covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

12 Messaging in Spring

This chapter covers

- Introduction to the Java Message Service (JMS)
- Sending and receiving asynchronous messages
- Message-driven POJOs

It's 4:55 p.m. on Friday. You're minutes away from starting a much-anticipated vacation. You have just enough time to drive to the airport and catch your flight. But before you pack up and head out, you need to be sure that your boss and colleagues know the status of the work you've been doing so that they can pick up where you left off on Monday. Unfortunately, some of your colleagues have already skipped out for an early weekend departure… and your boss is tied up in a meeting. What do you do?

You could call your boss's cell phone… but it's not necessary to interrupt his meeting for a mere status report. Maybe you could stick around and wait until he returns from the meeting. But it's anyone's guess how long the meeting will last and you have a plane to catch. Perhaps you could leave a sticky note on his monitor… right next to 100 other sticky notes that it'll blend in with.

The most practical way to communicate your status and still catch your plane is to send a quick email to your boss and your colleagues, detailing your progress and promising to send a postcard. You don't know where they are or when they'll read the email, but you do know that they'll eventually return to their desks and read it. Meanwhile, you're on your way to the airport.

Sometimes it's necessary to talk to someone directly. If you injure yourself and need an ambulance, you're probably going to pick up the phone—emailing the hospital just won't do. But often, sending a message is sufficient and offers some advantages over direct communication, such as letting you get on with your vacation.

A couple of chapters back, you saw how to use RMI, Hessian, Burlap, HTTP invoker, and web services to enable communication between applications. All of these communication mechanisms employ synchronous communication in which a client application directly contacts a remote service and waits for the remote procedure to complete before continuing.

Synchronous communication has its place, but it's not the only style of interapplication communication available to developers. Asynchronous messaging is a way of indirectly sending messages from one application to another without waiting for a response. Asynchronous messaging has several advantages over synchronous messaging, as you'll soon see.

The Java Message Service (JMS) is a standard API for asynchronous messaging. In this chapter, we'll look at how Spring simplifies sending and receiving messages with JMS. In addition to basic sending and receiving of messages, we'll look at Spring's support for message-driven POJOs, a way to receive messages that resembles EJB's message-driven beans (MDBs).

12.1 A brief introduction to JMS

Much like the remoting mechanisms and REST interfaces we've covered so far in this part of the book, JMS is all about applications communicating with one another. JMS differs from those other mechanisms in how information is transferred between systems.

Remoting options such as RMI and Hessian/Burlap are synchronous. As illustrated in figure 12.1, when the client invokes a remote method, the client must wait for the method to complete before moving on. Even if the remote method doesn't return anything back to the client, the client will be put on hold until the service is done.

JMS, on the other hand, provides asynchronous communication between applications. When messages are sent asynchronously, as shown in figure 12.2, the client doesn't have to wait for the service to process the message or even for the message to be delivered. The client sends its message and then moves along with the assumption that the service will eventually receive and process the message.

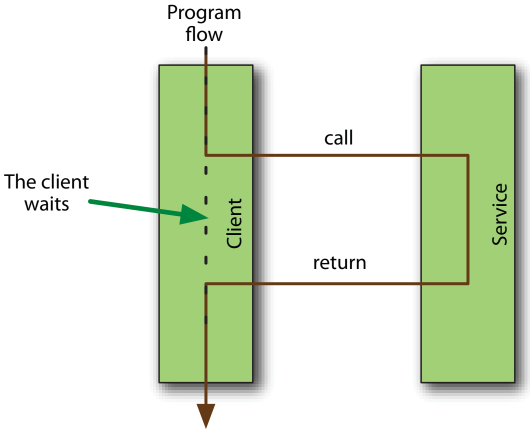

Figure 12.1 **When communicating synchronously, the client must wait for the operation to complete.**

Asynchronous communication through JMS offers several advantages over synchronous communication. We'll take a closer look at these advantages in a moment. But first, let's see how messages are sent using JMS.

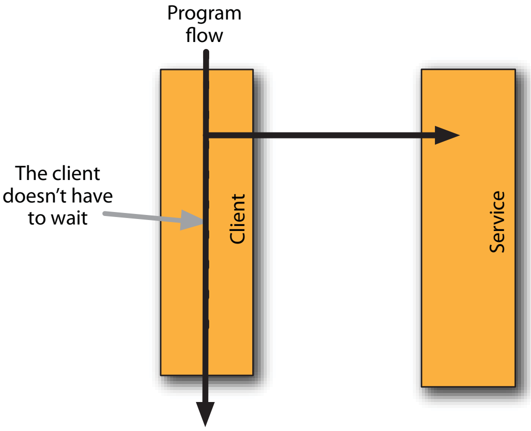

Figure 12.2 Asynchronous communication is a no-wait form of communication.

12.1.1 Architecting JMS

Most of us take the postal service for granted. Millions of times every day, people place letters, cards, and packages in the hands of postal workers, trusting that they'll get to the desired destination. The world's too big of a place for us to hand-deliver these things ourselves, so we rely on the postal system to handle it for us. We address them, place the necessary postage on them, and then drop them in the mail to be delivered without giving a second thought to how they might get there.

The key to the postal service is indirection. When Grandma's birthday comes around, it'd be inconvenient if we had to deliver a card directly to her. Depending on where she lives, we'd have to set aside anywhere from a few hours to a few days to deliver a birthday card. Fortunately, the postal service will deliver the card to her while we go about our lives.

Similarly, indirection is the key to JMS. When one application sends information to another through JMS, there's no direct link between the two applications. Instead, the sending application places the message in the hands of a service that will ensure delivery to the receiving application.

There are two main actors in JMS: *message brokers* and *destinations.*

When an application sends a message, it hands it off to a message broker. A message broker is JMS's analog of the post office. The message broker will ensure that the message is delivered to the specified destination, leaving the sender free to go about other business.

When you send a letter through the mail, it's important to address it so that the postal service knows where it should be delivered. Likewise, in JMS, messages are addressed with a destination. Destinations are like mailboxes where the messages are placed until someone comes to pick them up.

But unlike mail addresses, which may indicate a specific person or street address, destinations are less specific. Destinations are only concerned about *where* the message will be picked up—not *who* will pick them up. In this way, destinations are like sending a letter addressed, "To current resident."

In JMS, there are two types of destination: queues and topics. Each of these is associated with a specific messaging model, either point-to-point (for queues) or publish-subscribe (for topics).

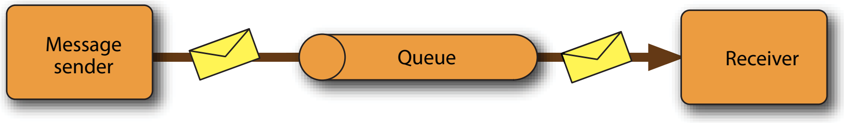

Figure 12.3 A message queue decouples a message sender from the message receiver. Though a queue may have several receivers, each message is picked up by exactly one receiver.

POINT-TO-POINT MESSAGING

In the point-to-point model, each message has exactly one sender and one receiver, as illustrated in figure 12.3. When the message broker is given a message, it places the message in a queue. When a receiver comes along and asks for the next message in the queue, the message is pulled from the queue and delivered to the receiver. Because the message is removed from the queue as it's delivered, it's guaranteed that the message will be delivered to only one receiver.

Although each message in a message queue is delivered to only one receiver, this doesn't imply that only one receiver is pulling messages from the queue. In fact, it's likely that several receivers are processing messages from the queue. But they'll each be given their own messages to process.

This is analogous to waiting in line at the bank. As you wait, you may notice that multiple tellers are available to help you with your financial transaction. After each customer is helped and a teller is freed up, she will call for the next person in line. When it's your turn at the front of the line, you'll be called to the counter and helped by one teller. The other tellers will help other banking customers.

Another observation to be made at the bank is that when you get in line, you probably won't know which teller will eventually help you. You could count how many people are in line, match that up with the number of available tellers, note which teller is fastest, and then come up with a guess as to which teller will call you to their window. But chances are you'll be wrong and end up at a different teller's window.

Likewise, in JMS, if multiple receivers are listening to a queue, there's no way of knowing which one will actually process a specific message. This uncertainty is a good thing because it enables an application to scale up message processing by simply adding another listener to the queue.

PUBLISH-SUBSCRIBE MESSAGING

In the publish-subscribe messaging model, messages are sent to a topic. As with queues, many receivers may be listening to a topic. But unlike queues, where a message is delivered to exactly one receiver, all subscribers to a topic will receive a copy of the message, as shown in figure 12.4.

As you may have guessed from its name, the publish-subscribe message model is much like the model of a magazine publisher and its subscribers. The magazine (a message) is published, sent to the postal service, and then all subscribers receive their own copy.

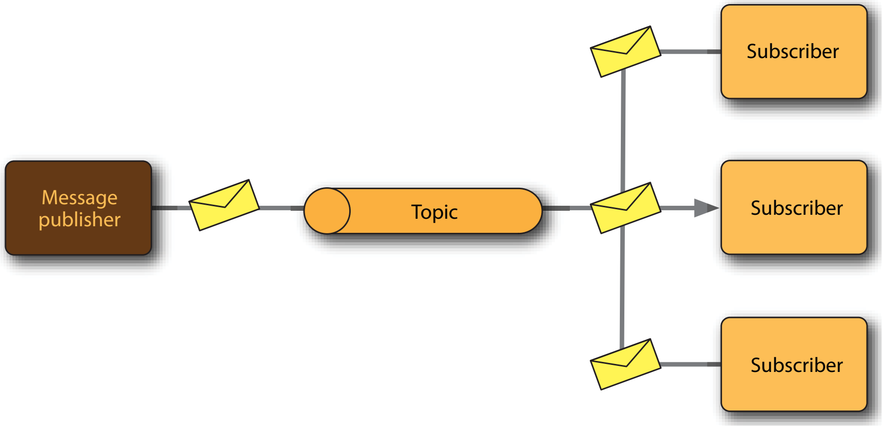

Figure 12.4 Like queues, topics decouple message senders from message receivers. Unlike queues, a topic message could be delivered to many topic subscribers.

The magazine analogy breaks down when you realize that in JMS, the publisher has no idea of who its subscribers are. The publisher only knows that its message will be published to a particular topic—not who's listening to that topic. This also implies that the publisher has no idea of how the message will be processed.

Now that we've covered the basics of JMS, let's see how JMS messaging compares to synchronous RPC.

12.1.2 Assessing the benefits of JMS

Even though it's intuitive and simple to set up, synchronous communication imposes several limitations on the client of a remote service. Most significantly:

- Synchronous communication implies waiting. When a client invokes a method on a remote service, it must wait for the remote method to complete before the client can continue. If the client communicates frequently with the remote service and/or the remote service is slow to respond, this could negatively impact performance of the client application.
- The client is coupled to the service through the service's interface. If the interface of the service changes, all of the service's clients will also need to change accordingly.
- The client is coupled to the service's location. A client must be configured with the service's network location so that it knows how to contact the service. If the network topology changes, the client will need to be reconfigured with the new location.
- The client is coupled to the service's availability. If the service becomes unavailable, the client is effectively crippled.

Though synchronous communication has its place, these shortcomings should be taken into account when deciding what communication mechanism is a best fit for your application's needs. If these constraints are a concern for you, you may want to consider how asynchronous communication with JMS addresses these issues.

NO WAITING

When a message is sent with JMS, the client doesn't need to wait around for it to be processed or even delivered. The client drops the message off with the message broker and moves along with faith that the message will make it to the appropriate destination.

Since it doesn't have to wait, the client will be freed up to perform other activities. With all of this free time, the client's performance can be dramatically improved.

MESSAGE ORIENTATION AND DECOUPLING

Unlike RPC communication that's typically oriented around a method call, messages sent with JMS are data-centric. This means that the client isn't fixed to a specific method signature. Any queue or topic subscriber that can process the data sent by the client can process the message. The client doesn't need to be aware of any service specifics.

LOCATION INDEPENDENCE

Synchronous RPC services are typically located by their network address. The implication of this is that clients aren't resilient to changes in network topology. If a service's IP address changes or if it's configured to listen on a different port, the client must be changed accordingly or the client will be unable to access the service.

In contrast, JMS clients have no idea who will process their messages or where the service is located. The client only knows the queue or topic through which the messages will be sent. As a result, it doesn't matter where the service is located, as long as it can retrieve messages from the queue or topic.

In the point-to-point model, it's possible to take advantage of location independence to create a cluster of services. If the client is unaware of the service's location and if the service's only requirement is that it must be able to access the message broker, there's no reason why multiple services can't be configured to pull messages from the same queue. If the service is being overburdened and falling behind in its processing, all we need to do is turn up a few more instances of the service to listen to the same queue.

Location independence takes on another interesting side effect in the publish-subscribe model. Multiple services could all subscribe to a single topic, receiving duplicate copies of the same message. But each service could process that message differently. For example, let's say you have a set of services which together process a message that details the new hire of an employee. One service might add the employee to the payroll system, another to the HR portal, and yet another makes sure that the employee is given access to the systems they'll need to do their job. Each service works independently on the same data that they each received from a topic.

GUARANTEED DELIVERY

In order for a client to communicate with a synchronous service, the service must be listening at the IP address and port specified. If the service were to go down or otherwise become unavailable, the client wouldn't be able to proceed.

But when sending messages with JMS, the client can rest assured that its messages will be delivered. Even if the service is unavailable when a message is sent, it'll be stored until the service is available again.

Now that you have a feel for the basics of JMS and asynchronous messaging, let's set up a JMS message broker that we'll use in our examples. Although you're free to use any JMS message broker you'd like, we'll use the popular ActiveMQ message broker.

12.2 Setting up a message broker in Spring

ActiveMQ is a great open source message broker and a wonderful option for asynchronous messaging with JMS. As I'm writing this, the current version of ActiveMQ is 5.4.2. To get started with ActiveMQ, you'll need to download the binary distribution from http://activemq.apache.org. Once you've downloaded ActiveMQ, unzip it to your local hard drive. In the lib directory of the unzipped distribution, you'll find activemq-core-5.4.2.jar. This is the JAR file you'll need to add to the application's classpath to be able to use ActiveMQ's API.

Under the bin directory, you'll find several subdirectories for various operating systems. Within those, you'll find scripts that you can use to start ActiveMQ. For example, to start ActiveMQ on Mac OS X, run `activemq start` from the bin/macosx directory. Within moments ActiveMQ will be ready and waiting to broker your messages.

12.2.1 Creating a connection factory

Throughout this chapter, we're going to see different ways that Spring can be used to both send and receive messages through JMS. In all cases, we'll need a JMS connection factory to be able to send messages through the message broker. Since we're using ActiveMQ as our message broker, we'll have to configure the JMS connection factory so that it knows how to connect to ActiveMQ. `ActiveMQConnectionFactory` is the JMS connection factory that comes with ActiveMQ, and it's configured in Spring like this:

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Optionally, since we know that we're dealing with ActiveMQ, we can use ActiveMQ's own Spring configuration namespace (available with all versions of ActiveMQ since version 4.1) to declare the connection factory. First, be sure to declare the `amq` namespace in the Spring configuration XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:jms="http://www.springframework.org/schema/jms"
 xmlns:amq="http://activemq.apache.org/schema/core"
 xsi:schemaLocation="http://activemq.apache.org/schema/core
   http://activemq.apache.org/schema/core/activemq-core-5.5.0.xsd
   http://www.springframework.org/schema/jms
   http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
 ...
</beans>
```

Then we can use the <amq:connectionFactory> element to declare the connection factory:

```
<amq:connectionFactory id="connectionFactory"
      brokerURL="tcp://localhost:61616"/>
```

Note that the <amq:connectionFactory> element is clearly specific to ActiveMQ. If you're using a different message broker implementation, there may or may not be a Spring configuration namespace available. If not, then you'll need to wire the connection factory as a <bean>.

Later in this chapter we'll use this connectionFactory bean a lot. But for now, suffice it to say that the brokerURL tells the connection factory where the message broker is located. In this case, the URL given to brokerURL tells the connection factory to connect to ActiveMQ on the local machine at port 61616 (which is the port that ActiveMQ listens to by default).

12.2.2 Declaring an ActiveMQ message destination

In addition to a connection factory, we'll need a destination for the messages to be passed along to. The destination can be either a queue or a topic, depending on the needs of the application.

Regardless of whether you're using a queue or a topic, you must configure the destination bean in Spring using a message broker–specific implementation class. For example, the following <bean> declaration declares an ActiveMQ queue:

```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
  <constructor-arg value="spitter.queue"/>
</bean>
```

Similarly, the following <bean> declares a topic for ActiveMQ:

```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
  <constructor-arg value="spitter.topic"/>
</bean>
```

In either case, the <constructor-arg> specifies the name of the queue, as it's known to the message broker—spitter.topic in this case.

As with the connection factory, the ActiveMQ namespace offers an alternative way to declare queues and topics. For queues, we could also use the <amq:queue> element:

```
<amq:queue id="queue" physicalName="spitter.queue" />
```

Or, if it's a JMS topic that's in order, use the <amq:topic>:

```
<amq:topic id="topic" physicalName="spitter.topic" />
```

Either way, the physicalName attribute sets the name of the message channel.

At this point we've seen how to declare the essential components of working with JMS, whether you're sending or receiving messages. Now we're ready to start sending and receiving messages. For that, we'll use Spring's JmsTemplate, the centerpiece of Spring's JMS support. But first, let's gain an appreciation for what JmsTemplate provides by looking at what JMS is like without JmsTemplate.

12.3 Using Spring's JMS template

As you've seen, JMS gives Java developers a standard API for interacting with message brokers and for sending and receiving messages. Furthermore, virtually every message broker implementation out there supports JMS. So there's no reason to learn a proprietary messaging API for every message broker you deal with.

But though JMS offers a universal interface to all message brokers, its convenience comes at a cost. Sending and receiving messages with JMS isn't a simple matter of licking a stamp and placing it on an envelope. As you'll see, JMS demands that you also (figuratively) fuel up the mail carrier's truck.

12.3.1 Tackling runaway JMS code

In section 5.3.1 I showed you how conventional JDBC code can be an unwieldy mess of code to handle connections, statements, result sets, and exceptions. Unfortunately, conventional JMS follows a similar model, as you'll observe in the following listing.

Listing 12.1 Sending a message using conventional (non-Spring) JMS

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination = new ActiveMQQueue("spitter.queue");
  MessageProducer producer = session.createProducer(destination);
  TextMessage message = session.createTextMessage();

  message.setText("Hello world!");
  producer.send(message);                         <— Send message
} catch (JMSException e) {
  // handle exception?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

At the risk of sounding repetitive—holy runaway code, Batman! As with the JDBC example, there are almost 20 lines of code here just to send a simple "Hello world!" message. Only a few of those actually send the message; the rest are merely setting the stage for sending the message.

It isn't much better on the receiving end, as you can see in the following listing.

Listing 12.2 Receiving a message using conventional (non-Spring) JMS

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
  conn = cf.createConnection();
  conn.start();
  session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
  Destination destination =
      new ActiveMQQueue("spitter.queue");
  MessageConsumer consumer = session.createConsumer(destination);
  Message message = consumer.receive();
  TextMessage textMessage = (TextMessage) message;
  System.out.println("GOT A MESSAGE: " + textMessage.getText());
  conn.start();
} catch (JMSException e) {
  // handle exception?
} finally {
  try {
    if (session != null) {
      session.close();
    }
    if (conn != null) {
      conn.close();
    }
  } catch (JMSException ex) {
  }
}
```

Again, just as in listing 12.1, that's a lot of code to do something so darn simple. If you take a line-by-line comparison, you'll find that they're almost identical. And if you were to look at a thousand other JMS examples, you'd find them all to be strikingly similar. Some may retrieve their connection factories from JNDI and some may use a topic instead of a queue. Nevertheless, they all follow roughly the same pattern.

A consequence of all of this boilerplate code is that you'll find that you repeat yourself every time you work with JMS. Worse still, you'll find yourself repeating other developers' JMS code.

We've already seen in chapter 5 how Spring's `JdbcTemplate` handles runaway JDBC boilerplate. Now let's look at how Spring's `JmsTemplate` can do the same thing for JMS boilerplate code.

12.3.2 Working with JMS templates

`JmsTemplate` is Spring's answer to verbose and repetitive JMS code. `JmsTemplate` takes care of creating a connection, obtaining a session, and ultimately sending or receiving messages. This leaves you to focus your development efforts on constructing the message to send or processing the messages that are received.

What's more, `JmsTemplate` can handle any clumsy `JMSException` that may be thrown along the way. If a `JMSException` is thrown in the course of working with

JmsTemplate, JmsTemplate will catch it and rethrow it as one of the unchecked subclasses of Spring's own JmsException.

Table 12.1 shows how Spring maps standard JMSExceptions to Spring's unchecked JmsExceptions.

In fairness to the JMS API, JMSException does come with a rich and descriptive set of subclasses that give you a better sense of what went wrong. Nevertheless, all of these subclasses of JMSException are checked exceptions and thus must be caught. JmsTemplate will attend to that for you by catching those exceptions and rethrowing an appropriate unchecked subclass of JmsException.

Table 12.1 Spring's JmsTemplate catches standard JMSExceptions and rethrows them as unchecked subclasses of Spring's own JmsException.

Spring (org.springframework.jms.*)	Standard JMS (javax.jms.*)
DestinationResolutionException	Spring-specific—thrown when Spring can't resolve a destination name
IllegalStateException	IllegalStateException
InvalidClientIDException	InvalidClientIDException
InvalidDestinationException	InvalidDestinationException
InvalidSelectorException	InvalidSelectorException
JmsSecurityException	JmsSecurityException
ListenerExecutionFailedException	Spring-specific—thrown when execution of a listener method fails
MessageConversionException	Spring-specific—thrown when message conversion fails
MessageEOFException	MessageEOFException
MessageFormatException	MessageFormatException
MessageNotReadableException	MessageNotReadableException
MessageNotWriteableException	MessageNotWriteableException
ResourceAllocationException	ResourceAllocationException
SynchedLocalTransactionFailedException	Spring-specific—thrown when a synchronized local transaction fails to complete
TransactionInProgressException	TransactionInProgressException
TransactionRolledBackException	TransactionRolledBackException
UncategorizedJmsException	Spring-specific—thrown when no other exception applies

A tale of two `JmsTemplates`

Spring actually comes with two JMS template classes: `JmsTemplate` and `JmsTemplate102`. `JmsTemplate102` is a special version of `JmsTemplate` for JMS 1.0.2 providers. In JMS 1.0.2, topics and queues are treated as completely different concepts known as *domains*. In JMS 1.1+, topics and queues are unified under a domain-independent API. Because topics and queues are treated so differently in JMS 1.0.2, there has to be a special `JmsTemplate102` for interacting with older JMS implementations. In this chapter, we'll assume a modern JMS provider and therefore will focus our attention on `JmsTemplate`.

WIRING A JMS TEMPLATE

To use `JmsTemplate`, we'll need to declare it as a bean in the Spring configuration file. The following XML should do the trick:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

Because `JmsTemplate` needs to know how to get connections to the message broker, we must set the `connectionFactory` property with a reference to the bean that implements JMS's `ConnectionFactory` interface. Here, we've wired it with a reference to the `connectionFactory` bean that we declared earlier in section 12.2.1.

That's all you need to do to configure `JmsTemplate`—it's now ready to go. Let's start sending messages!

SENDING MESSAGES

One of the features we'd like to build into the Spitter application is the option of alerting (perhaps by email) other users whenever a spittle has been created. We could build that feature directly into the application at the point where a spittle is added. But figuring out who to send alerts to and actually sending those alerts may take a while and it could hurt the perceived performance of the application. When a new spittle is added, we want the application to be snappy and respond quickly with a response.

Rather than taking the time to send those messages at the moment when the spittle is added, it makes more sense to queue up that work and deal with it later, after the response has gone back to the user. The time it takes to send a message to a message queue or a topic is negligible, especially compared to the time it may take to send the alerts to other users.

To support sending spittle alerts asynchronously with the creation of spittles, let's introduce `AlertService` to the Spittle application:

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

As you can see, AlertService is a interface that defines a single operation, sendSpittleAlert(). AlertServiceImpl is an implementation of the AlertService interface that uses JmsTemplate to send Spittle objects to a message queue to be processed at some later time.

Listing 12.3 Sending a Spittle using JmsTemplate

```
package com.habuma.spitter.alerts;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

import com.habuma.spitter.domain.Spittle;

public class AlertServiceImpl implements AlertService {
  public void sendSpittleAlert(final Spittle spittle) {
    jmsTemplate.send(                                          Sends message
      "spittle.alert.queue",              Specifies destination
      new MessageCreator() {
        public Message createMessage(Session session)
                throws JMSException {
          return session.createObjectMessage(spittle);         Creates message
        }
      }
    );
  }

  @Autowired                                  Inject JMS
  JmsTemplate jmsTemplate;                    template
}
```

The first parameter to the JmsTemplate's send() method is the name of the JMS destination that the message will be sent to. When the send() method is called, JmsTemplate will deal with obtaining a JMS connection and session and will send the message on behalf of the sender (see figure 12.5).

As for the message itself, it's constructed using a MessageCreator, implemented here as an anonymous inner class. In MessageCreator's createMessage() method, we simply ask for an object message from the session, giving it the Spittle object to build the object message from.

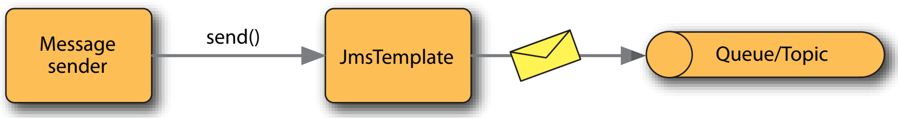

Figure 12.5 JmsTemplate **deals with the complexities of sending a message on behalf of the sender.**

And that's it! Note that the `sendSpittleAlert()` method is focused entirely on assembling and sending a message. There's no connection or session management code; `JmsTemplate` handles all of that for us. And there's no need to catch `JMSException`; `JmsTemplate` will catch any `JMSException` that's thrown and then rethrow it as one of Spring's unchecked exceptions from table 12.1.

SETTING A DEFAULT DESTINATION

In listing 12.3, we explicitly specified a specific destination that the spittle message would be sent to in the `send()` method. That form of `send()` method comes in handy when we want to programmatically choose a destination. But in the case of `AlertServiceImpl`, we'll always be sending the spittle message to the same destination, so the benefits of that form of `send()` aren't as clear.

Instead of explicitly specifying a destination each time we send a message, we could opt for wiring a default destination into `JmsTemplate`:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="defaultDestinationName"
            value="spittle.alert.queue"/>
</bean>
```

Now the call to `JmsTemplate`'s `send()` method can be simplified slightly by removing the first parameter:

```
jmsTemplate.send(
  new MessageCreator() {
  ...
  }
);
```

This form of the `send()` method only takes a `MessageCreator`. There's no need to specify a destination because the default destination is the one we want to send messages to.

CONSUMING MESSAGES

Now you've seen how to send a message using `JmsTemplate`. But what if you're on the receiving end? Can `JmsTemplate` be used to receive messages too?

Yes, it can. In fact, it's even easier to receive messages with `JmsTemplate`. All you need to do is call `JmsTemplate`'s `receive()` method, as shown.

Listing 12.4 Receiving a message using `JmsTemplate`

```
public Spittle getAlert() {
  try {
    ObjectMessage receivedMessage =
        (ObjectMessage) jmsTemplate.receive();          Receive message

    return (Spittle) receivedMessage.getObject();          Get object
  } catch (JMSException jmsException) {
    throw JmsUtils.convertJmsAccessException(jmsException);
  }
}
```

Throw converted exception

When the JmsTemplate's receive() method is called, it'll attempt to retrieve a message from the message broker. If no message is available, the receive() method will wait until a message becomes available. This interaction is illustrated in figure 12.6.

Since we know that the spittle message was sent as an object message, it can be cast to ObjectMessage upon arrival. After that, we call getObject() to extract the Spittle object from the ObjectMessage and return it.

The one gotcha here is that we have to do something about the JMSException that may be thrown. As I already mentioned, JmsTemplate is good about handling any checked JMSExceptions that are thrown and then rethrowing them as one of Spring's unchecked JmsExceptions. But that's only applicable when calling one of JmsTemplate's methods. JmsTemplate can't do much about the JMSException that may be thrown by the call to ObjectMessage's getObject() method.

Therefore, we must either catch that JMSException or declare that the method throws it. In keeping with Spring's philosophy of avoiding checked exceptions, we don't want to let the JMSException escape this method, so we'll catch it instead. In the catch block, we can use the convertJmsAccessException() method from Spring's JmsUtils class to convert the checked JMSException to an unchecked JmsException. This is effectively the same thing that JmsTemplate does for us in other cases.

The big downside of consuming messages with JmsTemplate is that the receive() method is synchronous. This means that the receiver must wait patiently for the message to arrive, as the receive() message will block until a message is available (or until a timeout condition occurs). Doesn't it seem odd to synchronously consume a message that was asynchronously sent?

That's where message-driven POJOs come in handy. Let's see how to receive messages asynchronously using components that react to messages rather than waiting on them.

The big downside of consuming messages with JmsTemplate is that the receive() method is synchronous. This means that the receiver must wait patiently for the message to arrive, as the receive() method will block until a message is available (or until a timeout condition occurs). Doesn't it seem odd to synchronously consume a message that was asynchronously sent?

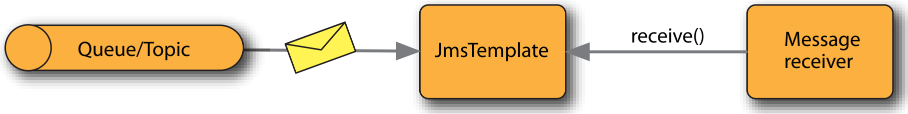

Figure 12.6 **Receiving messages from a topic or queue using JmsTemplate is as simple as calling the receive() method. JmsTemplate takes care of the rest.**

12.4 Creating message-driven POJOs

During one summer in college, I had the privilege of working in Yellowstone National Park. The job wasn't one of the high-profile jobs like park ranger or the guy who turns Old Faithful on and off. Instead, I held a position in housekeeping at Old Faithful Inn, changing sheets, cleaning bathrooms, and vacuuming floors. Not glamorous, but at least I was working in one of the most beautiful places on Earth.

Every day after work, I'd head over to the local post office to see if I had any mail. I was away from home for several weeks, so it was nice to receive a letter or card from my friends back at school. I didn't have my own post box, so I'd walk up and ask the man sitting on the stool behind the counter if I had received any mail. That's when the wait would begin.

You see, the man behind the counter was approximately 195 years old. And like most people that age he had a difficult time getting around. He'd drag his keister off the stool, slowly scoot his feet across the floor, and then disappear behind a partition. After a few moments, he'd emerge, shuffle his way back to the counter, and lift himself back up onto the stool. Then he'd look at me and say, "No mail today."

`JmsTemplate`'s `receive()` method is a lot like that aged postal employee. When you call `receive()`, it goes away and looks for a message in the queue or topic and doesn't return until a message arrives or until the timeout has passed. Meanwhile, your application is sitting there doing nothing, waiting to see if there's a message. Wouldn't it be better if your application could go about its business and be notified when a message arrives?

One of the highlights of the EJB 2 specification was the inclusion of the *message-driven bean (MDB)*. MDBs are EJBs that process messages asynchronously. In other words, MDBs react to messages in a JMS destination as events and respond to those events. This is in contrast to synchronous message receivers, which block until a message is available.

MDBs were a bright spot in the EJB landscape. Even many of EJB's most rabid detractors would concede that MDBs were an elegant way of handling messages. The only blemish to be found in EJB 2 MDBs was that they had to implement `javax.ejb.MessageDrivenBean`. In doing so, they also had to implement a few EJB lifecycle callback methods. Put simply, EJB 2 MDBs were very un-POJO.

With the EJB 3 specification, MDBs were cleaned up to have a slightly more POJO feel to them. No longer must you implement the `MessageDrivenBean` interface. Instead, you implement the more generic `javax.jms.MessageListener` interface and annotate MDBs with `@MessageDriven`.

Spring 2.0 addresses the need for asynchronous consumption of messages by providing its own form of message-driven bean that's quite similar to EJB 3's MDBs. In this section, you'll learn how Spring supports asynchronous message consumption using message-driven POJOs (we'll call them *MDPs*, for short).

12.4.1 Creating a message listener

If we were to build our spittle alert handler using EJB's message-driven model, it'd need to be annotated with `@MessageDriven`. And, although it's not strictly required, it's recommended that the MDB implement the `MessageListener` interface. The result would look something like this:

```
@MessageDriven(mappedName="jms/spittle.alert.queue")
public class SpittleAlertHandler implements MessageListener {
  @Resource
  private MessageDrivenContext mdc;

  public void onMessage(Message message) {
     ...
  }
}
```

For a moment, try to imagine a simpler world where message-driven components don't have to implement the `MessageListener` interface. In such a happy place, the sky would be the brightest of blues, the birds would always whistle your favorite song, and you wouldn't have to implement the `onMessage()` method or have a `MessageDrivenContext` injected.

Okay, maybe the demands placed on an MDB by the EJB 3 specification aren't that arduous. But the fact is that the EJB 3 implementation of `SpittleAlertHandler` is too tied to EJB's message-driven APIs and isn't as POJO-ish as we'd like. Ideally, we'd like the alert handler to be capable of handling messages, but not coded as if it knows that's what it'll be doing.

Spring offers the ability for a method on a POJO to handle messages from a JMS queue or topic. For example, the following POJO implementation of `SpittleAlertHandler` is perfectly sufficient.

Listing 12.5 A Spring MDP asynchronously receives and processes messages.

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public class SpittleAlertHandler {

  public void processSpittle(Spittle spittle) {       <-- Handler method
    // ... implementation goes here...
  }

}
```

Although the color of the sky and training birds to sing are out of scope for Spring, listing 12.5 shows that the dream world I described is much closer to reality. We'll fill in the details of the `processSpittle()` method later. For now, consider that nothing in `SpittleAlertHandler` shows any hint of JMS. It's a POJO in every sense of the term. It can nevertheless handle messages just like its EJB cousin. All it needs is some special Spring configuration.

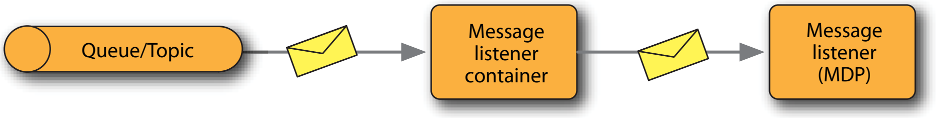

Figure 12.7 **A message listener container listens to a queue/topic. When a message arrives, it's forwarded to a message listener (such as a message-driven POJO).**

12.4.2 Configuring message listeners

The trick to empowering a POJO with message-receiving abilities is to configure it as a message listener in Spring. Spring's `jms` namespace provides everything we need to do that. First, we must declare the handler as a `<bean>`:

```
<bean id="spittleHandler"
      class="com.habuma.spitter.alerts.SpittleAlertHandler" />
```

Then, to turn `SpittleAlertHandler` into a message-driven POJO, we can declare the bean to be a message listener:

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
       ref="spittleHandler" method="processSpittle" />
</jms:listener-container>
```

Here we have a message listener that's contained within a message listener container. A *message listener container* is a special bean that watches a JMS destination, waiting for a message to arrive. Once a message arrives, it retrieves the message and then passes it on to any message listeners that are interested. Figure 12.7 illustrates this interaction.

To configure the message listener container and message listener in Spring, we're using two elements from Spring's `jms` namespace. The `<jms:listener-container>` is used to contain `<jms:listener>` elements. Here its `connectionFactory` attribute is configured with a reference to the `connectionFactory` that's to be used by each of the child `<jms:listener>`s as they listen for messages. In this case, the `connectionFactory` attribute could've been left off because it defaults to `connectionFactory`.

Regarding the `<jms:listener>` element, it's used to identify a bean and a method that should handle incoming messages. For the purposes of handling spittle alert messages, the `ref` element refers to our `spittleHandler` bean. When a message arrives on spitter.alert.queue (as designated by the `destination` attribute), the `spittleHandler` bean's `processSpittle()` method gets the call (per the `method` attribute).

12.5 Using message-based RPC

In chapter 10, we explored several of Spring's options for exposing bean methods as remote services and for making calls on those services from clients. In this chapter, we've seen how to send messages between applications over message queues and topics. Now we're going to bring those two concepts together and see how to make remote calls that use JMS as a transport.

There are two options for message-based RPC in Spring:

- Spring itself offers JmsInvokerServiceExporter for exporting beans as message-based services and JmsInvokerProxyFactoryBean for clients to consume those services.
- Lingo provides a similar approach to message-based remoting with its JmsServiceExporter and JmsProxyFactoryBean.

As you'll see, these two options are very similar to each other, but each has advantages and disadvantages. I'll show you both approaches and let you decide which works best for you. Let's start by looking at how to work with Spring's own support for JMS-backed services.

12.5.1 Working with Spring message-based RPC

As you'll recall from chapter 10, Spring provides several options for exporting beans as remote services. We used RmiServiceExporter to export beans as RMI services over JRMP, HessianExporter and BurlapExporter for Hessian and Burlap services over HTTP, and HttpInvokerServiceExporter to create HTTP invoker services over HTTP. But Spring has one more service exporter that we didn't talk about in chapter 10.

CONSUMING JMS-BASED SERVICES

JmsInvokerServiceExporter is much like those other service exporters. In fact, note that there's some symmetry in the names of JmsInvokerServiceExporter and HttpInvokerServiceExporter. If HttpInvokerServiceExporter exports services that communicate over HTTP, then JmsInvokerServiceExporter must export services that converse over JMS.

To demonstrate how JmsInvokerServiceExporter works, consider AlertServiceImpl.

Listing 12.6 AlertServiceImpl is a JMS-free POJO that will handle JMS messages.

```
package com.habuma.spitter.alerts;

import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;

import com.habuma.spitter.domain.Spittle;

@Component("alertService")
public class AlertServiceImpl implements AlertService {
  private JavaMailSender mailSender;
  private String alertEmailAddress;
  public AlertServiceImpl(JavaMailSender mailSender,
                          String alertEmailAddress) {
    this.mailSender = mailSender;
    this.alertEmailAddress = alertEmailAddress;
  }

  public void sendSpittleAlert(final Spittle spittle) {       Send Spittle alert
    SimpleMailMessage message = new SimpleMailMessage();
    String spitterName = spittle.getSpitter().getFullName();
    message.setFrom("noreply@spitter.com");
    message.setTo(alertEmailAddress);
    message.setSubject("New spittle from " + spitterName);
    message.setText(spitterName + " says: " + spittle.getText());
    mailSender.send(message);
  }
}
```

Don't concern yourself too much with the inner details of the `sendSpittleAlert()` method at this point. We'll talk more about how to send emails with Spring later, in section 14.3. The important thing to notice is that `AlertServiceImpl` is a simple POJO and has nothing that indicates that it'll be used to handle JMS messages. It does implement the simple `AlertService` interface, as shown here:

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public interface AlertService {
  void sendSpittleAlert(Spittle spittle);
}
```

As you can see, `AlertServiceImpl` is annotated with `@Component` so that it'll be automatically discovered and registered as a bean in the Spring application context with an ID of `alertService`. We'll refer to this bean as we configure a `JmsInvokerServiceExporter`:

```
<bean id="alertServiceExporter"
      class="org.springframework.jms.remoting.JmsInvokerServiceExporter"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

This bean's properties describe what the exported service should look like. The `service` property is wired to refer to the `alertService` bean, which is the implementation of the remote service. Meanwhile, the `serviceInterface` property is set to the fully-qualified class name of the interface that the service provides.

The exporter's properties don't describe the specifics of how the service will be carried over JMS. But the good news is that `JmsInvokerServiceExporter` qualifies as a JMS listener. Therefore, we can configure it as such within a `<jms:listener-container>` element:

```
<jms:listener-container connection-factory="connectionFactory">
  <jms:listener destination="spitter.alert.queue"
       ref="alertServiceExporter" />
</jms:listener-container>
```

The JMS listener container is given the connection factory so that it can know how to connect to the message broker. Meanwhile, the `<jms:listener>` declaration is given the destination that the remote message will be carried on.

CONSUMING JMS-BASED SERVICES

At this point, the JMS-based alert service should be ready and waiting for RPC messages to arrive on the queue whose name is spitter.alert.queue. On the client side, `JmsInvokerProxyFactoryBean` will be used to access the service.

JmsInvokerProxyFactoryBean is a lot like the other remoting proxy factory beans that we looked at in chapter 10. It hides the details of accessing a remote service behind a convenient interface, through which the client interacts with the service. The big difference is that instead of proxying RMI- or HTTP-based services, JmsInvokerProxyFactoryBean proxies a JMS-based service that was exported by JmsInvokerServiceExporter.

To consume the alert service, we can wire the JmsInvokerProxyFactoryBean like this:

```
<bean id="alertService"
    class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="queueName" value="spitter.alert.queue" />
  <property name="serviceInterface"
          value="com.habuma.spitter.alerts.AlertService" />
</bean>
```

The connectionFactory and queueName properties specify how RPC messages should be delivered—here, on the queue named spitter.alert.queue at the message broker configured in the given connection factory. As for the serviceInterface, this specifies that the proxy should be exposed through the AlertService interface.

JmsInvokerServiceExporter and JmsInvokerProxyFactoryBean offer an JMS-based alternative to Spring's other remoting options. But it's not the only way to export beans and consume JMS-based services. It may not even be the best way. Let's look at Lingo and see how it offers something that the JMS invoker doesn't.

12.5.2 Asynchronous RPC with Lingo

Lingo[1] is a Spring-based remoting option that's similar to Spring's own JMS invoker support. In fact, the Javadoc for Spring's JMS invoker classes even indirectly gives credit to Lingo as their inspiration.[2]

What makes Lingo different is that, unlike the JMS invoker, it can truly take advantage of the asynchronous nature of JMS to invoke services asynchronously. That means that the server doesn't have to even be available when the client makes the call. Furthermore, if the service is long-running, then the client won't have to wait for it to finish.

Unlike the other remoting option we discussed in chapter 10 or even Spring's own JMS invoker classes, Lingo isn't part of the Spring Framework. It's a separate project that builds upon Spring Remoting, offering a JMS-based service exporter and client proxy.

We'll start our exploration of Lingo by seeing how to export services with Lingo's JmsServiceExporter. Then we'll consume that service using Lingo's JmsServiceProxy.

[1] http://lingo.codehaus.org

[2] The Javadoc doesn't mention Lingo, but it does give credit to James Strachan, Lingo's creator.

EXPORTING THE ASYNCHRONOUS SERVICE

As you can see from the following `<bean>` declaration, `JmsServiceExporter` and `JmsInvokerServiceExporter` are configured in much the same way:

```
<bean id="alertServiceExporter"
      class="org.logicblaze.lingo.jms.JmsServiceExporter"
      p:connectionFactory-ref="connectionFactory"
      p:destination-ref="alertServiceQueue"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

The `service` and `serviceInterface` properties are exactly the same as with `JmsInvokerServiceExporter`. But a new property is injected on the `JmsServiceExporter` bean. `JmsServiceExporter` can't be used as a message listener in a Spring listener container, so we must tell it about the JMS connection factory and the message destination in the `connectionFactory` and `destination` properties so that it knows how to send the message.

Note that the `destination` property is of `javax.jms.Destination`. So we'll need to wire in a reference to a destination bean. The following `alertServiceQueue` bean will make sure that JMS RPC messages are transported over the queue named spittle .alert.queue:

```
<amq:queue id="alertServiceQueue"
           physicalName="spitter.alert.queue" />
```

At this point, Lingo hasn't given us anything that Spring's own JMS invoker didn't provide. So you may be wondering why I'd bother telling you about Lingo if Spring's own JMS RPC mechanism provides effectively the same capabilities.

As you're about to see, the client side of Lingo offers something that the JMS invoker doesn't: asynchronous invocation.

PROXYING ASYNCHRONOUS SERVICES

When we called methods on a proxy created by Spring's `JmsInvokerServiceProxy`, we had to wait. Even though the underlying transport was JMS, the proxy would wait until it received a response.

Lingo's `JmsProxyFactoryBean`, on the other hand, can be configured to treat `void` methods as one-way asynchronous methods. For example, the client side of a Lingo-based alert service, might be configured something like this:

```
<bean id="alertService"
      class="org.logicblaze.lingo.jms.JmsProxyFactoryBean"
  p:connectionFactory-ref="connectionFactory"
  p:destination-ref="queue"
  p:serviceInterface="com.habuma.spitter.alerts.AlertService">
   <property name="metadataStrategy">
     <bean id="metadataStrategy"
           class="org.logicblaze.lingo.SimpleMetadataStrategy">
       <constructor-arg value="true"/>
     </bean>
   </property>
</bean>
```

The connectionFactory, destination, and serviceInterface properties serve the same purpose as in previous examples. What's new here is the metadataStrategy property, which we've set using an inner-bean declaration of type SimpleMetadataStrategy.

Among other things, a metadata strategy is Lingo's way of determining which methods should be asynchronous one-way operations. The only implementation available is SimpleMetadataStrategy, whose constructor can take a single argument Boolean value to indicate whether void methods should be asynchronous. Here, we've declared the constructor argument as true, indicating that any void methods on the service should be considered one-way methods and thus should be asynchronous and immediately return.

Had we declared the constructor argument as false or not injected JmsProxyFactoryBean's metadataStrategy property at all, all service methods would be treated synchronously and JmsProxyFactoryBean would be roughly equivalent in power to Spring's JmsInvokerServiceProxy.

12.6 Summary

Asynchronous messaging presents several advantages over synchronous RPC. Indirect communication results in applications that are loosely coupled with respect to one another, and thus reduces the impact of any one system going down. Additionally, because messages are forwarded to their recipients, there's no need for a sender to wait for a response. In many circumstances, this can be a boost to application performance.

Although JMS provides a standard API for all Java applications wishing to participate in asynchronous communication, it can be cumbersome to use. Spring eliminates the need for JMS boilerplate code and exception-handling code and makes asynchronous messaging easier to use.

In this chapter, we've seen several ways that Spring can help establish asynchronous communication between two applications by way of message brokers and JMS. Spring's JMS template eliminates the boilerplate that's commonly required by the traditional JMS programming model. And Spring-enabled message-driven beans make it possible to declare bean methods which react to messages that arrive in a queue or topic.

We also looked at using Spring's JMS invoker as well as Lingo to provide message-based RPC with Spring beans. Although Spring's JMS invoker drew inspiration from Lingo and may be considered its replacement, it only offers synchronous communication. Meanwhile Lingo offers something that Spring's JMS invoker doesn't: the ability to invoke remote methods asynchronously.

Now that we've seen how Spring simplifies JMS, let's look at how Spring works with a similarly named Java standard. In the next chapter, we'll explore Spring's ability to export beans as managed beans using JMX.