Modern Web Development for Java Programmers

Unit 9. JNDI and Messaging with JMS

Unit 9 Timeline

Walkthrough 1	10 min
Intro to JNDI	15 min
Walkthrough 2	10 min
Messaging terms and concepts	10 min
Bringing Messaging in Auction	15 min
Break	10 min
Configuring Admin objects in JNDI/HornetQ	15 min
Walkthrough 3	15 min
Walkthrow 4	10 min

The Cast

- Java EE 7 App Server - JBoss Wildfly 8
- Messaging server: HornetQ embedded in Wildfly
- JMS admin object are published as JNDI resources
- Message producers: Java client
- Message consumers: Message-Driven Beans
- Monitoring Tool: Wildfly Administrator Console

Walkthrough 1(start)

- The goal of this walkthrough is to run Wildfly Admin Console.

- Start the server (In IDEA, select Run | Edit Configuration…). Watch http://vimeo.com/91668238, but don't add any artifacts.

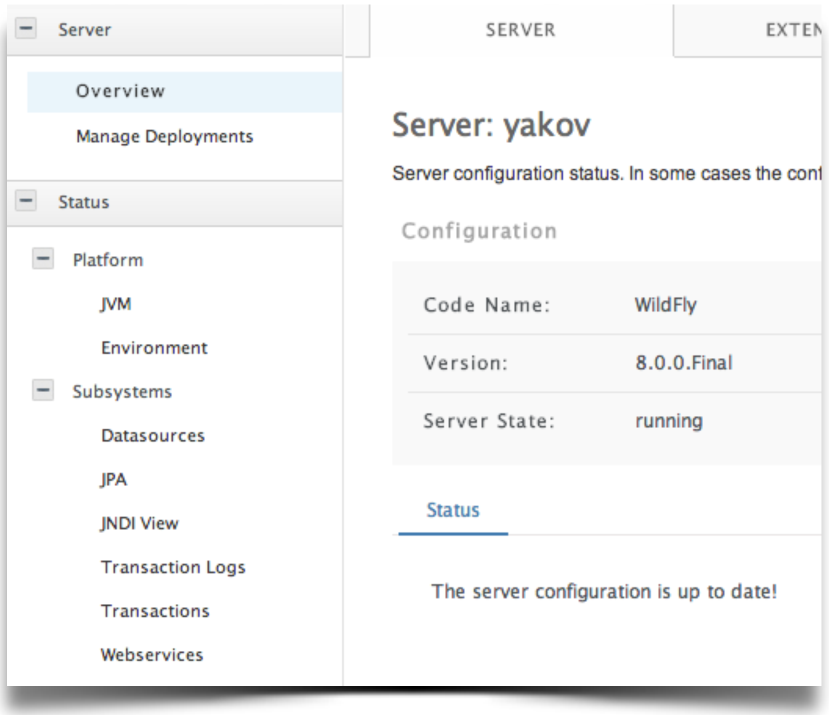
- Enter http://localhost:8080 in your Web browser. In the Welcome page click on Administration Console. You'll see a message that you have not added any users yet.

- From command window run add-user script from Wildfly bin folder. Add the Management User named *FARATA* with password *farata1$*. This user is added to ManagementRealm, does not belong to any groups, and won't be used for AS to AS connections.

- Connect to Administration Console.

Walkthrough 1(end)

Login and you'll see the console.



Server: yakov

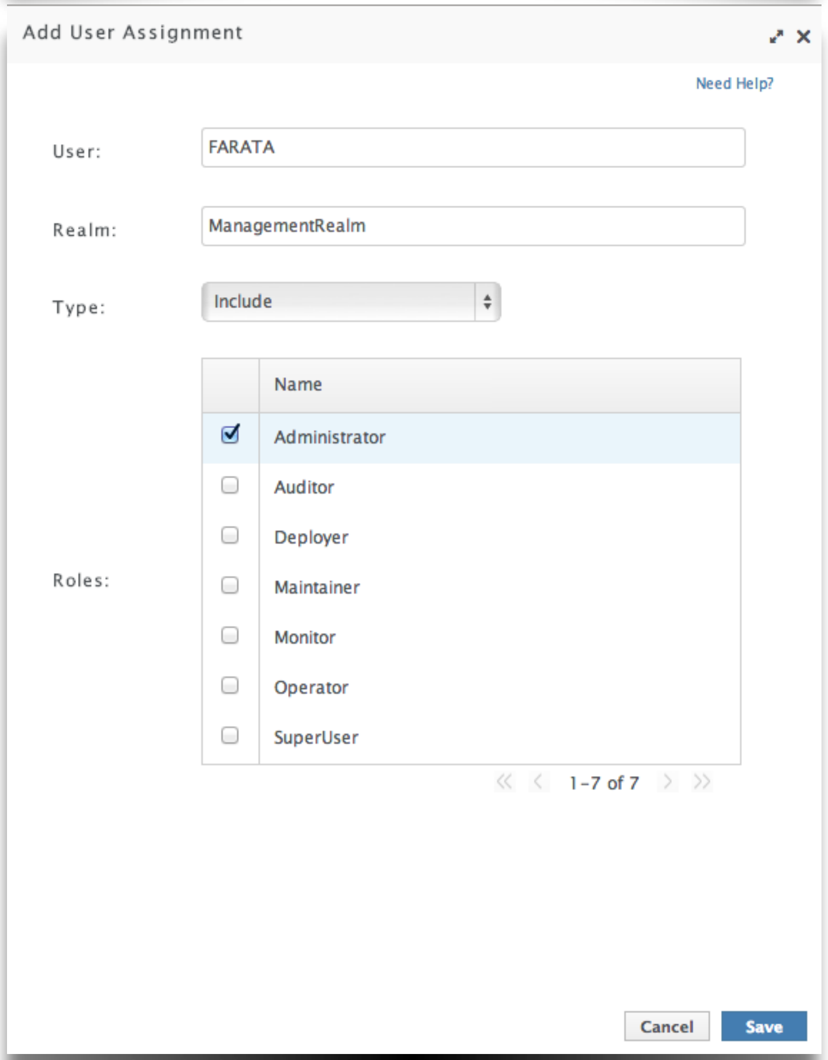
Server configuration status. In some cases the conf

Configuration

Code Name:	WildFly
Version:	8.0.0.Final
Server State:	running

The server configuration is up to date!

In Administration tab make FARATA user administrator at ManagementRealm



User: FARATA

Realm: ManagementRealm

Type: Include

Roles:

	Name
✓	Administrator
	Auditor
	Deployer
	Maintainer
	Monitor
	Operator
	SuperUser

1-7 of 7

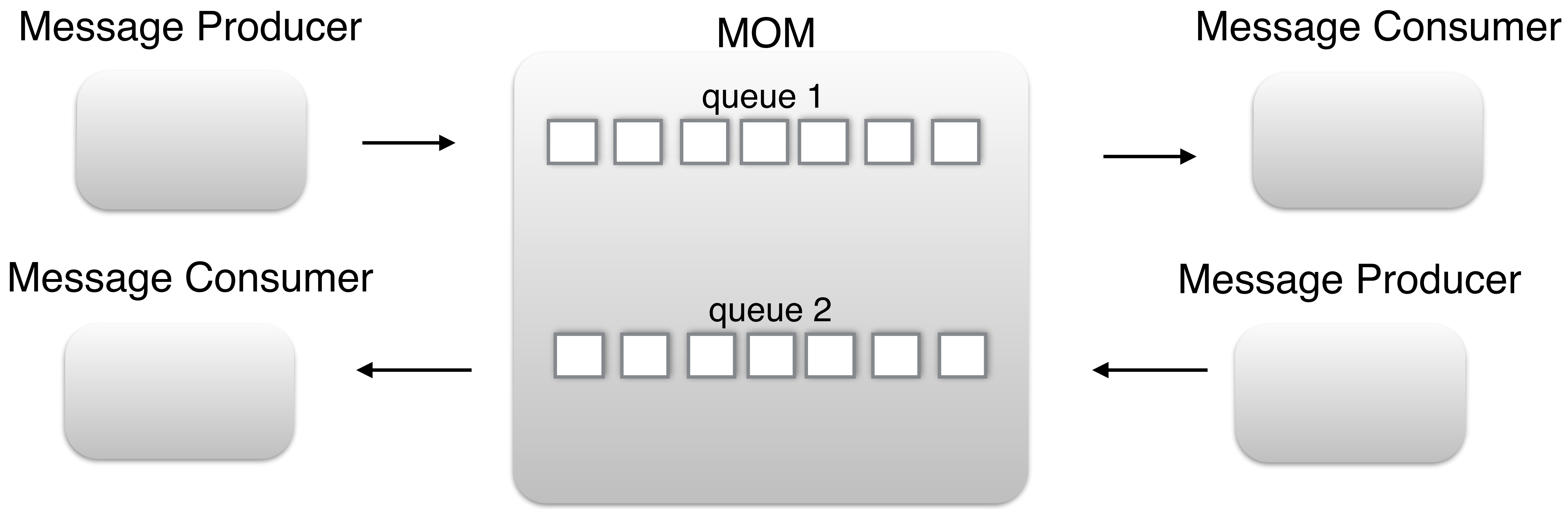
Messaging Concepts

MOM and JMS

- Message Oriented Middleware (MOM) is a transport for messages, e.g. WebSphereMQ, ActiveMQ, HornetQ, Soniq MQ, et al.
- JMS stands for Java Messaging Service.
- JMS is an API for working with one of the MOM servers.
- MOM allows you to build losely coupled distributed systems.

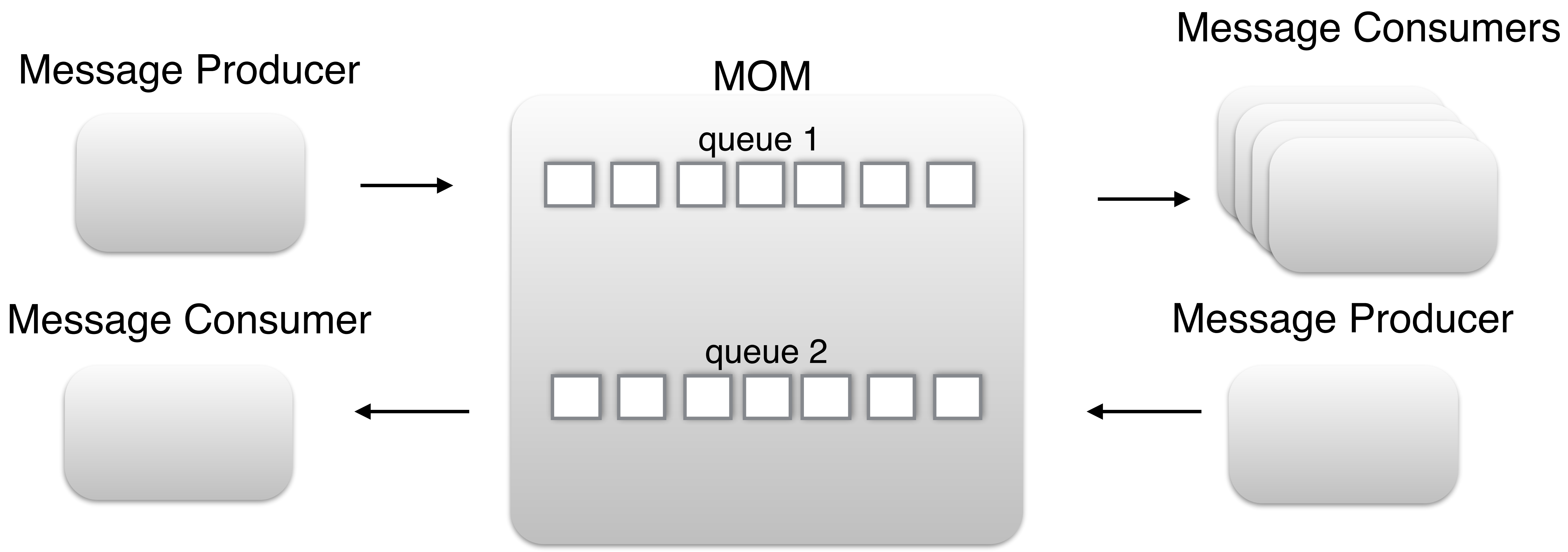
WildFly 8 Comes with embedded HornetQ MOM

Point to Point Messaging



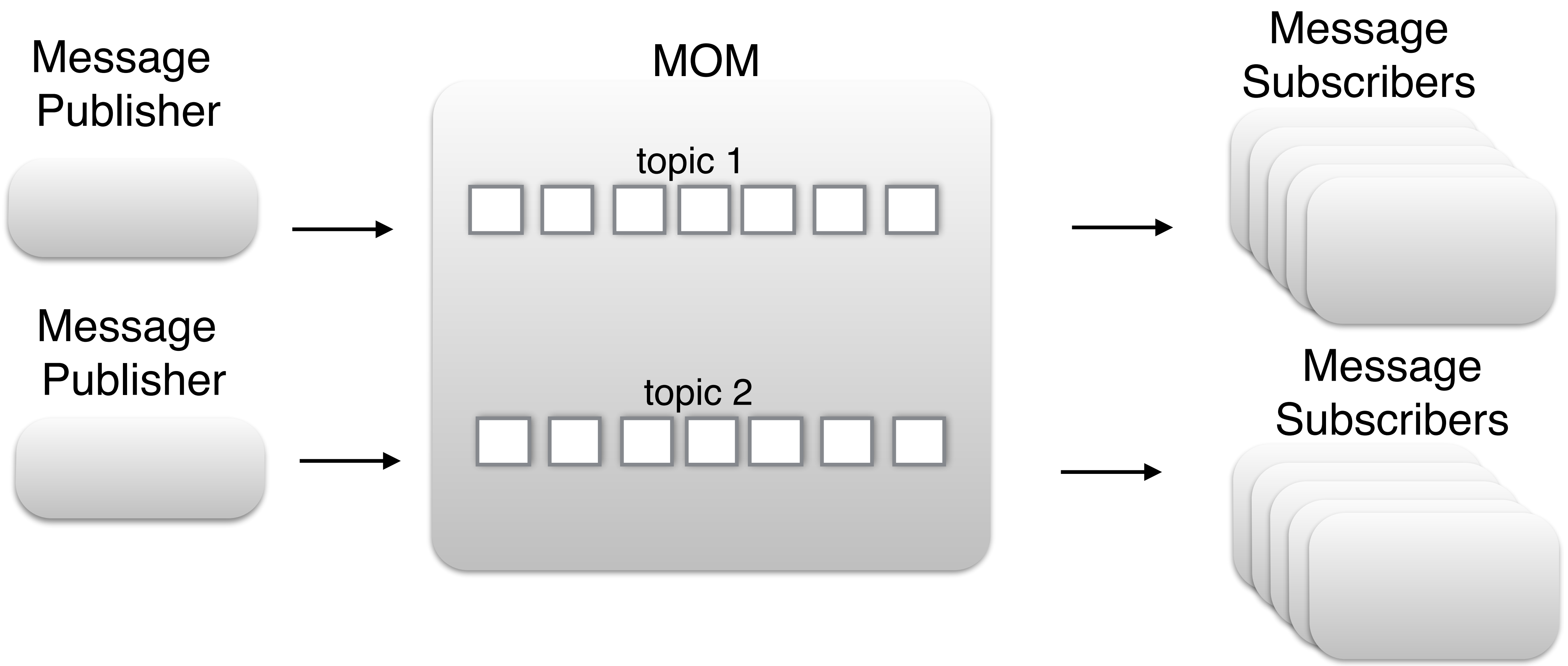
Each message goes to only one consumer.

Point to Point Messaging



Each message goes to only one consumer even if is there are multiple consumers.

Publish-Subscribe Messaging



Each message may go to multiple subscribers (consumers).

What’s new in JMS 2.0 (JSR 343)

- JMS 2.0 has simplified API, but the old JMS 1.1 code will still work.
- `JMSContext` encapsulates both `Connection` and `Session`. It implements `AutoCloseable.`
- Asynchronous send mode
- Inject with CDI
- `JMSProducer`, `JMSConsumer`
- `JMSExeption` is replaced with `JMSRuntimeException`
- Can schedule future message delivery
- Shared Subscribtions

JNDI Basics

Java Naming and Directory Interface

Naming servers make are registries of objects.

JNDI helps Java objects in finding required resources (e.d. data source, message queue, etc.)

Every Java app server runs internal JNDI server

Administrator *binds* resources to the names in the JNDI tree. This is done via Admin Console, using scripts, or XML deployment descriptors.

Java Naming and Directory Interface

- JNDI `InitialContext` is the root of JNDI tree
- If your Java code runs inside Java EE server, it can *inject* the entries from JNDI to your code using @Resource annotation.
- Your program can also run a `lookup()` on JNDI tree to find resources.
- Remote Java programs can only run `lookup()` to find the objects.

Getting InitialContext

- Java program inside the app server:

```
Context namingContext = new InitialContext();
```

- Java program outside of the app server:

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,"org.jboss.naming.remote
.client.InitialContextFactory");
env.put(Context.PROVIDER_URL, “http-remoting://127.0.0.1:8080");
env.put(Context.SECURITY_PRINCIPAL, “Alex123”);
env.put(Context.SECURITY_CREDENTIALS, “MySecretPwd”;

Context namingContext = new InitialContext(env);
```

JMS Administered Objects

- JMS destinations (queues, topics) and connection factorues are typically maintained by adminitrators.
- Administrators configure (bind) administered objects to naming servers (JNDI, LDAP).
- Connection factory provides connectivity to MOM server.
- Connection factory is an instance of `ConnectionFactory`, `QueueConnectionFactory`, `TopicConnectionFactory`
- Destinations are instances of `Topic` or `Queue`.

Resource Lookup

- Finding JMS Connection factory:

```
ConnectionFactory connectionFactory = (ConnectionFactory)
         naamingContext.lookup(connectionFactoryString);
```

- Finding a JMS destination (e.g. a msg queue *test*):

```
Destination destination = (Destination)
                    namingContext.lookup(“jms/queue/test");
```

Resource Injection

- Injection decouples your code from implementation of its dependencies

- Resource injection allows to inject any **JNDI resource** into a container-managed object, e.g. servlet, ejb, REST endpoint.

```
@Resource(name=“java:comp/DefaultDataSource)
private javax.sql.DataSource myDataSource;
```

```
@Resource(lookup ="java:/ConnectionFactory")
    ConnectionFactory connectionFactory;

    @Resource(lookup = "queue/test")
    Queue testQueue;
```

As opposed to Resource Injection, CDI allows to inject any managed Java object with `@Inject`

Walkthrough 2 (start)

- The goal of this walktrough is to create an application user and a message queue named jms/queue/test in Wildfly's JNDI tree.

- Follow instructions from Reame.md at Wildfly helloworld-jms sample: http://goo.gl/X2uzpY . Ignore Maven instructions.

a) Stop Wildfly server if it's running. Start it in stanalone full configuration. In IDEA,go to Run | Edit Configurations | Startup/Connection. Uncheck Use Default for startup script. Add *-c standalone-full.xml* to startup command.

b) Configure JMS by running the JBoss CLI Script. This will bind JMS queue testQueue to JNDI.

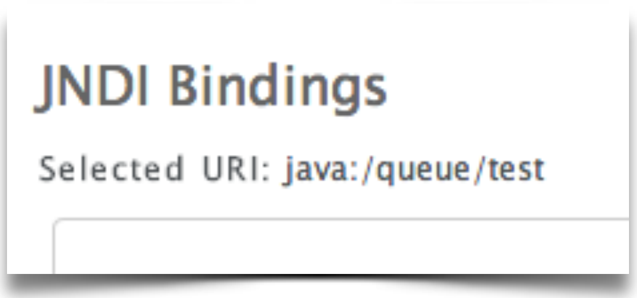
c) Add Application user according to these instructions http://goo.gl/ym1mYy

Walkthrough 2 (cont)

- Login to Wildfly's admin console as user FARATA at http://localhost:9990/console.

- Browse the content of JNDI View. Select the queue section.

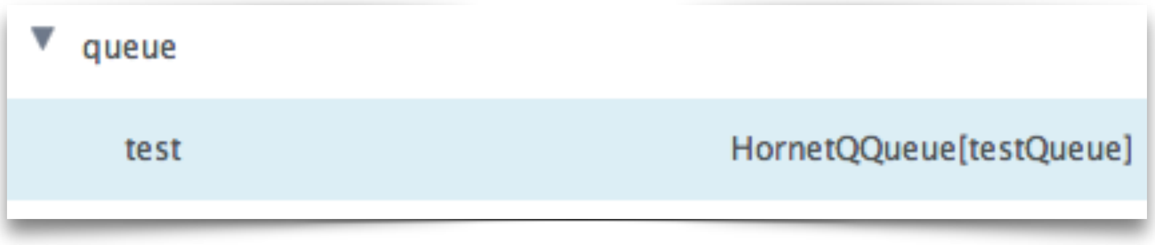
- Note that the the name used for JNDI binding is java:/queue/test



JNDI Bindings

Selected URI: java:/queue/test

- Note that the physical name of the queue is **testQueue**, and its type is `HornetQQueue` (see HornetQ javadoc http://goo.gl/11yKPg) .

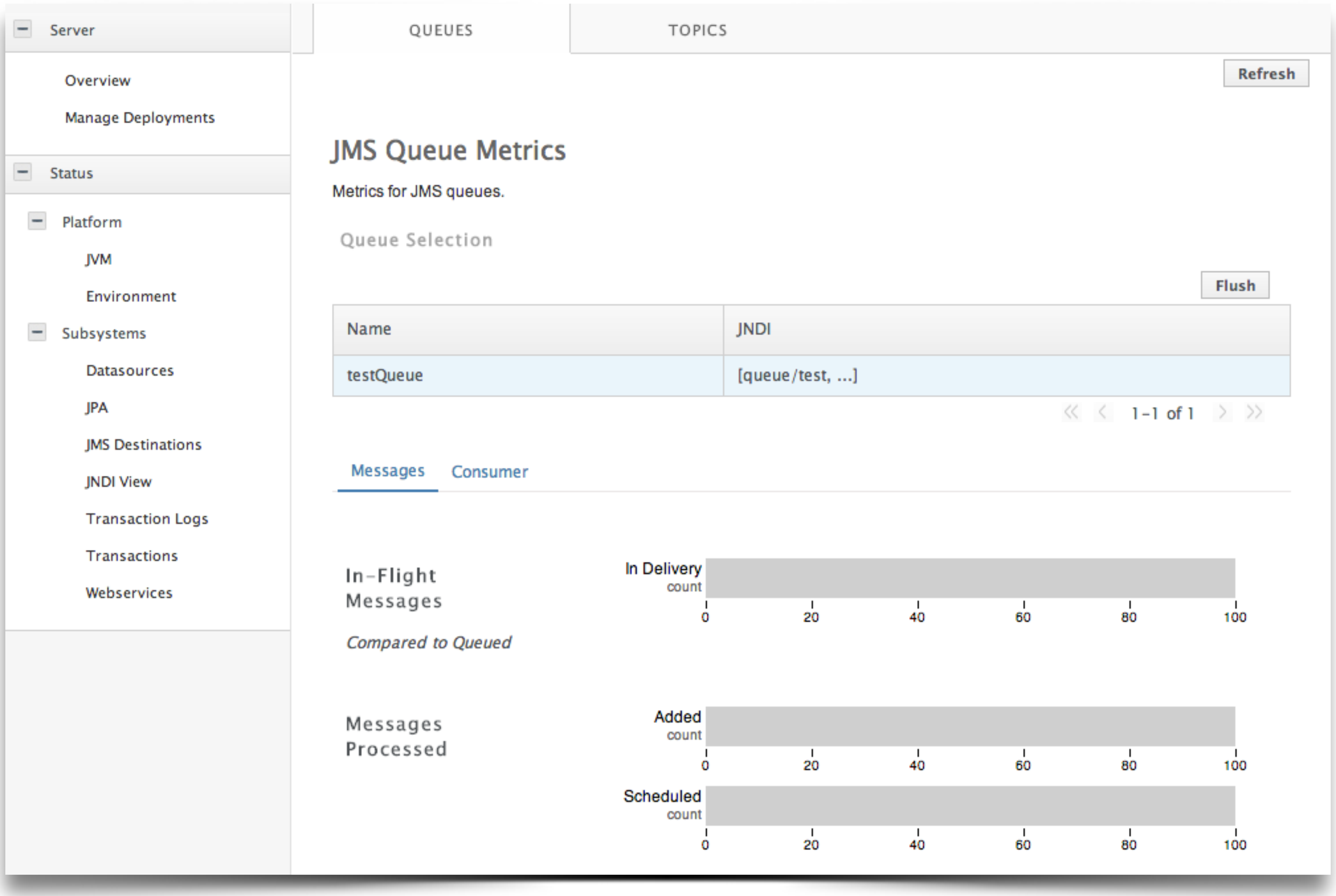


▼ queue

test HornetQQueue[testQueue]

Walkthrough 2 (end)

- Click on the testQueue under JMS destinations. you should see something like this:



JMS Queue Metrics

Metrics for JMS queues.

Queue Selection

Name	JNDI
testQueue	[queue/test, ...]

1-1 of 1

In-Flight Messages

Compared to Queued

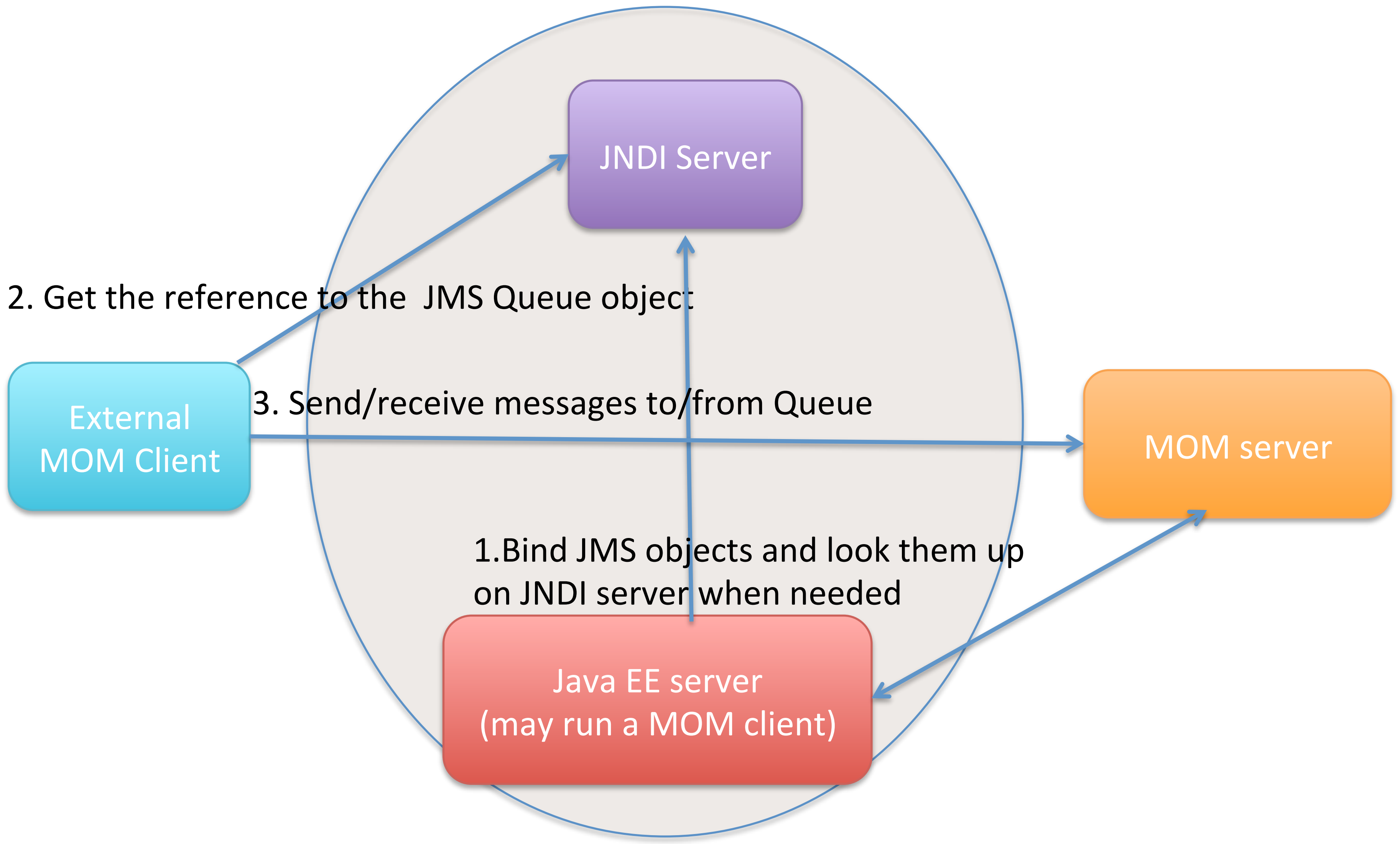
In Delivery count: 0 20 40 60 80 100

Messages Processed

Added count: 0 20 40 60 80 100

Scheduled count: 0 20 40 60 80 100

Naming, Messaging and App Servers


JNDI Server
2. Get the reference to the JMS Queue object
External MOM Client
3. Send/receive messages to/from Queue
MOM server
1.Bind JMS objects and look them up on JNDI server when needed
Java EE server (may run a MOM client)

To replace MOM, just rebind new admin objects (queues, topics) to JNDI server (e.g. LDAP)

JMSContext

- `ConnectionFactory` creates `JMSContext`.
- `JMSContext` combines connection and a session objects
- Create `JMSContext` using try-with-resources:

```
try (JMSContext context = connectionFactory.createContext(userName,
password))
```

- `JMSContext` creates message producers, consumers and messages:

```
JMSProducer producer = context.createProducer();
producer.send(destination, content);

JMSConsumer consumer = context.createConsumer(destination);
```

Walkthrough 3 (start)

- The goal is to run a remote client that sends/receives messages to HornetQ embedded in Wildfly. This walkthrough is based on the Wildfly Quickstart example published at https://github.com/wildfly/quickstart/tree/master/helloworld-jms - download this example.

- Create a new empty IDEA project and add a new Java module to it. In Dependencies tab add jboss-client.jar from your Wildfly's *bin/client* directory.

- Copy just the file HelloWorldJMSClient.java to the *src* folder.

- Your Wildfly server should be running in standalone full configuration as explained in Walkthrough 2.

Walkthrough 3 (cont)

- Run the program HelloJMSClient. You should see the messages on the console that the message HelloWorld was sent and received.

- Change the value of the DEFAULT_MESSAGE_COUNT to 57.

- Fix the code in the line 76 to move `context.createProducer()` out of the loop.

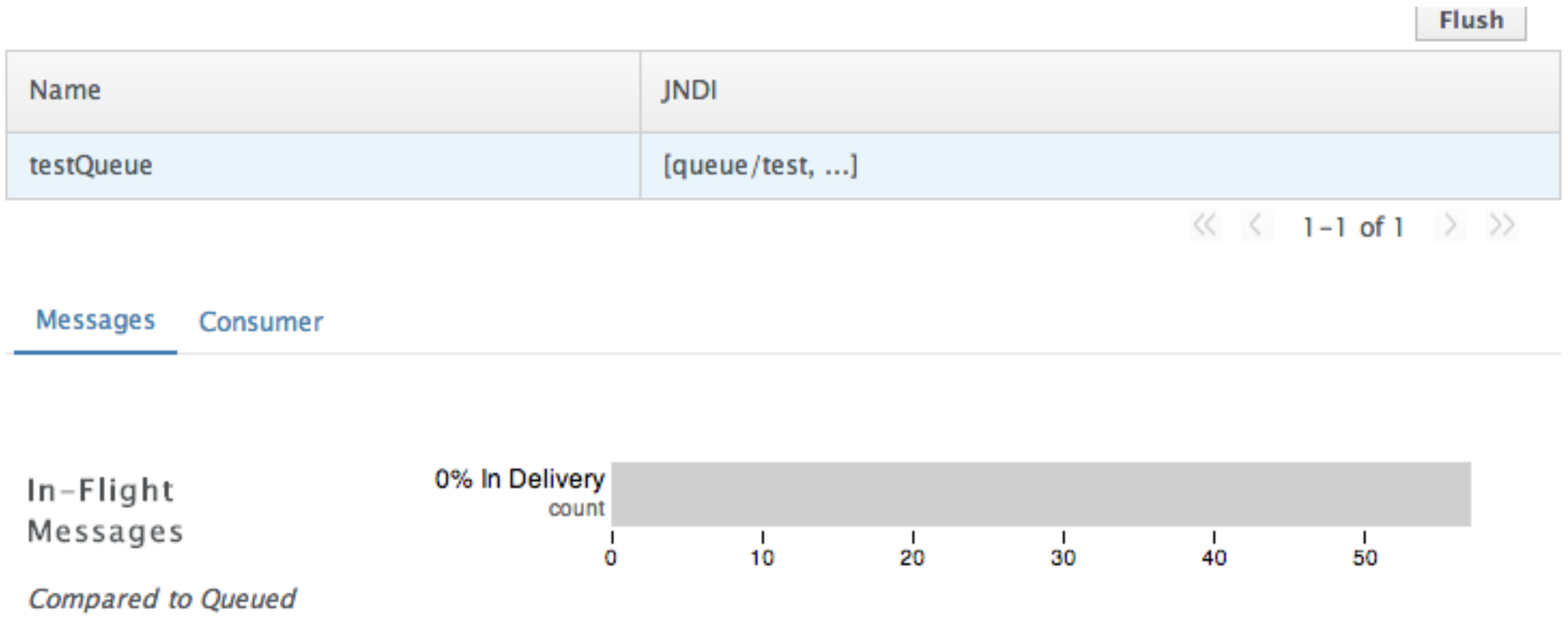
```
JMSProducer producer = context.createProducer();
for (int i = 0; i < count; i++) {
    producer.send(destination, content);
}
```

- Comment out the line 80 that creates the JMSConsumer and the for loop after it.

Walkthrough 3 (cont)

- Run the program HelloJMSClient. It'll send 57 messages, which won't be consumed.

- Open the Wildfly admin console and click on JMS destinations. 57 "in-flight" messages are sitting in a queue.



Name	JNDI
testQueue	[queue/test, ...]

Message Types

- `TextMessage` - a String object
- `MapMessage` - a set of key-value pairs. Key's a String, value's a primitive
- `BytesMessage` - a stream of bytes
- `StreamMessage` - a stream of Java primitives
- `ObjectMessage` - a Serialized Java object
- `Message` - an message with a header, but no body

Setting Message Header

```
context.createProducer()
         .setProperty("MyProperty", "MyValue")
         .setTimeToLive(10000)
         .setDeliveryMode(NON_PERSISTENT)
         .setPriority(2)
         .send(queue, body);
```

Acknowlegement Modes and Trannsactions

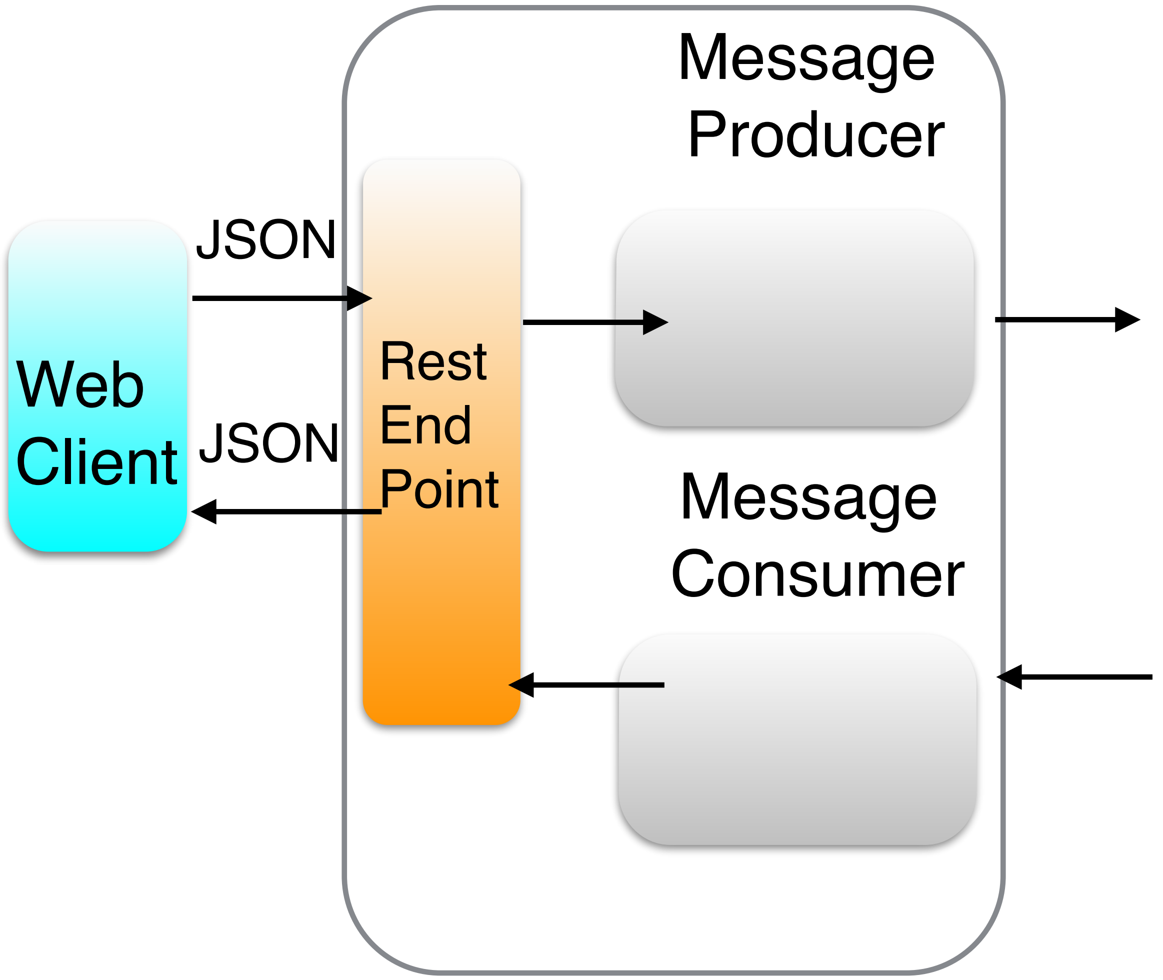
- JMSContext.AUTO_ACKNOWLEDGE
- JMSContext.CLIENT_ACKNOWLEDGE
- JMSContext.DUP_OK_ACknoledge

```
JMSContext context = myConnectionFactory.createContext("myID",
"myPWD",JMSContext.CLIENT_ACKNOWLEDGE);
```

In transacted mode, the message can be committed or rolled back.

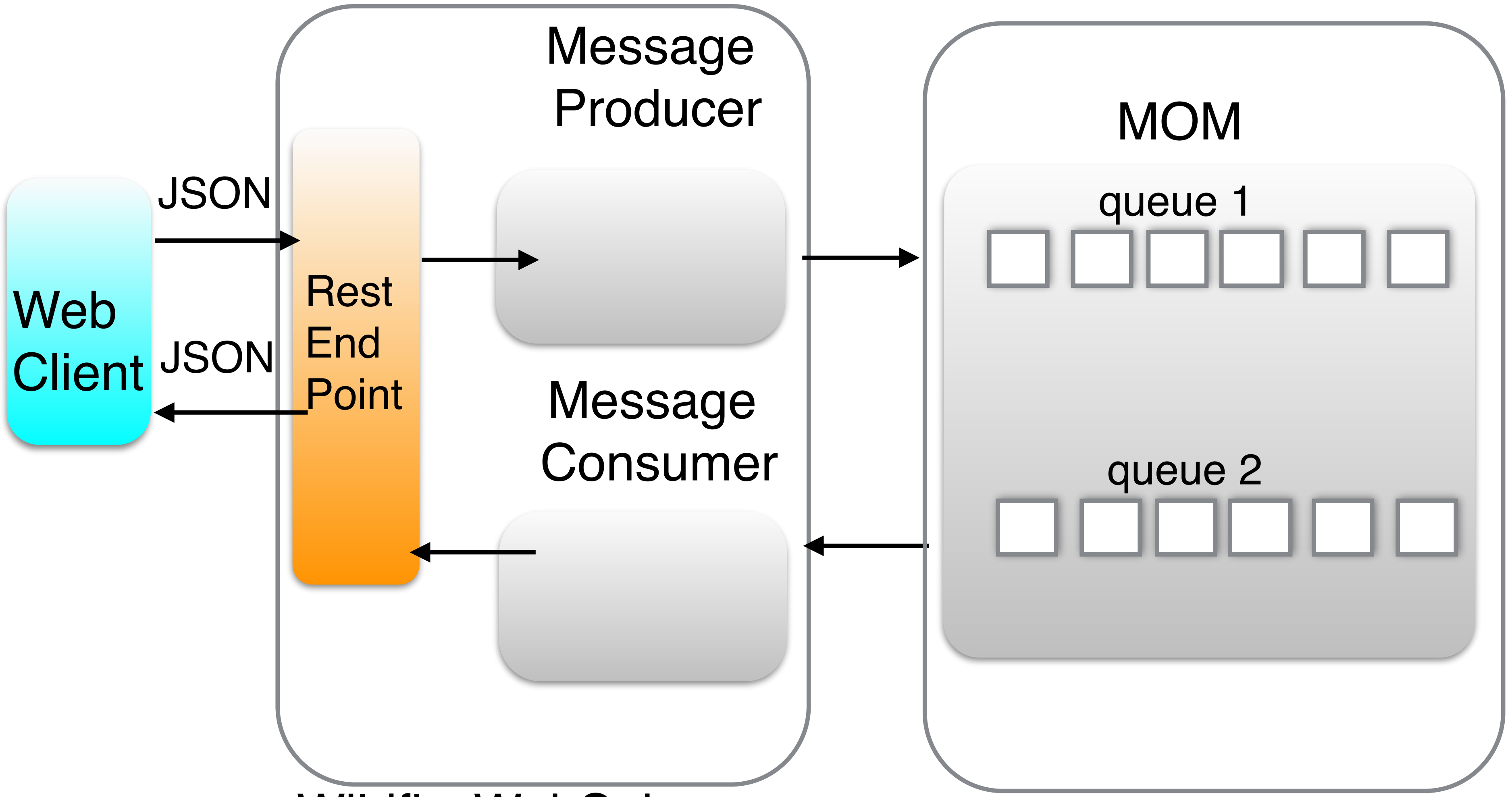
```
JMSContext context = myConnectionFactory.createContext("myID",
"myPWD",JMSContext.TRANSACTED);
```

Messaging in Auction 1



Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

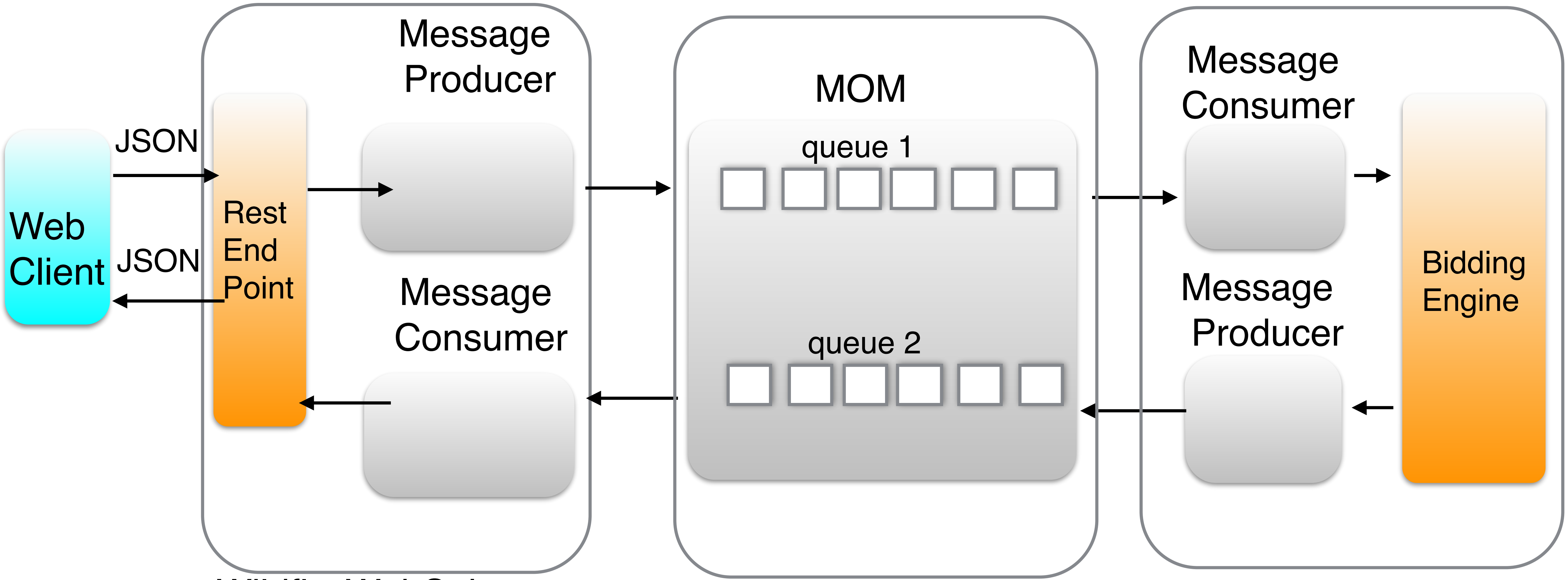
Messaging in Auction 2



Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

WebSphereMQ, ActiveMQ, HornetQ, RabbitMQ, Tibco EMS, MSMQ

Messaging in Auction 3

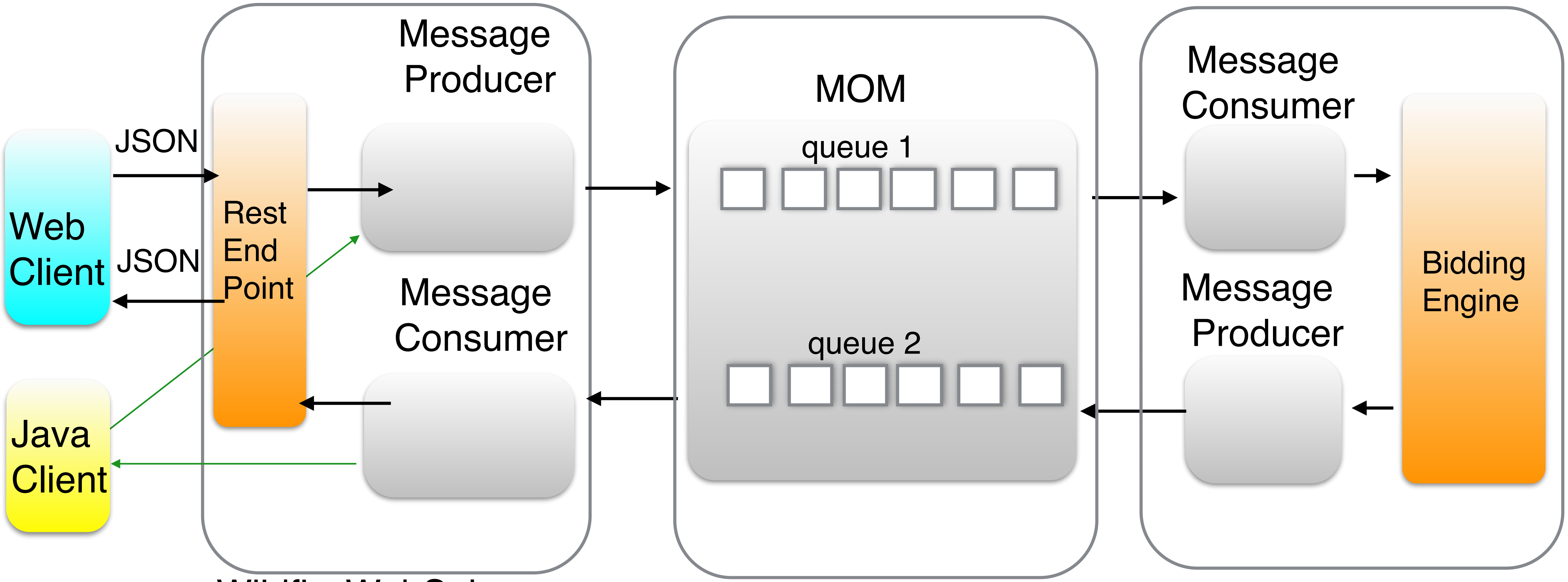


Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

WebSphereMQ, ActiveMQ, HornetQ, RabbitMQ, Tibco EMS, MSMQ

Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

Messaging in Auction 4



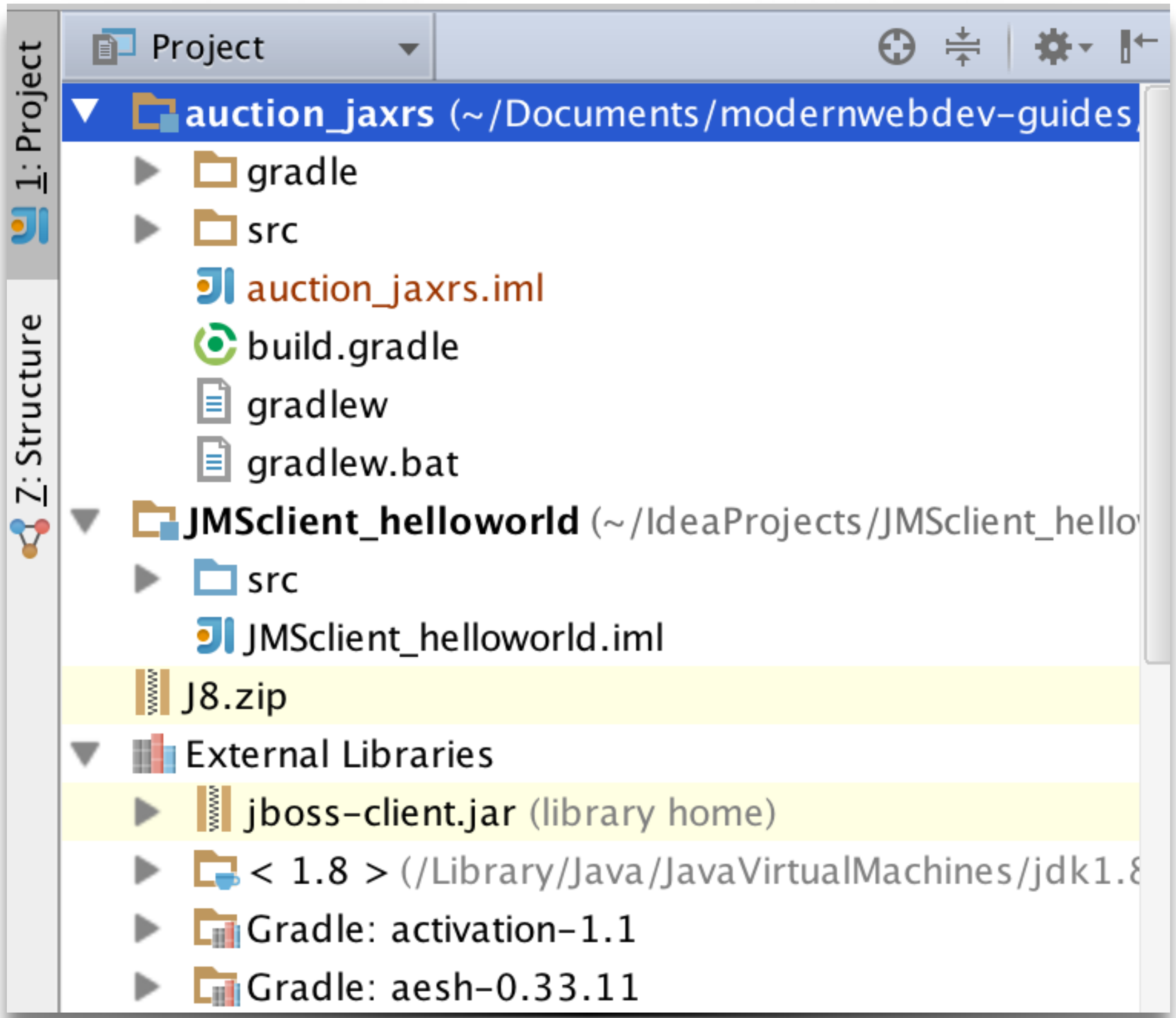
Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

WebSphereMQ, ActiveMQ, HornetQ, RabbitMQ, Tibco EMS, MSMQ

Wildfly, WebSphere, GlassFish, WebLogic, Tomcat, TomEE, Spring Framework

Walkthrough 4(start)

- The goal of this walkthrough is to incorporate JMS client into a REST endpoint
- Import the module auction_jaxrs into your IDEA project: find it on your disk and select build.gradle. Select Use default gradle wrapper.
- It's a slightly modified version of the module from Unit 7.



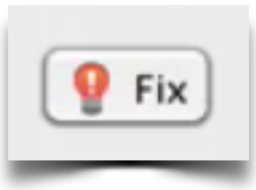
```
▼ auction_jaxrs (~/Documents/modernwebdev-guides
    ▶ gradle
    ▶ src
      auction_jaxrs.iml
      build.gradle
      gradlew
      gradlew.bat
▼ JMSclient_helloworld (~/IdeaProjects/JMSclient_hello
    ▶ src
      JMSclient_helloworld.iml
  J8.zip
▼ External Libraries
    ▶ jboss-client.jar (library home)
    ▶ < 1.8 > (/Library/Java/JavaVirtualMachines/jdk1.
    ▶ Gradle: activation-1.1
    ▶ Gradle: aesh-0.33.11
```

Walkthrough 4(cont)

- Stop your Wildfly server if it's running.

- Go to Run | Edit Configuration, and select your server. Make sure it starts wih -c standalone-full.xml.

- Deploy the artifact auction_jaxrs by pressing the button Fix.



- Start your Wildfly server.

Walkthrough 4(cont)

Open the code of the `BidService` - it includes the modified code of the HelloJMSClient from Walkthrough 3. Note resource injection.

```
private static final Logger log = Logger.getLogger(BidService.class.getName());

// Set up all the default values
private static final String DEFAULT_MESSAGE = "Hello, World!";
private static final String DEFAULT_CONNECTION_FACTORY = "jms/RemoteConnectionFactory";
private static final String DEFAULT_DESTINATION = "jms/queue/test";
private static final String DEFAULT_MESSAGE_COUNT = "57";
private static final String DEFAULT_USERNAME = "quickstartUser";
private static final String DEFAULT_PASSWORD = "quickstartPwd1!";
private static final String INITIAL_CONTEXT_FACTORY =
                                   "org.jboss.naming.remote.client.InitialContextFactory";
private static final String PROVIDER_URL = "http-remoting://127.0.0.1:8080";

@Resource(lookup ="java:/ConnectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup = "queue/test")
Queue testQueue;
```

Walkthrough 4(cont)

Review the code of the `placeBid()` - it invokes `sendBidToQueue()` and returns an empty `Bid` object.

```
@POST
public Bid placeBid(/*@Valid Bid bid*/) {

    sendBidToQueue();

    return new Bid();

}
```

At this point `sendBidToQueue()` sends a Hello Bid message. As a part of your homework, make it send the `Bid` object retrieved from client.

Walkthrough 4(cont)

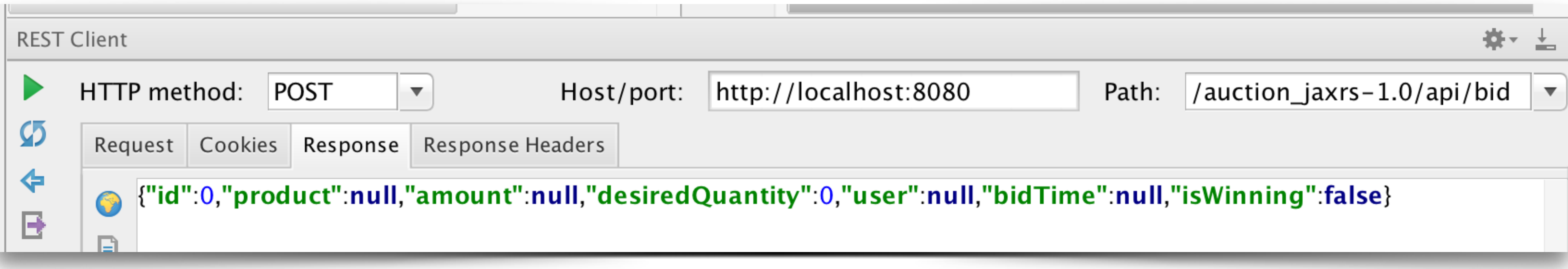
Review the code of `sendMessages()`. It sends Hello Bid message to the test queue.

```
private void sendBidToQueue(){

    try (JMSContext context =
           connectionFactory.createContext(DEFAULT_USERNAME, DEFAULT_PASSWORD)) {
        log.info("\n Sending Hello Bid message from BidService to the queue");

        JMSProducer producer = context.createProducer();

        producer.send(testQueue, "Hello Bid!");
    }
}
```

Walkthrough 4(end)

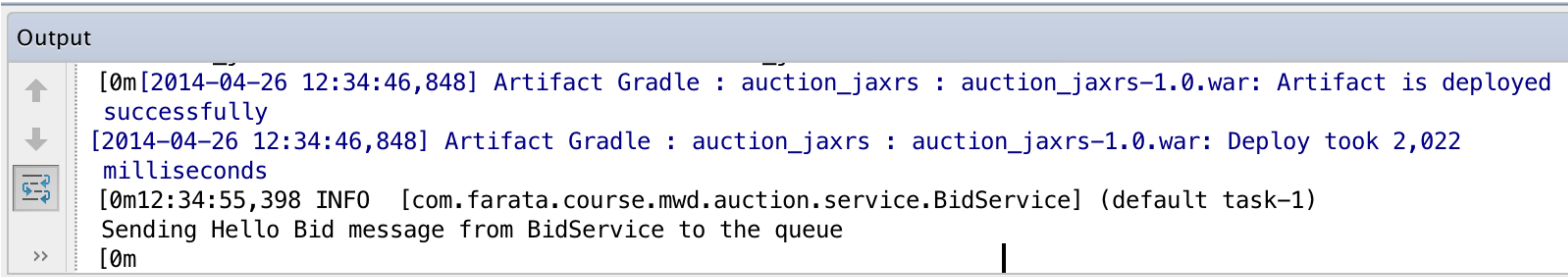
- Open the Rest Client in IDEA, select HTTP method Post.
  Host/Port: http://localhost:8080
  Path: /auction_jaxrs-1.0/api/bid

- Press green button Play. You should see this:

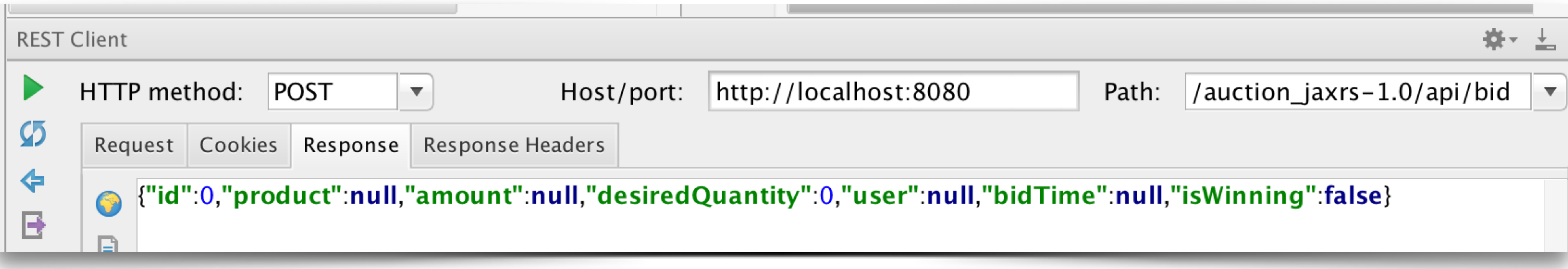


```
{"id":0,"product":null,"amount":null,"desiredQuantity":0,"user":null,"bidTime":null,"isWinning":false}
```

- The Output window shows this:



```
[0m[2014-04-26 12:34:46,848] Artifact Gradle : auction_jaxrs : auction_jaxrs-1.0.war: Artifact is deployed
successfully
[2014-04-26 12:34:46,848] Artifact Gradle : auction_jaxrs : auction_jaxrs-1.0.war: Deploy took 2,022
milliseconds
[0m12:34:55,398 INFO  [com.farata.course.mwd.auction.service.BidService] (default task-1)
Sending Hello Bid message from BidService to the queue
[0m
```



Async Message Listener

In the code snippet below, myMessageListener should point at the object that implements MessageListener interface and has an onMessage() callback.

```
MyMessageListener myMessageListener = new MyMessageListener();

JMSConsumer consumer = context.createConsumer(“myQueue”);
consumer.setMessageListener (myMessageListener);

======

class MyMessageListener implements MessageListener{

   void onMessage(){

         Bid receivedBid = message.getBody(Bid.class);

    }

}
```

Async Message Sender

- Synchronous send is a blocking call that waits for message acknowledgement.
- Async send returns control to the app immediately. This allows other action to be taken, e.g. update display or write into DB.
- When the MOM responds, the callback `onCompletion()` on the `CompletionListener` object is invoked.

Async Send Sample

```
class MyCompletionListener implements CompletionListener {

   CountDownLatch latch;
   Exception exception;

   public MyCompletionListener(CountDownLatch latch) {
      this.latch=latch;
   }

   @Override
   public void onCompletion(Message message) {
      latch.countDown();
   }

   @Override
   public void onException(Message mxessage, Exception exception) {
      latch.countDown();
      this.exception=exception;
   }

   public Exception getException(){
      return exception;
   }
}
```

Message-Driven Beans

- MDB is a managed annotated POJO that serves as a mesage consumer

- MDB implements MessageListener

- Clients never access MDB

- When the message arrives in a queue, the MDB's `onMesage()` is invoked.

- Regular session beans don't support async send/ receive.

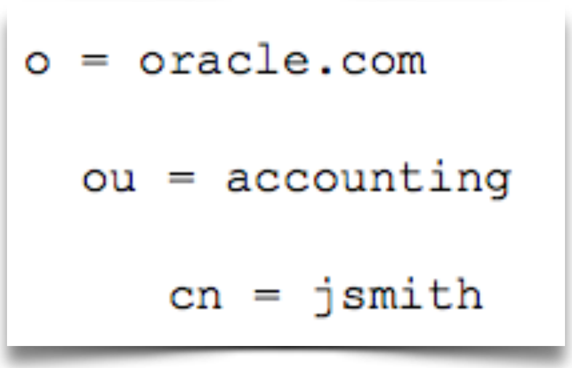
Sample MDB

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "queue/
HELLOWORLDMDBQueue"),
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
public class HelloWorldQueueMDB implements MessageListener {

    private final static Logger LOGGER = Logger.getLogger(HelloWorldQueueMDB.class.toString());

    /**
     * @see MessageListener#onMessage(Message)
     */
    public void onMessage(Message rcvMessage) {
        TextMessage msg = null;
        try {
            if (rcvMessage instanceof TextMessage) {
                msg = rcvMessage.getBody(String.class);
                LOGGER.info("Received Message from queue: " + msg.getText());
            } else {
                LOGGER.warning("Message of wrong type: " + rcvMessage.getClass().getName());
            }
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }
}
```

This sample is taken from Whildfly/quickstart Github repository:
https://github.com/wildfly/quickstart/tree/master/helloworld-mdb

LDAP

- LDAP is Light Weight Directory Accees Protocol
- LDAP servers are optinized for reading. They often used for storing names/roles of employees in lare organizations.
- A Distinguished Name sample:
  cn=*jsmith*, ***ou***=*accounting*, ***o***=*oracle.com*
- The corresponding hierarchy:



```
o = oracle.com

   ou = accounting

        cn = jsmith
```

LDAP and Messaging

Binding JMS administered objects to independent LDAP servers improves recoverability of the system.

```
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_AUTHENTICATION,"simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=Directory Manager");
env.put(Context.SECURITY_CREDENTIALS,"myPassword");

DirContext ctx = new InitialDirContext(env);
Destination myQueue = (Destination) ctx.lookup(
        "cn=BidConfirmationsQueue, ou=OnlineAuction,o=faratasystems.com");
```

Homework (start)

- Modify the script used in Walkthrough 2 to create two more queues: *incomingbids* and *bidconfirmations*.

- Modify the code of the class `BidService` to send the `Bid` object to the *incomingbids* queue. Create an asyncronous message listener that consumes messages from the *bidincoming* queue, extracts received `Bid` object and passes it over to Bid Engine.

- Add the method `confirmBid()` to Bid Engine with the JMS producer that sends bid confirmations to the *bidconfirmations* queue.

Homework (end)

- Add the private `getBidConfirmations()` to the `BidService` endpoint. Add the code there to consume messages from the *bidconfirmations* queue.

- Add the method `confirmBid()` to the `BidService` endpoint. This method should be pushing bid confirmations to the Web client using WebSockets.

- Modify the code of the AngularJS client to handle WebSocket. Use the instructions provided in the Homework for Unit 7.

Additional Materials

- Oracle's Messaging tutorial: http://goo.gl/Q65d0A
- Wildfly Admin Guilde: http://goo.gl/Osqit4
- HornetQ Docs: https://www.jboss.org/hornetq/docs.html