Spring Boot with maven:

pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.springframework</groupId>

<artifactId>gs-spring-boot</artifactId>

<version>0.1.0</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.2.RELEASE</version>

</parent>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

<properties>

<java.version>1.8</java.version>

</properties>

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

The [Spring Boot Maven plugin](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-tools/spring-boot-maven-plugin) provides many convenient features:

* It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
* It searches for the public static void main() method to flag as a runnable class.
* It provides a built-in dependency resolver that sets the version number to match [Spring Boot dependencies](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-dependencies/pom.xml). You can override any version you wish, but it will default to Boot’s chosen set of versions.

@SpringBootApplication is a convenience annotation that adds all of the following:

* @Configuration tags the class as a source of bean definitions for the application context.
* @EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
* Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
* @ComponentScan tells Spring to look for other components, configurations, and services in the hello package, allowing it to find the controllers.

The main() method uses Spring Boot’s SpringApplication.run() method to launch an applicationNo **web.xml** file either. This web application is 100% pure Java and you didn’t have to deal with configuring any plumbing or infrastructure.

Execute:

mvn package && java -jar target/gs-spring-boot-0.1.0.jar

To gracefully exit the application hit ctrl-c.

To create an executable jar we need to add the spring-boot-maven-plugin to our pom.xml. Insert the following lines just below the dependencies section:

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

Use of starter:

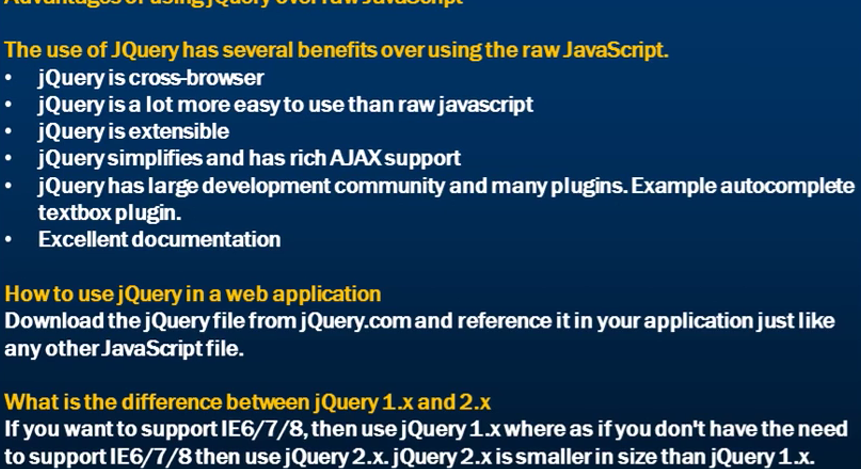
for a simple Spring MVC project, the following artifacts are to be included (Read from *spring-boot-starter*, *spring-boot-starter-web*, *spring-boot-starter-security* respectively):

* Spring Boot artifacts:
  + *org.springframework.boot:spring-boot*
  + *org.springframework.boot:spring-boot-autoconfigure*
  + *org.springframework.boot:spring-boot-starter-logging*
* Spring Core, Web, MVC, Security artifacts:
  + *org.springframework:spring-core*
  + *org.springframework:spring-web*
  + *org.springframework:spring-webmvc*
  + *org.springframework:spring-beans*
  + *org.springframework:spring-context*
  + *org.springframework:spring-expression*
  + *org.springframework:spring-aop*
  + *org.springframework.security:spring-security-config*
  + *org.springframework.security:spring-security-web*
  + *org.hibernate:hibernate-validator*
  + *com.fasterxml.jackson.core:jackson-databind*

Angular Module link: <http://ngmodules.org/modules>

Javascript:

JQuery Tutorial:

s

Jms:

Java Message Service (JMS) API, a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communications. Messaging is a method of communication between software components or applications. A messaging system is a peer-to-peer facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages. Messaging enables distributed communication that is **loosely coupled**. A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate. In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which message format and which destination to use. In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application’s methods. The JMS API enables communication that is not only loosely coupled but also:

* **Asynchronous**: A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.
* **Reliable**: The JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The Java EE platform, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages. The JMS provider can be integrated with the application server using the Java EE Connector architecture. You access the JMS provider through a resource adapter. This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers.

* A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.
* **JMS clients** are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.
* **Messages** are the objects that communicate information between JMS clients.
* **Administered objects** are preconfigured JMS objects created by an administrator for the use of clients. The two kinds of JMS administered objects are destinations and connection factories

Resource injection enables you to inject any resource available in the JNDI namespace into any container-managed object, such as a servlet, an enterprise bean, or a managed bean. For eg, we can use resource injection to inject data sources, connectors, or any other desired resources available in the JNDI namespace.

@Resource has the following elements:

·  name: The JNDI name of the resource

·  type: The Java  type of the resource

·  authenticationType: The authentication type to use for the resource

·  shareable: Indicates whether the resource can be shared

·  mappedName: A non-portable, implementation-specific name to which the resource should be mapped

·  description: The description of the resource

package com.example;

public class SomeClass {

@Resource(name="customerDB")

private javax.sql.DataSource myDB;

...

}

In the code above, the JNDI name is customerDB, and the inferred type is javax.sql.DataSource.class.

package com.example;

public class SomeClass {

@Resource

private javax.sql.DataSource myDB;

...

}

In the code above, the container infers the name of the resource based on the class name and the field name: com.example.SomeClass/myDB. The inferred type isjavax.sql.DataSource.class.

package com.example;

public class SomeClass {

private javax.sql.DataSource myDB;

...

@Resource

private void setMyDB(javax.sql.DataSource ds) {

myDB = ds;

}

https://pixel-geo.prfct.co/cs/?partnerId=opxhttps://pixel-geo.prfct.co/cs/?partnerId=pubhttps://pixel-geo.prfct.co/seg/?add=649205,5411417,5411489,5411540,5411554,5411559,5411562&source=js_tag&a_id=7618https://www.facebook.com/tr?id=1835809346650227&ev=PageView&noscript=1https://secure.adnxs.com/seg?t=2&add=5411562https://secure.adnxs.com/seg?t=2&add=5411559

In the code above, the container refers the name of the resource according to the class name and the field name: com.example.SomeClass/myDB. The type which is javax.sql.DataSource.class.

package com.example;

public class SomeClass {

private javax.sql.DataSource myDB;

...

@Resource (name="customerDB")

private void setMyDB (javax.sql.DataSource ds) {

myDB = ds;

}

...

}

In the code above, the JNDI name is customerDB, and the inferred type is javax.sql.DataSource.class.

Class Injection:

To use class-based injection, decorate the class with a @Resource annotation, and set the required name and type elements.

@Resource(name="myMessageQueue",

type="javax.jms.ConnectionFactory")

public class SomeMessageBean {

...

}

Declaring Multiple Resources

The **@Resources** annotation is used to group together multiple **@Resource** declarations for **class injection only.**

@Resources({

@Resource(name="myMessageQueue",

type="javax.jms.ConnectionFactory"),

@Resource(name="myMailSession",

type="javax.mail.Session")

})

public class SomeMessageBean {

...

}

### Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider. Messaging Domains

Before the JMS API existed, most messaging products supported either the point-to-point or the publish/subscribe approach to messaging. The JMS specification provides a separate domain for each approach and defines compliance for each domain. A stand-alone JMS provider can implement one or both domains. A Java EE provider must implement both domains.

#### Point-to-Point Messaging Domain

A **point-to-point** (PTP) product or application is built on the concept of message **queues**, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in [Figure 47-3](http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html#bncec), has the following characteristics:

* Each message has only one consumer.
* A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
* The receiver acknowledges the successful processing of a message.

#### Publish/Subscribe Messaging Domain

Pub/sub messaging has the following characteristics.

* Each message can have multiple consumers.
* Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.
* The JMS API relaxes this timing dependency to some extent by allowing subscribers to create **durable subscriptions**, which receive messages sent while the subscribers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

Message Consumption

Messages can be consumed in either of two ways:

* **Synchronously**: A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
* **Asynchronously**: A client can register a **message listener** with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener’s onMessage method, which acts on the contents of the message.

## The JMS API Programming Model

The basic building blocks of a JMS application are:

* Administered objects: connection factories and destinations
* Connections
* Sessions
* Message producers
* Message consumers
* Messages

### JMS Administered Objects

Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

With GlassFish Server, you can use the asadmin create-jms-resource command or the Administration Console to create JMS administered objects in the form of connector resources. You can also specify the resources in a file named glassfish-resources.xml that you can bundle with an application.

At the beginning of a JMS client program, you usually inject a connection factory resource into a ConnectionFactory object. For example, the following code fragment specifies a resource whose JNDI name is jms/ConnectionFactory and assigns it to a ConnectionFactory object:

@Resource(lookup = "jms/ConnectionFactory")

private static ConnectionFactory connectionFactory;

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resourceThe following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace.

@Resource(lookup = "jms/Queue")

private static Queue queue;

@Resource(lookup = "jms/Topic")

private static Topic topic;

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the ConnectionFactory interface, you can inject a QueueConnectionFactory resource and use it with a Topic, and you can inject a TopicConnectionFactory resource and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

**JMS Connections**

Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

Connection connection = connectionFactory.createConnection();

Before an application completes, you must close any connections you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

connection.close();

Before your application can consume messages, you must call the connection’s start method; for details, see [JMS Message Consumers](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncep). If you want to stop message delivery temporarily without closing the connection, you call the stop method.

### **JMS Sessions**

A **session** is a single-threaded context for producing and consuming messages. You use sessions to create the following:

* Message producers
* Message consumers
* Messages
* Queue browsers
* Temporary queues and topics

Sessions implement the Session interface. After you create a Connection object, you use it to create a Session:

Session session = connection.createSession(false,

Session.AUTO\_ACKNOWLEDGE);

The first argument means the session is not transacted; the second means the session automatically acknowledges messages when they have been received successfully

To create a **transacted session**, use the following code:

Session session = connection.createSession(true, 0);

Here, the first argument means the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions.

A **message producer** is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

MessageProducer producer = session.createProducer(dest);

MessageProducer producer = session.createProducer(queue);

MessageProducer producer = session.createProducer(topic);

You can create an unidentified producer by specifying null as the argument to createProducer. With an unidentified producer, you do not specify a destination until you send a message.

After you have created a message producer, you can use it to send messages by using the send method:

producer.send(message);

If you have created an unidentified producer, use an overloaded send method that specifies the destination as the first parameter. For example:

MessageProducer anon\_prod = session.createProducer(null);

anon\_prod.send(dest, message);

A **message consumer** is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

MessageConsumer consumer = session.createConsumer(dest);

MessageConsumer consumer = session.createConsumer(queue);

MessageConsumer consumer = session.createConsumer(topic);

You use the Session.**createDurableSubscriber** method to create a durable topic subscriber. This method is valid only if you are using a topic.

After you have created a message consumer it becomes active, and you can use it to receive messages. You can use the close method for a MessageConsumer to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling its start method. You use the receive method to consume a message **synchronously**. You can use this method at any time after you call the start method:

connection.start();

Message m = consumer.receive();

connection.start();

Message m = consumer.receive(1000); // time out after a second

#### JMS Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you define a class named Listener that implements the MessageListener interface, you can register the message listener as follows:

Listener myListener = new Listener();

consumer.setMessageListener(myListener);

In the Java EE platform, a MessageListener can be used only in an application client, not in a web component or enterprise bean. After you register the message listener, you call the start method on the Connection to begin message delivery. (If you call start before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener’s onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message, which your implementation of the method can cast to any of the other message types (see [Message Bodies](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncew)).

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created. A message listener does, however, usually expect a specific message type and format.

Your onMessage method should handle all exceptions. It must not throw checked exceptions, and throwing a RuntimeException is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session. At any time, only one of the session’s message listeners is running.

#### JMS Message Selectors

If your messaging application needs to **filter the messages** it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages that interest it

Message selectors assign the work of filtering messages to the JMS provider rather than to the application. A message selector is a String that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has a NewsType property that is set to the value 'Sports' or 'Opinion':

NewsType = ’Sports’ OR NewsType = ’Opinion’

The createConsumer and createDurableSubscriber methods allow you to specify a message selector as an argument when you create a message consumer. The message consumer then receives only messages whose headers and properties match the selector. (See [Message Headers](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncet), and [Message Properties](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncev).) A message selector cannot select messages on the basis of the content of the message body.

**JMS Messages**

A JMS message can have three parts: a header, properties, and a body. Only the header is required. every message has a unique identifier, which is represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the Message interface. Some header fields are intended to be set by a client, but many are set automatically by the send or the publish method, which overrides any client-set values.

**Table 47-1 How JMS Message Header Field Values Are Set**

|  |  |
| --- | --- |
| **Header Field** | **Set By** |
| JMSDestination | send or publish method |
| JMSDeliveryMode | send or publish method |
| JMSExpiration | send or publish method |
| JMSPriority | send or publish method |
| JMSMessageID | send or publish method |
| JMSTimestamp | send or publish method |
| JMSCorrelationID | Client |
| JMSReplyTo | Client |
| JMSType | Client |
| JMSRedelivered | JMS provider |

The JMS API defines five message body formats, also called message types, which allow you to send and receive data in many different forms and which provide compatibility with existing messaging formats. [Table 47-2](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncex) describes these message types.

Table 47-2 JMS Message Types

|  |  |
| --- | --- |
| **Message Type** | **Body Contains** |
| TextMessage | A java.lang.String object (for example, the contents of an XML file). |
| MapMessage | A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined. |
| BytesMessage | A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format. |
| StreamMessage | A stream of primitive values in the Java programming language, filled and read sequentially. |
| ObjectMessage | A Serializable object in the Java programming language. |
| Message | Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required. |

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a TextMessage, you might use the following statements:

TextMessage message = session.createTextMessage();

message.setText(msg\_text); // msg\_text is a String

producer.send(message);

At the consuming end, a message arrives as a generic Message object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the getText method:

Message m = consumer.receive();

if (m instanceof TextMessage) {

TextMessage message = (TextMessage) m;

System.out.println("Reading message: " + message.getText());

} else {

// Handle error

}

### **JMS Queue Browsers**

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a QueueBrowser object that allows you to browse the messages in the queue and display the header values for each message. To create a QueueBrowser object, use the Session.createBrowser method. For example:

QueueBrowser browser = session.createBrowser(queue);

The createBrowser method allows you to specify a message selector as a second argument when you create a QueueBrowserQueueBrowser. For information on message selectors, see [JMS Message Selectors](http://docs.oracle.com/javaee/6/tutorial/doc/bnceh.html#bncer).

The JMS API provides no mechanism for browsing a topic. Messages usually disappear from a topic as soon as they appear: If there are no message consumers to consume them, the JMS provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, no facility exists for examining them.

### **JMS Exception Handling**

The root class for exceptions thrown by JMS API methods is JMSException. Catching JMSException provides a generic way of handling all exceptions related to the JMS API.

The JMSException class includes the following subclasses, described in the API documentation:

* IllegalStateException
* InvalidClientIDException
* InvalidDestinationException
* InvalidSelectorException
* JMSSecurityException
* MessageEOFException
* MessageFormatException
* MessageNotReadableException
* MessageNotWriteableException
* ResourceAllocationException
* TransactionInProgressException
* TransactionRolledBackException

**Using Basic Reliability Mechanisms**

The most reliable way to produce a message is to send a PERSISTENT message within a transaction. JMS messages are PERSISTENT by default

* **Controlling message acknowledgment**: You can specify various levels of control over message acknowledgment.
* **Specifying message persistence**: You can specify that messages are persistent, meaning they must not be lost in the event of a provider failure.
* **Setting message priority levels**: You can set various priority levels for messages, which can affect the order in which the messages are delivered.
* **Allowing messages to expire**: You can specify an expiration time for messages so they will not be delivered if they are obsolete.
* **Creating temporary destinations**: You can create temporary destinations that last only for the duration of the connection in which they are created.

#### Controlling Message Acknowledgment

Until a JMS message has been acknowledged, it is not considered to be successfully consumed. The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged. Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see [Using JMS API Local Transactions](http://docs.oracle.com/javaee/6/tutorial/doc/bncfu.html#bncgh)), acknowledgment happens automatically when a transaction is committed. If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depend on the value specified as the second argument of the createSession method. The three possible argument values are as follows:

* Session.AUTO\_ACKNOWLEDGE: The session automatically acknowledges a client’s receipt of a message either when the client has successfully returned from a call to receive or when the MessageListener it has called to process the message returns successfully.

A synchronous receive in an AUTO\_ACKNOWLEDGE session is the one exception to the rule that message consumption is a three-stage process as described earlier. In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

* Session.CLIENT\_ACKNOWLEDGE: A client acknowledges a message by calling the message’s acknowledge method. In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of **all** messages that have been consumed by its session. For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

#### Session.DUPS\_OK\_ACKNOWLEDGE: This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages. (If the JMS provider redelivers a message, it must set the value of the JMSRedelivered message header to true.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates. If messages have been received from a queue but not acknowledged when a session terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue. The provider also retains unacknowledged messages for a terminated session that has a durable TopicSubscriber. (See [Creating Durable Subscriptions](http://docs.oracle.com/javaee/6/tutorial/doc/bncfu.html#bncgd).) Unacknowledged messages for a nondurable TopicSubscriber are dropped when the session is closed. If you use a queue or a durable subscription, you can use the Session.recover method to stop a nontransacted session and restart it with its first unacknowledged messageIn effect, the session’s series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from those that were originally delivered, if messages have expired or if higher-priority messages have arrived. For a nondurable TopicSubscriber, the provider may drop unacknowledged messages when its session is recovered. Specifying Message Persistence

The JMS API supports two delivery modes specifying whether messages are lost if the JMS provider fails. These **delivery modes** are fields of the DeliveryMode interface.

* The PERSISTENT delivery mode, the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure. A message sent with this delivery mode is logged to stable storage when it is sent.
* The NON\_PERSISTENT delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

* You can use the setDeliveryMode method of the MessageProducer interface to set the delivery mode for all messages sent by that producer. For example, the following call sets the delivery mode to NON\_PERSISTENT for a producer:

producer.setDeliveryMode(DeliveryMode.NON\_PERSISTENT);

* You can use the long form of the send or the publish method to set the delivery mode for a specific message. The second argument sets the delivery mode. For example, the following send call sets the delivery mode for message to NON\_PERSISTENT:

producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);

The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode**, the default is PERSISTENT**. Using the NON\_PERSISTENT delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

#### Setting Message Priority Levels

You can use message priority levels to instruct the JMS provider to deliver urgent messages first. You can set the priority level in either of two ways.

* You can use the setPriority method of the MessageProducer interface to set the priority level for all messages sent by that producer. For example, the following call sets a priority level of 7 for a producer:

producer.setPriority(7);

* You can use the long form of the send or the publish method to set the priority level for a specific message. The third argument sets the priority level. For example, the following send call sets the priority level for message to 3:

producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);

T**he ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4.**

#### Allowing Messages to Expire

By default, a message never expires. If a message will become obsolete after a certain period, however, you may want to set an expiration time. You can do this in either of two ways.

* You can use the setTimeToLive method of the MessageProducer interface to set a default expiration time for all messages sent by that producer. For example, the following call sets a time to live of one minute for a producer:

producer.setTimeToLive(60000);

* You can use the long form of the send or the publish method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. For example, the following send call sets a time to live of 10 seconds:

producer.send(message, DeliveryMode.NON\_PERSISTENT, 3, 10000);

**If the specified timeToLive value is 0, the message never expires.**

#### Creating Temporary Destinations

Normally, you create JMS destinations (queues and topics) administratively rather than programmatically. Your JMS provider includes a tool to create and remove destinations, and it is common for destinations to be long-lasting.

The JMS API also enables you to create destinations (TemporaryQueue and TemporaryTopic objects) that last only for the duration of the connection in which they are created. You create these destinations dynamically using the Session.createTemporaryQueue and the Session.createTemporaryTopic methods.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination. Any message producer can send to the temporary destination. If you close the connection to which a temporary destination belongs, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism. If you create a temporary destination and specify it as the value of the JMSReplyTo message header field when you send a message, then the consumer of the message can use the value of the JMSReplyTo field as the destination to which it sends a reply. The consumer can also reference the original request by setting the JMSCorrelationID header field of the reply message to the value of the JMSMessageID header field of the request. For example, an onMessage method can create a session so that it can send a reply to the message it receives. It can use code such as the following:

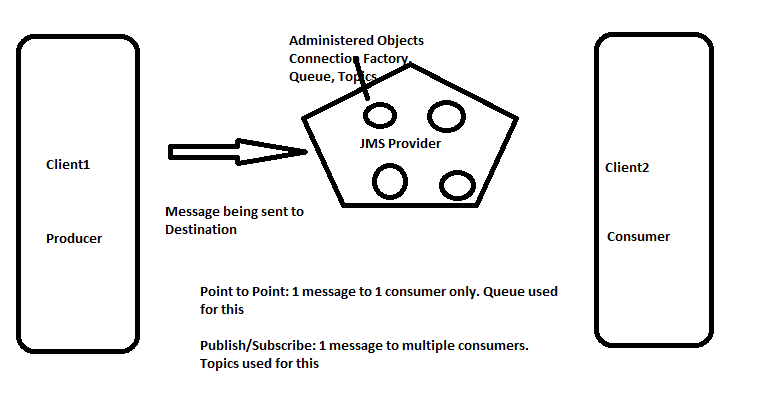
producer = session.createProducer(msg.getJMSReplyTo());

replyMsg = session.createTextMessage("Consumer " +

"processed message: " + msg.getText());

replyMsg.setJMSCorrelationID(msg.getJMSMessageID());

producer.send(replyMsg);



**Networking:**

TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications. When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, you end up with a jumbled HTML file, a corrupt zip file, or some other invalid information. The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another. Sending datagrams is much like sending a letter through the postal service: The order of delivery is not important and is not guaranteed, and each message is independent of any other.

Notification:

WebKit browsers (Google Chrome and Safari).

The new Notifications API gives you the ability to launch desktop notifications directly from your web applications. This can be very useful for updating user’s with key information, particularly in [AJAX applications](http://blog.teamtreehouse.com/a-quick-guide-to-understanding-ajax) where you may have HTTP requests taking place in the background.

In this blog post you are going to learn how to build notifications into your web applications.

## Detecting Browser Support

Support for the Notifications API is currently limited to WebKit browsers (Google Chrome and Safari).

To detect support for notifications you can check for the presence of the webkitNotifications interface on the window interface. The example below shows how you might do this.

if (window.webkitNotifications) {

console.log('Your web browser does support notifications!');

} else {

console.log('Your web browser does not support notifications!');

}

## Requesting Permission to Create Notifications

Before you can create notifications you need to request permission to do so from the user. It is worth noting that these permissions are tied to a particular domain name (or sub-domain), so a user will have to allow notifications for each website that wants to use them.

To check the current notification permissions you can use the checkPermission() function.

This function will return one of the following values:

* 0 – Allowed
* 1 – Not Allowed
* 2 – Denied

If the checkPermission() function returns 1 it means that the user has not yet authorized (or denied) access to notifications on the domain. If the function returns 2, this means that the user has explicitly blocked the domain from using notifications (they hit the ‘Deny’ button).

You can request permission to create notifications using the requestPermission() function. This will display an info bar at the top of the browser viewport, as shown in the figure below.

[Notification Permissions Bar in Google Chrome](http://blog.teamtreehouse.com/wp-content/uploads/2012/12/chrome-notifications-permissions.png)

The Notification Permissions Bar in Google Chrome

The requestPermissions() function will only work if it is called as part of a user action like a mouse click or key press.

Here is an example of how you might check if your web application has permission to create notifications; and request permission if it does not.

if (window.webkitNotifications.checkPermission() == 0) {

// Good to go, you can create a notification.

} else {

window.webkitNotifications.requestPermission(function(){});

}

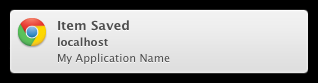
**Note:** Safari requires a callback to be passed to the requestPermission() function. This can just be an empty function as I have used above.

## Creating Notifications

Now it’s time to learn how you actually create notifications. You do this using the createNotification() function.

This function takes three parameters.

* iconUrl – The URL of an image that will be displayed in the notification. If you do not specify a valid URL the browser will display a default icon (usually the browser logo).
* title – The notification title.
* body – Secondary text that will be displayed below the title.

[](http://blog.teamtreehouse.com/wp-content/uploads/2012/12/chrome-notification.png)

A Notification in Google Chrome

Once you have created a notification you use the show() function to display the notification to the user. The example below shows how to create and show a notification like the one in the figure above.

var myNotification = window.webkitNotifications.createNotification('icon.png', 'Item Saved', 'My Application Name');

myNotification.show();

You can hide a notification using the cancel() function.

**Note:**It is possible that a notification may be in a queue waiting to be displayed. Due to screen size limitations the browser will only display a certain number of notifications at the same time. The cancel() function will also remove a pending notification from the queue.

## Listening for Notification Events

Notifications emit a number of events that you can use to execute code at key points in the notification lifecycle.

* ondisplay – Called when the notification is displayed. Note that this may not be immediately after the show() function is called.
* onerror – Called when the notification cannot be displayed due to an error.
* onclose – Called when the notification is closed by the user.
* onclick – Called when a user clicks on the notification dialog.

The most useful of these events is the onclick event. You could use this for a number of different applications. A common use case is to load a new page when the user clicks on the notification. The example below shows how you could achieve this behaviour.

var myNotification = window.webkitNotifications.createNotification('mike.png', 'New Content Available', 'Click to view');

myNotification.onclick = function() {

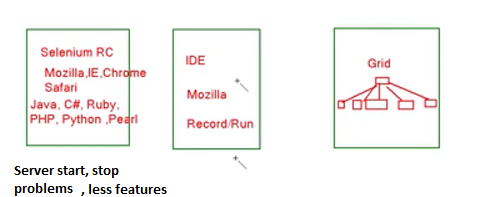
window.location = 'http://teamtreehouse.com/new/content';

}

myNotification.show();

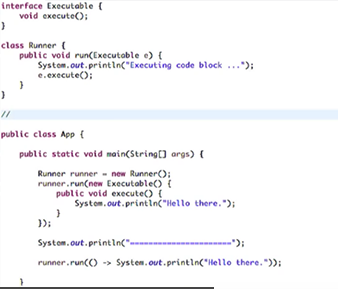
Link for downloading checksum utility: <https://raylin.wordpress.com/downloads/md5-sha-1-checksum-utility>

**Selenium:**

****

RC as selenium1, webdriver as selenium2.

Lambda expression:

s