9. Sending messages asynchronously

-Synchronous communication:

-**Asynchronous messaging**: indirectly sending messages from one app to another without waiting a response.

-We use asynchronous messaging to send orders from Taco Cloud web to a separate app in Taco Cloud kitchens where the tacos will be prepared.

-We consider 3 options for asynchronous messaging: Java Message Service (**JMS**), **RabbitMQ** and Advanced Message Queueing Protocol (**AMQP**), and **Apache Kafka**. To basic sending and receiving of messages, look Spring’s support for message-driven **POJOs**: a way to receive messages that resembles Enterprise JavaBeans’ message-driven beans (MDBs)

# 9.1 Sending messages with JMS

-**JMS** is a Java standard that defines an API for working with **message brokers**. All compliant implementations can be worked with via a common interface in the same way that JDBC has given relational database operations a common interface.

-Spring supports JMS through a template-based abstraction **JmsTemplate**. It’s easy to **send messages** across **queues** and **topics** from **producer side** and to **receive messages** on **consumer side**.

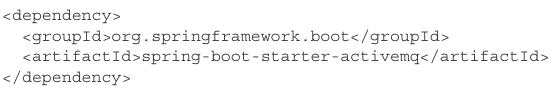
-Spring also supports the notion of message-driven **POJOs**: **Java objects** that **react** to **messages** arriving on a **queue** or **topic** in asynchronous fashion.

-Know more about JMS: ActiveMQ in Action by Bruce Snyder, Dejan Bosanac, and Rob Davies (Manning, 2011)

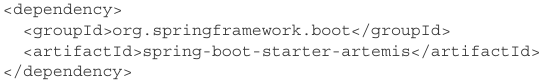
## 9.1.1 Setting up JMS

-Decide use Apache ActiveMQ or the newer Apache ActiveMQ Artemis broker.

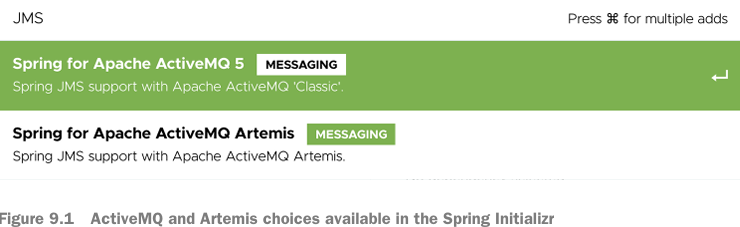
+Using ActiveMQ:



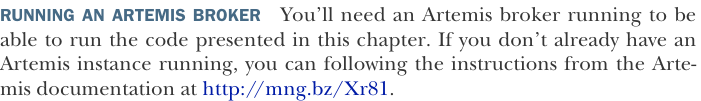
+Using ActiveMQ Artemis:



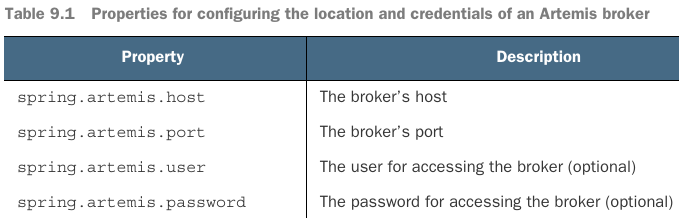
+Use Spring Initializer to configure:

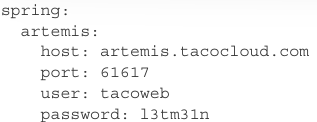


-**Artemis** is a next-generation reimplementation of ActiveMQ, making ActiveMQ a legacy option. -> Use Artemis. The only significant differences will be in how you configure Spring to create connections to broker.



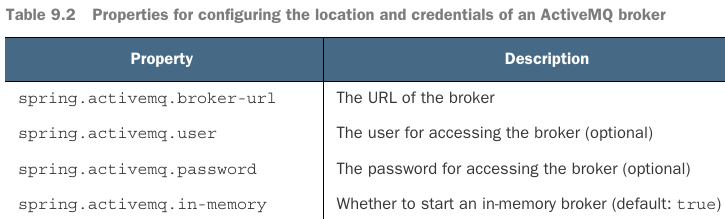
+By default, Artemis broker is listening on localhost at port 61616. Once you send your app into production, you can set few properties that tell Spring how to access the broker:



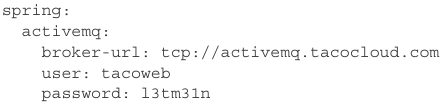


+This sets up Spring to create broker connection to Artemis broker listening at artemis.tacocloud.com port 61617.

-If you use ActiveMQ:



+URL should be: tcp: //URL



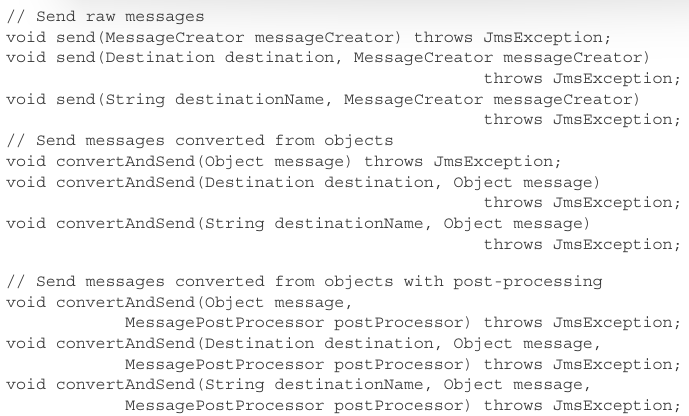
+You shouldn’t need to configure these properties for development when the broker is running locally.

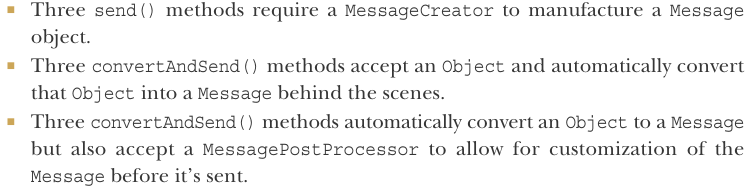
-Instead of using an **embedded broker**, you can install and **start** an **Artemis** (or ActiveMQ) broker before moving on: 

## 9.1.2 Sending messages with JmsTemplate

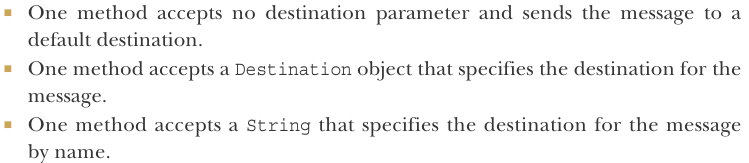
-JmsTemplate is the centerpiece of Spring’s JMS.

-Methods for sending messages:

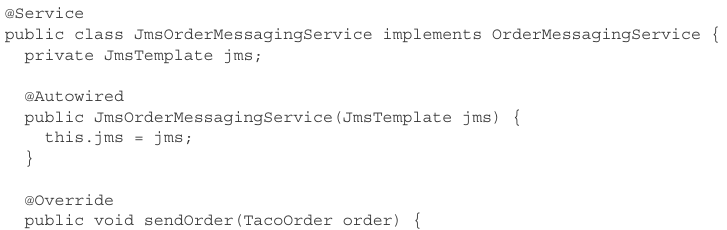


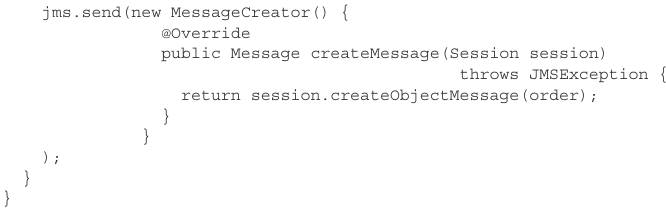


+Each of 3 method categories is composed of 3 overriding methods that are distinguished by how the JMS destination is specified:

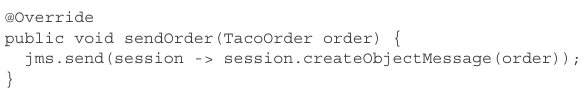


-Use send() in JmsOrderMessagingService:





+You can otherwise use functional interface:

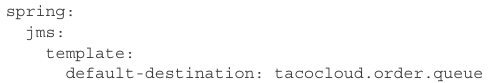


-Put this service to work by injecting it into OrderApiController and call sendOrder() when order is created:



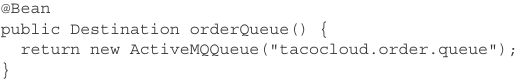
-When create an order, a message should be sent to broker for routing to another app that will receive the order. You can use Artemis console to view the content of the queue: <http://mng.bz/aZx9>

-Specify a **default destination** for jms.send()

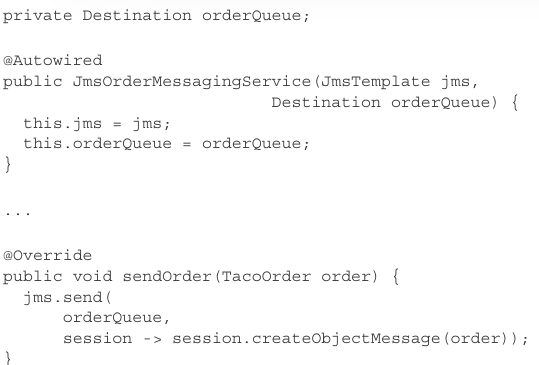


-If you need to send to other destination, specify that as a parameter to send():

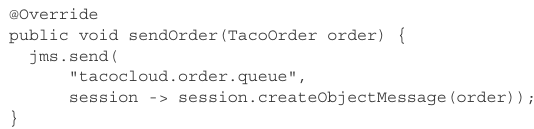
+Declare a **Destination** bean then inject it into the bean that performs messaging in configuration class.



+Inject it into service and specify the destination:

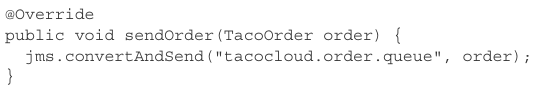


+Use it in method:



-**Converting Messages Before Sending:**

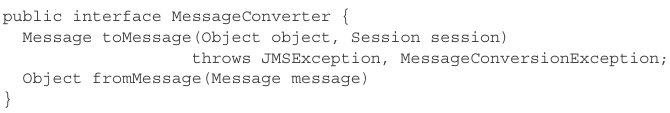
+JmsTemplatesconvertAndSend() simplifies message publication by eliminating the need to provide a MessageCreator. Instead, you pass the object to convertAndSend(), object will be converted into Message before being sent:



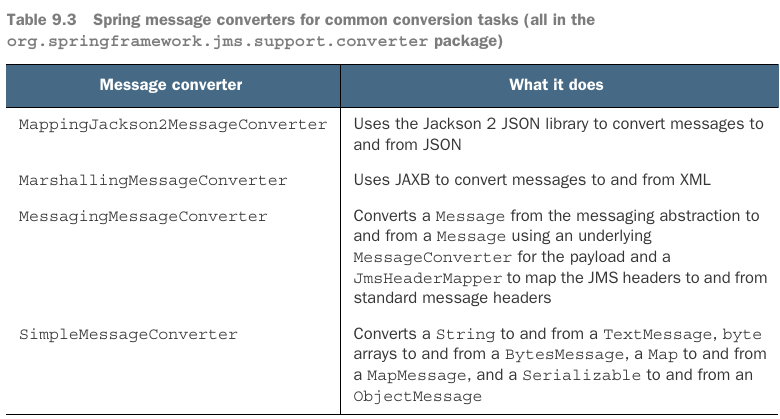
+It will accept either a Destination or String value to specify the destination.

-Configuring a Message Converter

+MessageConverter is a Spring-defined interface that has only 2 methods:

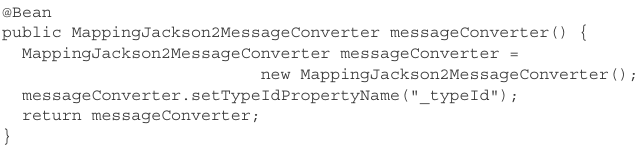


+This interface is simple, you often won’t need to create a custom implementation. Spring offers implementations



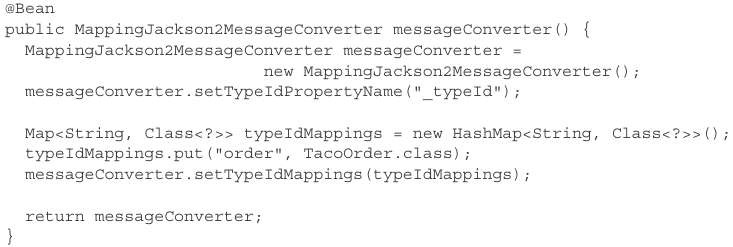
+SimpleMessageConverter is the default, requires the object implements **Serializable**.

+Apply a different message converter: declare an instance of the converter as a bean



setTypeIdPropertyName(): enable the receiver to know what type to convert an incoming message to. It requires the receiver also have the same type with the same fully qualified classname.

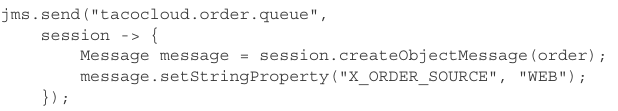
+Allow for more flexibility, map a synthetic type name to actual type by setTypeIdMappings(). Example: map a synthetic TacoOrder type ID to TacoOrder class:

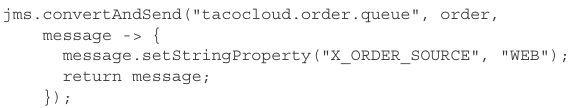


+Instead of fully qualified classname being sent in message’s \_typeId, the value “TacoOrder” will be sent.

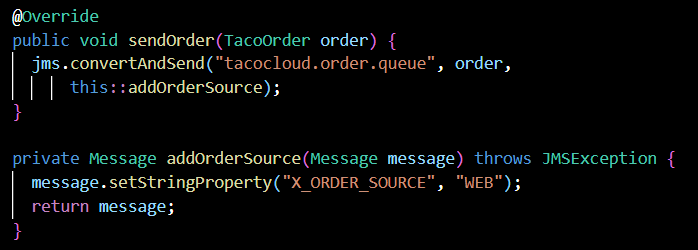
-**Postprocessing Messages**:

+Add a new **source** property to TacoOrder object to carry this info, populating it with WEB for orders placed online and STORE for ordre placed in the store.





+Avoid duplication:



## 9.1.3 Receiving JMS messages

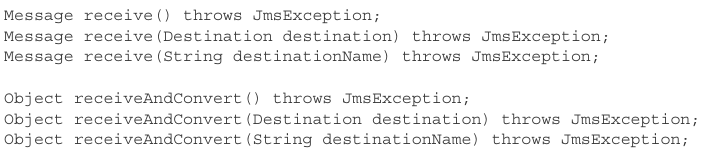
-To consume messages, we have the choice of a pull model (your code requests a message and waits until one arrives) + push model (messages are handed to your code as they become available)

-JmsTemplate use a pull model ->start with this

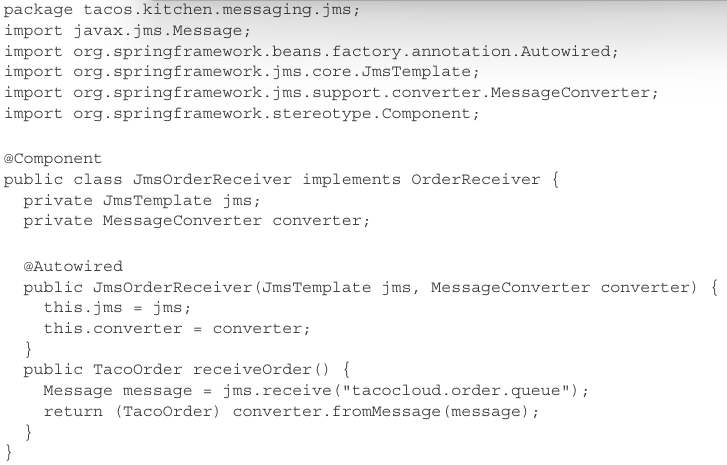
-You can use push model: define a message listener that’s invoked any time a message is available.

-Receiving with JmsTemplate

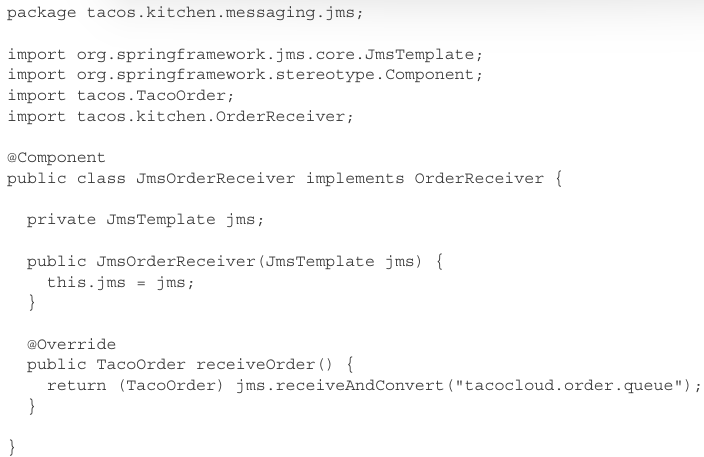
+JmsTemplate offers methods for pulling methods



+OrderReceiver service:

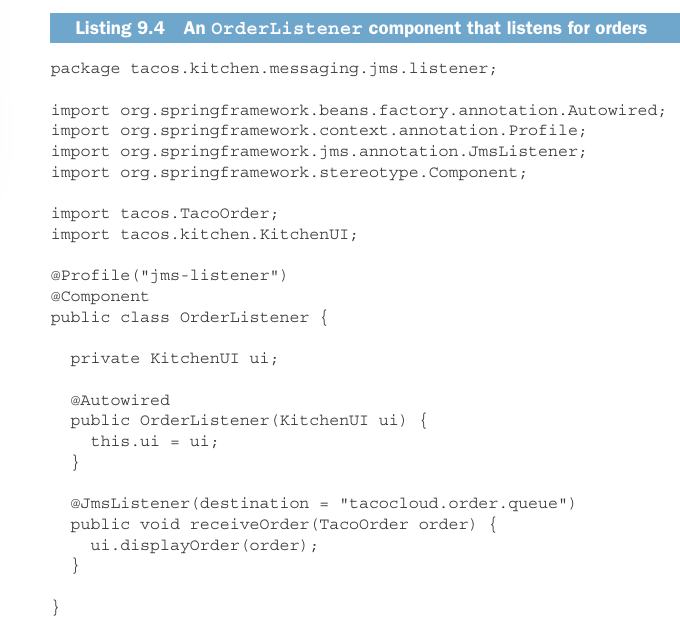


Use receiveAndConvert():



-**Declaring Message Listeners**

+Annotate a method in a component with @JmsListener



receiveOrder() with @JmsListener listen for messages on tacocloud.order.queue destination. This method is invoked automatically with the message’s TacoOrder payload as a parameter.

-JMS is defined by a standard Java specification and supported by many message broker implementation, it’s a common choice for messaging in Java. -> limited to Java apps. Newer messaging options are available for other languages and platforms beyond JVM.

# 9.2 Working with RabbitMQ and AMQP

## 9.2.1 Adding RabbitMQ to Spring

## 9.2.2 Sending messages with RabbitTemplate

## 9.2.3 Receiving messages from RabbitMQ

# 9.3 Messaging with Kafka

## 9.3.1 Setting up Spring for Kafka messaging

## 9.3.2 Sending messages with KafkaTempalte

## 9.3.3 Writing Kafka listeners

# -Summary