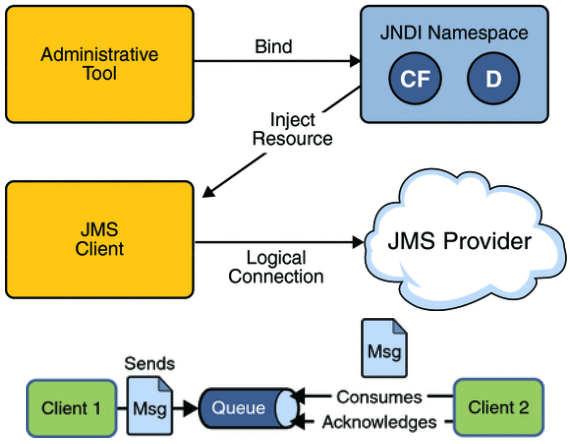
7. Java Messaging Service (JMS)

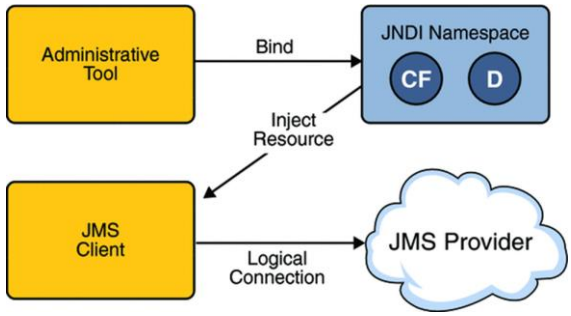
DGP Solutions

JMS High Level Architecture



Administered Objects

- A **connection factory** is the object a client uses to create a connection to a provider.
- A **destination** is the object a client uses to specify the target of messages it produces and the source of messages it consumes i.e. a Queue



19920612 Copyright DGP Solutions 7-3

Other Objects

- A **connection** encapsulates a virtual connection with a JMS provider. Created from the **connection factory**
- A **session** is a single-threaded context for producing and consuming messages. Created from the **connection**
- A **message producer** is an object that is created from a **session** and used for sending messages to a destination
- A **message consumer** is an object that is created by a session and used for receiving messages sent to a destination
- A **message listener** is an object that acts as an asynchronous event handler to poll for messages on a destination
- **JMS Message** is an object that represents the message itself whose payload could be Test or a Serializable Object

JmsTemplate

- Spring Boot hides all these complexities of object creation and interaction behind the class **org.springframework.jms.core.JmsTemplate** and with one maven dependency create an embedded ActiveMQ broker

- JmsTemplate simplifies the use of Java Messaging service (JMS) and gets rid of boilerplate code. It handles the creation and release of JMS resources when sending or receiving messages.


```
<!-- Embedded ActiveMQ -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

Spring Boot and ActiveMQ - Producer

```
@Autowired private JmsTemplate ;

private static Logger = Logger.getLogger("HealthPlanController");

@PostMapping(path="/domestic", consumes=MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Long> addToQueue(@RequestBody HealthPlan plan)  {
       logger.log(Level.INFO, "Received message");
       jmsTemplate.convertAndSend("myQueue", plan);
       logger.log(Level.INFO, "Sent message");
       return ResponseEntity.ok().build();
}
```

- ActiveMQ must be told what objects to safely deserialize (String OK)
- In application.properties add the key value pair of;
  - spring.activemq.packages.trust-all=true
  - OR spring.activemq.packages.trust=package-name,package-name

The application.properties file is used by *Spring Boot* to configure the *ConnectionFactory* needed by a listener

Spring Boot and ActiveMQ - Consumer

```
@Autowired private HealthPlanRepository repo;
private static Logger = Logger.getLogger("MessageConsumer");

@JmsListener(destination = "myQueue")
public void receiveMessage(Message msg) throws JMSException {
    ObjectMessage objMsg = (ObjectMessage) msg;
    HealthPlan plan = (HealthPlan) objMsg.getObject() ;
    logger.log(Level.INFO, "Received Message for Plan " + plan);
    HealthPlan pl = repo.save(plan);
    logger.log(Level.INFO, "Repository updated with Plan " + pl.getId());
}
```

- Note the destination name matches where the JmsTemplate put the message

Client is not kept waiting

- Upon invoking our Rest Endpoint, our client is notified of a successful call immediately

```
<200,[Content-Length:"0", Date:"Fri, 31 Jan 2020 18:34:21 GMT", Keep-Alive:"timeout=60", Connection:"keep-alive"]>
```

- We can then observe the logs to see the production and consumption of the message

```
HealthPlanController        : Received message
HealthPlanController        : Sent message
MessageConsumer             : Received Message for Plan HealthPlan [id=0, zip=77777, name=HMO, deductableIndividual=2500,
deductableFamily=3000, outOfPocketIndividual=5000, outOfPocketFamily=5500, copay=30.0]
MessageConsumer             : Repository updated with Plan 22
```

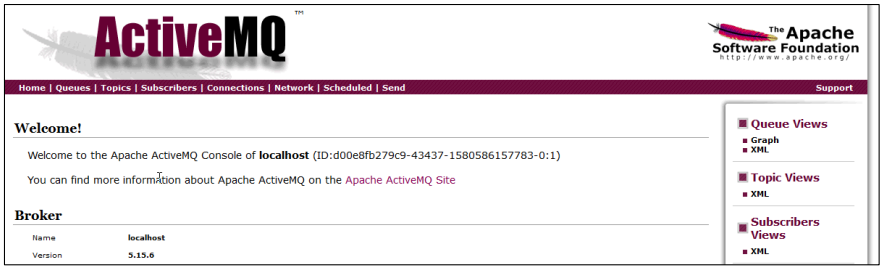
Using an External Broker

- We will use a **docker image** for our external ActiveMQ broker (https://hub.docker.com/r/rmohr/activemq).
- In PowerShell we pull the image from Docker hub.
  - $ docker pull rmohr/activemq (this can take some time)
- Then we run the container starting our ActiveMQ broker
  - $ docker run -p 61616:61616 -p 8161:8161 rmohr/activemq
- The broker listens on port 61616 (the port for our java clients – producer and consumer) and its web console is on port 8161
  - Check that the container is up
  - $ docker ps

```
d00e8fb279c9        rmohr/activemq      "/bin/sh -c 'bin/act…"   6 minutes ago       Up 5 minutes
, 0.0.0.0:8161->8161/tcp, 61613-61614/tcp, 0.0.0.0:61616->61616/tcp   sweet_lovelace
```


ActiveMQ Web Console

- Our docker container contains our ActiveMQ broker.
- If we navigate to http://localhost:8161/admin/ (the port was in our run command for the container), we see the ActiveMQ Web Console
  - Username is “admin” and Password is “admin”



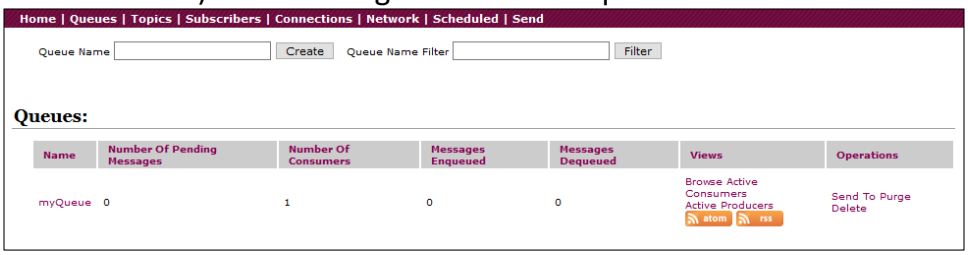
Updated configuration

- In our Spring boot Application we must register the url to the external broker (In our docker run command we used 61616.
- The application.properties is used by Spring Boot to configure some basic objects, such as the ConnectionFactory needed by a listener that points to a running broker.
- It now needs to write to an external broker rather than embedded broker, so we need to add the url of the new broker.

```
spring.activemq.packages.trust-all=true
spring.activemq.broker-url=tcp://localhost:61616
```


Create and register a Consumer to the Queue

- If we launch our Spring Boot Application, we will see in the “Queues” tab of the Brokers Web console, that a Queue has been created. It has created a queue with the queue name that are java code was using in the Spring boot Application. We also see that our consumer (Spring Boot Listener) has been registered to that queue

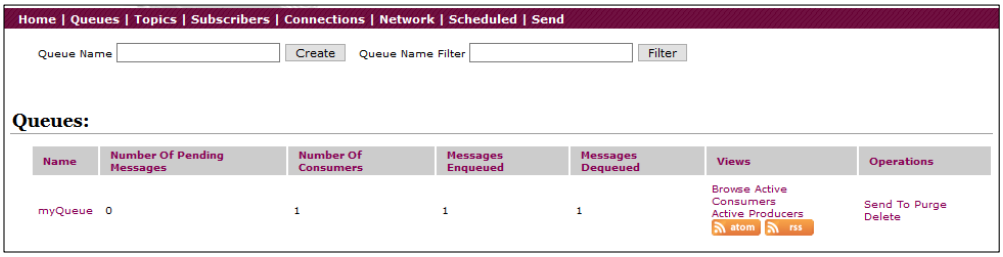


19920612 Copyright DGP Solutions 7-12

- We could create a queue here in ActiveMQ if we want. Just make sure our java code reflects that queue name.
- Messages Enqueue" - is the number of messages that have been produced on to the queue
- Messages Dequeued" - is the number of messages that have been consumed from the queue

Consume a message

- Accessing the RestController endpoint that is also the message producer, a message is placed on the queue and is subsequently consumed by our java listener. The Web console confirms the interaction, as do the application logs.

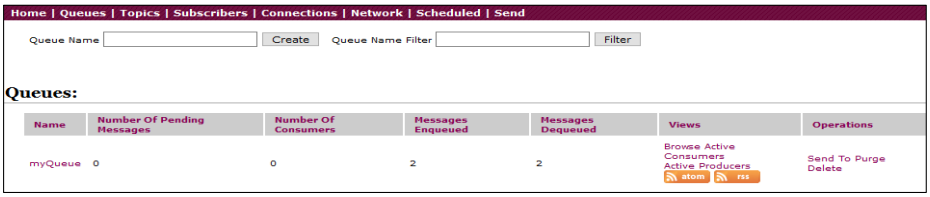


19920612 Copyright DGP Solutions 7-13

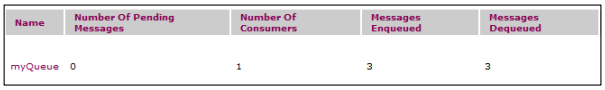
HealthPlanController : Received message
HealthPlanController : Sent message
MessageConsumer : Received Message for Plan HealthPlan [id=0, zip=77777, name=HMO, deductableIndividual=2500, deductableFamily=3000, outOfPocketIndividual=5000, outOfPocketFamily=5500, copay=30.0]
MessageConsumer : Repository updated with Plan 22

Broker Queue Lifecycle

- Terminating the Spring boot Application, unregisters the consumer but the queue remains in the broker



- When we start our Spring Boot Application, it will now use the existing queue



Broker Queue Lifecycle cont.

- If we stop the container and restart it
  - $ docker stop containerId
  - $ docker start containerId
- The queue is once again present, available for producers and consumers

Queues:

Name	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued
myQueue	0	0	0	0