQueue(queueName);
      MessageConsumer consumer =
         session.createConsumer(queue);
      consumer.setMessageListener(this);
   }

   public void onMessage(Message message) {
      //perform database query
      //format data as formattedData
      ObjectMessage msg =
         session.createObjectMessage(formattedData);
      MessageProducer sender =
         session.createProducer(message.getJMSReplyTo());
      sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
      sender.send(msg);
   }
}
```

Notice that the message listener above establishes a connection to the broker named `refbroker` through the vm protocol using the string `vm://refbroker?create=false`. As long as the listeners are in the same JVM, you can (and should) use the `vm` protocol.

In the reference manager class you then need to start each of the listeners. This can be done in the startup method after you start the embedded broker instance. Notice that you do not need to hold a references to these listeners in the `ReferenceManager`; they can be created anonymously because the only thing we care about is that the listener thread starts in each of the classes:

```
private void startListeners() {
   new RefTable1Listener("TABLE1_REQ.Q");
   new RefTable2Listener("TABLE2_REQ.Q");
   ...
}
```

Once the listeners are started, the next thing to do is to modify the original `loadRefData()` method to send a message to each of the listeners, then wait for the response for each of them. Once the response is received, the data from each response message containing the formatted cache data is added to the reference cache:

```
public void loadRefData() {
   Connection conn =
      new ActiveMQConnectionFactory(
      "vm://refbroker?create=false").createConnection();
   conn.start();

   //send the messages to the listeners
   startLoad(conn, "TABLE1_REQ.Q  , "TABLE1_RESP.Q");
   startLoad(conn, "TABLE2_REQ.Q  , "TABLE2_RESP.Q");
   ...

   //now get the results from the response queues
   cache.put("Table1", getLoadResults(conn, "TABLE1_RESP.Q");
   cache.put("Table2", getLoadResults(conn, "TABLE2_RESP.Q");
   ...
   conn.close();
}
```

The `startLoad()` method creates a `Message` object, sets the `JMSReplyTo`, and sends the message to the request queue as a non-persistent message. Notice that this method only creates a `Message` object (rather than a `TextMessage` or one of the other message types). This is because you only need to send an event to the listener to start the load process. Alternatively, you can set application properties on the message or use a `TextMessage` to send additional information to the listeners if needed.

```
private void startLoad(Connection conn,
   String requestQ, String responseQ) {

   Session session =
      conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
   Queue queueReq = session.createQueue(requestQ);
   Queue queueResp = session.createQueue(responseQ);
   Message msg = session.createMessage();
   msg.setJMSReplyTo(queueResp);
   MessageProducer sender =
      session.createProducer(queueReq);
   sender.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
   sender.send(msg);
   session.close();
}
```

The final step is to then use request/reply processing to wait for the data to be returned via the response queue. You don't need to use a correlation id and filtering here because there is only one response message in each of the dedicated response queues.

```
private ReferenceData getLoadResults(
   Connection conn, String queueName) {

   Session session =
      conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
   Queue queue = session.createQueue(queueName);
   MessageConsumer receiver = session.createConsumer(queue);
   ObjectMessage msg = (ObjectMessage)receiver.receive(30000);
   if (msg == null) {
      throw new RuntimeException(
         "No response from " + queueName +
         ": Unable to load data.");
   }
   session.close();
   return (ReferenceData)msg.getObject();
}
```

Notice here I added some error handling that is necessary when receiving a reply from the listeners. In this case, if the consumer doesn't receive a response from the response queue within 30 seconds it assumes something went wrong. You will then need to take corrective

action (or in this case throws an exception). In the actual `ReferenceManager` class I wrote, on an error such as this I revert back to sequential processing in the spirit of reactive architecture so that the request can still be completed (albeit slower).

In this real-world example I skipped all of the ceremony involved with error handling, cache synchronization, method delegation, constants, and many other improvements you can make to streamline this code. While the actual code is rather complicated, the embedded messaging portions are not. Hence, I only highlight those elements in the code to demonstrate the technique of using embedded messaging in your applications.

Alternatives

While embedded messaging is an easy and reliable technique for introducing asynchronous behavior into your applications, there are some alternatives you might want to consider. Within the EJB space you can use asynchronous beans (coupled with `Future` objects if you need a return value). For a more general approach, you can use Actors such as those supplied by the Akka framework (which is available in Java and Scala). In cases where you cannot use asynchronous beans (such as application server startup), threads, or third-party asynchronous libraries, embedded messaging provides a very fast, reliable, and easy way to perform parallel processing and increase both performance and scalability.

References and Further Reading

- Enterprise Messaging: JMS 1.1 and JMS 2.0 Fundamentals Video (Mark Richards, O'Reilly, 2014) (O'Reilly Shop, Safari Books Online)
- Enterprise Messaging: Advanced Topics and Spring JMS Video (Mark Richards, O'Reilly, 2014) (O'Reilly Shop, Safari Books Online)
- Java Message Service, 2nd Edition (Mark Richards, O'Reilly, 2009) (O'Reilly Shop, Safari Books Online)
- ActiveMQ Embedded Broker Messaging
- OpenMQ Embedded Broker Messaging
- HornetQ Embedded Broker Messaging

About the Author

Mark Richards is a hands-on software architect involved in the architecture, design, and implementation of Microservices Architectures, Service Oriented Architectures, and distributed systems in J2EE and other technologies. He has been involved in the software industry since 1983, and has significant experience and expertise in application, integration, and enterprise architecture. Mark served as the President of the New England Java Users Group from 1999 thru 2003. He is the author of several technical books and videos, including "Software Architecture Fundamentals" (O'Reilly video), "Enterprise Messaging" (O'Reilly video), and "Java Message Service 2nd Edition" (O'Reilly). Mark carries a masters degree in computer science, and has numerous architect and developer certifications from IBM, Sun, The Open Group, and BEA. He is a regular conference speaker at the No Fluff Just Stuff (NFJS) Symposium Series, and has spoken at over 100 conferences and user groups around the world on a variety of enterprise-related technical topics.