JBoss Seam 3 JMS Module

Reference Guide

by John Ament and Jordan Ganoff

Introduction

Seam extends the CDI programming model into the messaging world by allowing you to inject JMS resources into your beans. Furthermore, Seam bridges the CDI event bus over JMS; this gives you the benefits of CDI-style type-safety for inter-application communication.

1.1. Mission statement

The JMS module for Seam 3 is to provide injection of JMS resources and the necessary scaffolding for a bidirectional propagation of CDI event over JMS.

1.2. Seam 3 JMS Module Overview

The general goals can be divided into two categories: injection of JMS resources and bridging of events:

JMS Resource Injection

- ConnectionFactory
- Connection
- Session
- Topics & Queues
- Message Producer
- Message Consumer

Event Bridge

- Inbound: Routes CDI events to JMS destinations
- Outbound: Fires CDI events based on the reception of JMS messages

Installation

Seam JMS can be used by including a few libraries in your application's library folder:

- seam-jms-api.jar
- seam-jms.jar
- solder-api.jar
- solder-impl.jar
- solder-logging.jar

If you are using *Maven* [http://maven.apache.org/] as your build tool use the following dependency, which will bring in both API and Implementation for Seam JMS:

```
<dependency>
  <groupId>org.jboss.seam.jms</groupId>
  <artifactId>seam-jms</artifactId>
  <version>${seam.jms.version}</version>
</dependency>
```

Tip

Define or replace the property ${seam.jms.version} with a valid version of Seam JMS.

The runtime of Seam JMS is defined in two sections. The first section is related to creating observers, which happens within the Seam JMS CDI Extension. Observers need to be defined prior to starting up the container, and cannot be created once the application is running. This part happens automatically. The second section is related to creating listeners. This is managed in the component org.jboss.seam.jms.bridge.RouteBuilder.

Tip

In order to start any listeners, you may need to inject an instance of the RouteBuilder in to your class.

If you are running within a Servlet Container, and include the Solder, RouteBuilder will automatically start up.

The default implementation expects to find a ConnectionFactory at the JNDI location `/ConnectionFactory`. This can be changed by using Solder Config by

using a snippet similar to the one below in seam-beans.xml. This will change the JNDI location Seam JMS looks to `jms/ConnectionFactory`

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:s="urn:java:ee"
    xmlns:jmsi="urn:java:org.jboss.seam.jms.inject"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://docs.jboss.org/
cdi/beans_1_0.xsd">

    <jmsi:JmsConnectionFactoryProducer>
      <s:modifies />
                    <jmsi:connectionFactoryJNDILocation>jms/ConnectionFactory</
jmsi:connectionFactoryJNDILocation>
    </jmsi:JmsConnectionFactoryProducer>
  </beans>
```

Resource Injection

In this chapter we'll look at how to inject some of the common JMS resource types.

3.1. JMS Resource Injection

The following JMS resources are available for injection:

- javax.jms.Connection
- javax.jms.Session

Destination-based resources:

- javax.jms.Topic
- javax.jms.Queue

3.1.1. Destination Based Injection

The qualifier `@JmsDestination` is available to decorate JNDI oriented objects. This includes instances of `javax.jms.Destination` as well as MessageConsumers and MessageProducers.

```
@Inject @JmsDestination(jndiName="jms/MyTopic") Topic t;
@Inject @JmsDestination(jndiName="jms/MyQueue") Queue q;
```

3.1.2. Resource Configuration

You can use the `@JmsSession` annotation when injecting `javax.jms.Session` to specify transacted and acknowledgement mode:

```
@Inject @JmsSession(transacted=false, acknowledgementMode=Session.CLIENT_ACKNOWLEDGE) Session s;
```

3.2. Module Extensions

The Seam JMS module has certain points of extension, where the application developer can customize the behavior to match their needs. This is done by extending any of three base classes:

- org.jboss.seam.jms.inject.JmsConnectionFactoryProducer

- org.jboss.seam.jms.inject.JmsConnectionProducer
- org.jboss.seam.jms.inject.JmsSessionProducer

This can be done using CDI specializations and extending the base class to change the produced object. This allows the application developer to customize the produced behavior. For example, the base implementation assumes a Java EE container, however an extension to the JmsConnectionFactoryProducer could bootstrap a JMS container for you or change the default JNDI location of the `ConnectionFactory`

Each producer in these classes generates an instance based on The `@JmsDefault` annotation. This object is used within other places of the API, so you can control the Session generated that is injected into a MessageManager this way.

Messaging API

The Seam JMS Messaging API is a higher level abstraction of the JMS API to provide a number of convenient methods for creating consumers, producers, etc.

4.1. QueueBuilder and TopicBuilder

The `QueueBuilder` and `TopicBuilder` interfaces are meant to ease the integration of JMS while still sticking close to the base APIs. Within the single class you can work with both listeners and send messages. References to these classes can be injected.

Some example usages.

```
@RequestScoped
@Named
public class FormBean {
  private String formData;
  @Inject QueueBuilder queueBuilder;
  @Inject TopicBuilder topicBuilder;
  @Resource(mappedName="jms/SomeQueue") Queue someQueue;
  @Resource(mappedName="jms/SomeTopic") Topic someTopic;
  @Resource(mappedName="jms/ConnectionFactory") ConnectionFactory cf;

  public void sendFormDataToQueue() {
     queueBuilder.connectionFactory(cf).destination(someQueue).sendString(formData);
  }

  public void sendFormDataToTopic() {
     topicBuilder.connectionFactory(cf).destination(someTopic).sendString(formData);
  }
}
```

It is strongly recommended that you proxy the injection of the builders to avoid repeating your configuration. If you are often times connecting to the same queues/topics, you can provide your own producer method.

```
public class OrderTopicProducer {
  @Inject BuilderFactory factory;
  @Resource(mappedName="jms/OrderTopic") Topic orderTopic;
  @Resource(mappedName="jms/ConnectionFactory") ConnectionFactory cf;
  @Produces @OrderTopic
```

```
  public TopicBuilder sendFormDataToQueue() {
     return factory.newTopicBuilder().connectionFactory(cf).destination(orderTopic);
  }

}
```

4.2. Message Manager

The `MessageManager` interface (`org.jboss.seam.jms.MessageManager`) is the main consolidated API for Seam JMS. It provides almost all of the background functionality for Seam JMS's features (Observer Interfaces, Routing API). The default implementation works against `javax.naming.Context` assuming running within the same local application server.

```
public interface MessageManager {
   public ObjectMessage createObjectMessage(Object object);
   public TextMessage createTextMessage(String string);
   public MapMessage createMapMessage(Map<Object,Object> map);
   public BytesMessage createBytesMessage(byte[] bytes);
   public void sendMessage(Message message, String... destinations);
   public void sendObjectToDestinations(Object object, String... destinations);
   public void sendTextToDestinations(String string, String... destinations);
   public void sendMapToDestinations(Map map, String... destinations);
   public void sendBytesToDestinations(byte[] bytes, String... destinations);
   public void sendMessage(Message message, Destination... destinations);
   public void sendObjectToDestinations(Object object, Destination... destinations);
   public void sendTextToDestinations(String string, Destination... destinations);
   public void sendMapToDestinations(Map map, Destination... destinations);
   public void sendBytesToDestinations(byte[] bytes, Destination... destinations);
   public Session getSession();
   public MessageProducer createMessageProducer(String destination);
   public TopicPublisher createTopicPublisher(String destination);
   public QueueSender createQueueSender(String destination);
   public MessageConsumer createMessageConsumer(String destination, MessageListener... listeners);
   public MessageConsumer createMessageConsumer(Destination destination, MessageListener... listeners);
   public TopicSubscriber createTopicSubscriber(String destination, MessageListener... listeners);
  public QueueReceiver createQueueReceiver(String destination, MessageListener... listeners);
}
```

The interface above defines a full set of capabilities for creating and sending messages. In addition, we expose methods for creating producers (and A destination specific publisher and sender) as well as consumers (and A destination specific subscriber and receiver). In addition,

if injected within a session scoped object, or similar, you can define a durable subscriber and unsubscriber for that subscriber. Below is an example.

The durable subscriber pattern works very well for session based message management. If you want to define a durable subscriber per user session, this is the easiest way to do it.

```
@SessionScoped
        public class UserSession {
           @Inject MessageManager messageManager;
           @Inject MySessionJMSListener listener;
           private String clientId;
           @PostConstruct
           public void registerListener() {
              clientId = UUID.randomUUID().toString();
              messageManager.createDurableSubscriber("jms/UserTopic",clientId,listener);
           }
           @PreDestroy
           public void shutdownListener() {
              messageManager.unsubscribe(clientId);
           }
        }
```

4.3. Durable Messaging Capabilities

Seam JMS provides a Messaging API around the JMS Durable Topic Subscriber concept. In order to use it within your code, you need to inject a DurableMessageManager.

```
@Inject @Durable DurableMessageManager durableMsgManager;
```

This implementation of `MessageManager` provides additional methods to first login to the connection with a `ClientID`, additional methods to create subscribers and an unsubscribe that can be called to unsubscribe a listener.

```
public void login(String clientId);
public TopicSubscriber createDurableSubscriber(String topic, String id, MessageListener... listeners);
public TopicSubscriber createDurableSubscriber(Topic topic, String id, MessageListener... listeners);
public void unsubscribe(String id);
```

Tip

From a design pattern standpoint, it makes sense to create an `ApplicationScoped` object that all subscribers are created from, injecting a `DurableMessageManager` for use across the application, producing `SessionScoped` sessions for use by clients.

4.4. MessageListeners versus Message Driven Beans

One of the difficult choices we had to make was support for Message-Driven Beans. MDBs are a little complicated in CDI as they are not managed within the CDI life cycle. This makes integration with them a bit cumbersome. We wouldn't be able to work with a JMS Session in these cases, as an example. As a result, Seam JMS only supports defining instances of `javax.jms.MessageListener`. To support this, we have created a partial implementation - `org.jboss.seam.jms.AbstractMessageListener`. This special MessageListener is designed for bridging the external context of a JMS Message into the application you are working with. We do this by tweaking classloaders.

The best way to work with MessageListeners is to simply instantiate your own based on our base implementation.

```
//where cl is the class loader, and beanManager is the BeanManager
MessageListener ml = new SessionListener(beanManager,cl,this);
messageManager.createTopicSubscriber("/jms/myTopic", ml);
```

Or you may define your own subclass that makes specific invocations to the parent. Here is an example of that:

```
@SessionScoped
    public class SessionListener extends AbstractMessageListener {
       private MySessionObject mso;
       public SessionListener(BeanManager beanManager, ClassLoader classLoader,
          MySessionObject mso){
          super(beanManager,classLoader);
          this.mso = mso;
       }
```

```
    @Override
    protected void handlMessage(Message msg) throws JMSException {
        //your business logic goes here
    }
}
```

Bridging the Gap

This chapter is designed to detail how to configure the CDI to JMS event bridge. Routing has two sides, sending of events to JMS destinations and translating received messages from JMS destinations back into CDI events. The sections of this chapter describe how to achieve both.

5.1. Event Routing

Simply sending or receiving a message over JMS involves a few players: Connection, Session, Destination, and the message itself. Surely you can inject all required resources and perform the routing yourself but that takes away from the whole reason you're using a tool in the first place!

5.1.1. Routes

Routing CDI events to and from JMS can be configured by defining a `Route`. As you would normally create an observer method for an event you can define a route to control which events get forwarded to what destination. Or conversely, what message types sent to which destinations generate CDI events.

```
public interface Route {
  public <D extends Destination> Route connectTo(Class<D> d, D destination);
  public Route addQualifiers(Annotation... qualifiers);

  ...
}
```

Routes allows for simple mapping of event types, complete with qualifiers, to a set of destinations. They can be configured by adding qualifiers and providing destinations they should interact with and are created from a `RouteManager`. Here's a simple route that forwards CDI events on to a queue:

```
@EventRouting public Route registerMyRoute(RouteManager routeManager)
{
   Queue myQueue = lookupQueue("/jms/MyQueue");
   return routeManager.createRoute(RouteType.EGRESS, MyEvent.class).connectTo(Queue.class, myQueue);
}
```

A `RouteManager` is a factory object for creating new Routes. An instance of it is injected into every `@EventRouting` method. Classes with methods that are decorated with `EventRouting` must meet a few criteria items:

- A default, no arg constructor.
- Be a non bean (no dependencies on injection)
- Return either `Route` instances or `Collection<Route>` instances.

These requirements exist because of when the generation of `Routes` must happen. There are no CDI beans active within the context. A class identified for routing will automatically be veto'd from the context.

Routes are registered by returning them from a non-bean method annotated with `@EventRouting`:

```
@EventRouting public Route myConfig()
{
   return bridge.createRoute(RouteType.INGRESS, MyEvent.class).addDestinationJndiName("/jms/MyTopic");
}
```

5.2. Routing CDI Events to JMS

Forwarding CDI events to JMS is configured by creating an egress route. Let's say you wanted to forward all `MyEvent` events with `@Bridged` qualifier to the queue `jms/EventQueue`. Simple, register a route:

```
AnnotationLiteral<Bridged> BRIDGED = new AnnotationLiteral<Bridged>() {};
@EventRouting public Route registerMyEventRoute(RouteManager routeManager)
{
routeManager.createRoute(RouteType.EGRESS, MyEvent.class).addQualifiers(BRIDGED).addDestinationJndiName("/jms/EventQueue");
}
```

5.2.1. Usage

With your routing defined you can simply fire events that match the route's payload type and qualifiers and these events will be forwarded over JMS as object messages. A special note,

we have added the qualifier `@Routing(RouteType.EGRESS)`. This is necessary to avoid circular routings.

```
@Inject @Bridged @Routing(RouteType.EGRESS) Event<MyEvent> event;
...
event.fire(myEvent);
```

5.3. CDI Events from JMS Messages

Similar to egress routes, ingress routes are defined the same way. In this case, they listen for messages on the specified destination(s) and fire events. All of the data will be type safe, assuming you have defined your routes correctly.

Similar to the above example, this creates ingress routes from the Queue jms/EventQueue and fires events based on the MyEvent objects that are carried over the wire.

```
AnnotationLiteral<Bridged> BRIDGED = new AnnotationLiteral<Bridged>() {};
@EventRouting public Route registerMyEventRoute(RouteManager routeManager)
{
routeManager.createRoute(RouteType.INGRESS, MyEvent.class).addQualifiers(BRIDGED).addDestinationJndiName("/
jms/EventQueue");
}
```

5.3.1. Usage

Once you define an ingress route, you handle it using an observer method. We use the same payload type and qualifiers, however we need to add the same qualifier, but for ingress `@Routing(RouteType.INGRESS)`

```
public void handleInboundMyEvent(@Observes @Routing(RouteType.INGRESS) MyEvent e) {
  ....
}
```

Annotation Routing APIs

This chapter is meant to describe the behavior of mapping interfaces, where event mapping to data flowing through JMS Queues and Topics are handled via events. These APIs are an alternate way to define routes as mentioned earlier in the document.

6.1. Observer Method Interfaces

Observer Method Interfaces are simple Plain Old Java Interfaces (POJIs) that define either a route. These interfaces exist within your code and are read at deployment time. This is a sample interface:

```
public interface MappingInterface {
    @Inbound
public void routeStringsFromTopic(@Observes String s, @JmsDestination(jndiName="jms/MyTopic") Topic t);

    @Outbound
public void routeLongsToQueue(@Observes Long l, @JmsDestination(jndiName="jms/MyQueue") Queue q);

public void bidirectionRouteDoublesToQueue(@Observes Double d, @JmsDestination(jndiName="jms/DblQueue") Queue q);
}
```

This interface defines three routes. The first one being an ingress route - messages coming in to the topic jms/MyTopic will be fired as events with the type String. We indicate this by using the @Inbound annotation or @Routing(RouteType.INGRESS). The second being an egress route - events fired of type Long will be turned into ObjectMessages and using a MessageProducer sent to the queue jms/MyQueue. We indicate this by using the @Outbound annotation or @Routing(RouteType.EGRESS). The last is a bidirectional route, it defines messages that get fired in both directions. You can leave the method unannotated or use the @Routing(RouteType.BOTH) annotation.

The object being observed can have qualifiers. These qualifiers will be carried over in the fired event and follow the CDI rules for observer method selection. In all cases, the return type of the method is ignored.

The destinations can have any qualifier. In addition, there is basic support for @Resource on the method level to define the destination. This in general not 100% portable from the application developer perspective, we recommend heavy testing of the behavior on your application server.

In order to work with these routes, you raise events in CDI. In order to fire an event, first inject the `Event` object into your code with necessary annotations, for any egress route. For ingress routes, you need to define an observer method. Taking the third route as an example, here is how you would raise events to it

```
@Inject @Outbound Event<Double> doubleEvent
...
doubleEvent.fire(d);
```

and this is the appropriate observer method to handle the incoming messages.

```
public class MyEventObserverBean {
 public void sendMessage(@Observes @Inbound Double d) {
    System.out.println(d);
 }
}
```