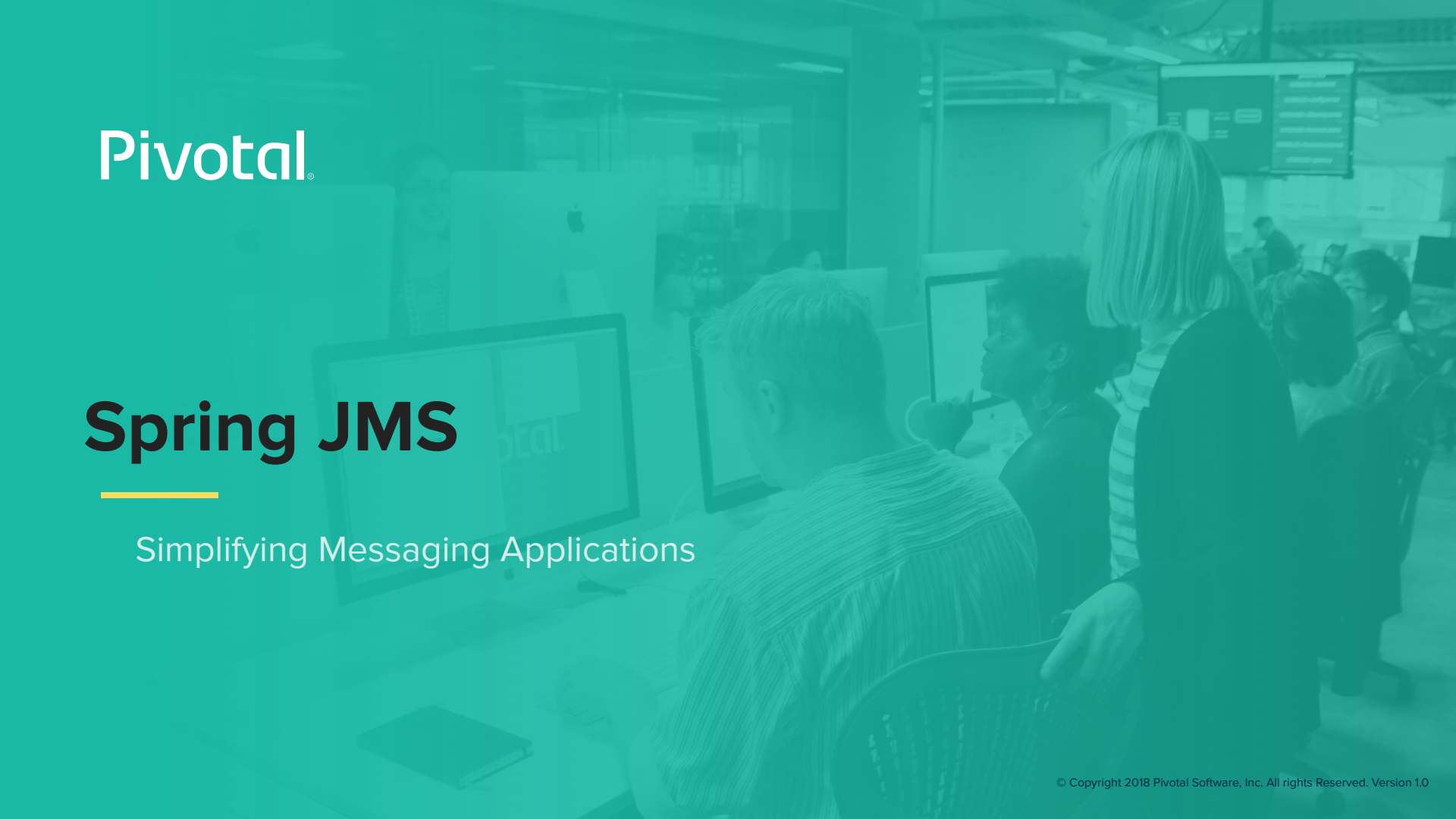


Pivotal

Spring JMS

Simplifying Messaging Applications

Objectives

After completing this lesson, you should be able to

- Explain the basic Concepts behind JMS
- Configure JMS Resources with Spring
- Use Spring's *JmsTemplate* to send & receive Messages
- Implement Asynchronous Messaging using a Listener Container

Agenda

- **Introduction to JMS**
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages

Java Message Service (JMS)

- The JMS API provides an abstraction for accessing Message Oriented Middleware
  - Avoid vendor lock-in
  - Increase portability of your code

- JMS does *not* enable different MOM vendors to communicate
  - Need a bridge (expensive)
  - Or use AMQP (standard msg protocol, like SMTP)
    - See RabbitMQ

JMS Core Components

- Message
- Destination
- Connection
- Session
- MessageProducer
- MessageConsumer

JMS Message Types

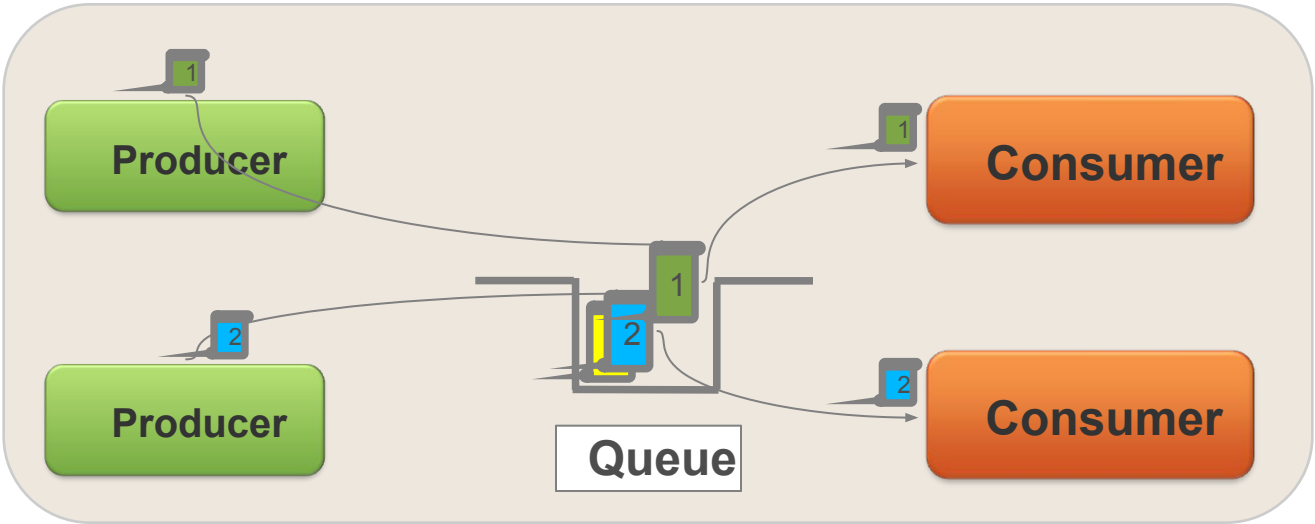
- Implementations of the *Message* interface
  - TextMessage
  - ObjectMessage
  - MapMessage
  - BytesMessage
  - StreamMessage

JMS Destination Types

- Implementations of the Destination interface
  - Queue
    - Point-to-point messaging
  - Topic
    - Publish/subscribe messaging
- Both support *multiple* producers and consumers
  - Messages are different
  - Let's take a closer look ...

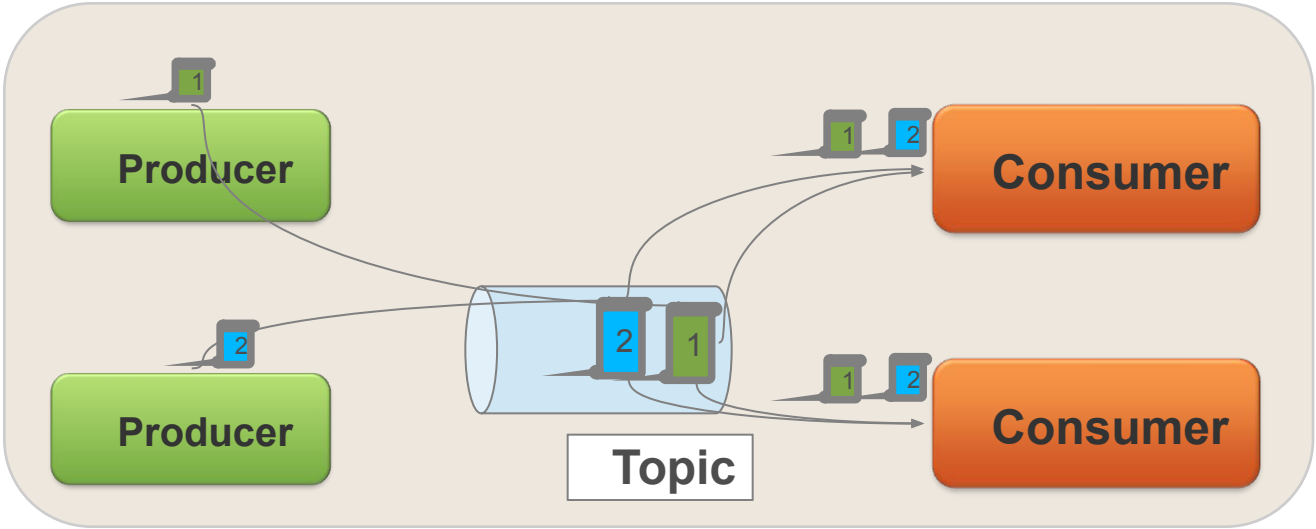
JMS Queues: Point-to-point

1. Message sent to queue
2. Message queued
3. Message consumed by *single* consumer



JMS Topics: Publish-subscribe

1. Message sent to topic
2. Message optionally stored
3. Message distributed to *all* subscribers



The JMS Connection

- A JMS Connection is obtained from a factory

```
Connection conn = connectionFactory.createConnection();
```

- Typical enterprise application:
  - *ConnectionFactory* is a managed resource bound to JNDI

```
Properties env = new Properties();
// provide JNDI environment properties
Context ctx = new InitialContext(env);
ConnectionFactory connectionFactory =
        (ConnectionFactory) ctx.lookup("connFactory");
```

The JMS Session

- A Session is created from the Connection
  - Represents a unit-of-work
  - Provides transactional capability

```
Session session = conn.createSession(
                boolean transacted, int acknowledgementMode);

// use session
if (everythingOkay) {
    session.commit();
} else {
    session.rollback();
}
```

Creating Messages

- The *Session* is responsible for the creation of various JMS Message types

```
session.createTextMessage("Some Message Content");

session.createObjectMessage(someSerializableObject);

MapMessage message = session.createMapMessage();
message.setInt("someKey", 123);

BytesMessage message = session.createBytesMessage();
message.writeBytes(someByteArray);
```

Producers and Consumers

- The *Session* is also responsible for creating instances of *MessageProducer* and *MessageConsumer*

```
producer = session.createProducer(someDestination);

consumer = session.createConsumer(someDestination);
```

Agenda

- Introduction to JMS
- **Apache ActiveMQ**
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages

JMS Providers

- Most providers of Message Oriented Middleware (MoM) support JMS
  - WebSphere MQ, Tibco EMS, Oracle EMS, JBoss AP, SwiftMQ, …
  - Some are Open Source, some commercial
  - Some are implemented in Java themselves

- *The lab for this module uses Apache ActiveMQ*

Apache ActiveMQ

- Open source message broker written in Java
- Supports JMS and many other APIs
  - Including non-Java clients!
- Can be used stand-alone in production environment
  - 'activemq' script in download starts with default config
- Can also be used *embedded* in an application
  - Configured through ActiveMQ or Spring configuration
  - *What we use in the labs*

Apache ActiveMQ Features

- Many cross language clients & transport protocols
  - Incl. excellent Spring integration
- Flexible & powerful deployment configuration
  - Clustering incl. load-balancing & failover, ...
- Advanced messaging features
  - Message groups, virtual & composite destinations, wildcards, etc.
- Enterprise Integration Patterns when combined with Spring Integration or Apache Camel
  - from the book by Gregor Hohpe & Bobby Woolf

Agenda

- Introduction to JMS
- Apache ActiveMQ
- **Configuring JMS Resources with Spring**
- Spring's JmsTemplate
- Sending Messages
- Receiving Messages

Configuring JMS Resources with Spring

- Spring enables decoupling of your application code from the underlying infrastructure
  - Container provides the resources
  - Application is simply coded against the API

- Provides deployment flexibility
  - use a standalone JMS provider
  - use an application server to manage JMS resources

See: Spring Framework Reference – Using Spring JMS
https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#remoting-jms

Configuring a ConnectionFactory

- *ConnectionFactory* may be standalone

```
@Bean
public ConnectionFactory connectionFactory() {
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    cf.setBrokerURL("tcp://localhost:60606");
    return cf;
}
```

- Or retrieved from JNDI

```
@Bean
public ConnectionFactory connectionFactory() throws Exception {
    Context ctx = new InitialContext();
    return (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");
}
```

```
<jee:jndi-lookup id="connectionFactory" jndi-name="jms/ConnectionFactory"/>
```

Configuring Destinations

- Destinations may be standalone

```
@Bean
public Destination orderQueue() {
    return new ActiveMQQueue( "order.queue" );
}
```

- Or retrieved from JNDI

```
@Bean
public Destination orderQueue() throws Exception {
    Context ctx = new InitialContext();
    return (Destination) ctx.lookup("jms/OrderQueue");
}
```

```
<jee:jndi-lookup id="orderQueue" jndi-name="jms/OrderQueue"/>
```

Agenda

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- **Spring's JmsTemplate**
- Sending Messages
- Receiving Messages

Spring's JmsTemplate

- The template simplifies usage of the API
  - Reduces boilerplate code
  - Manages resources transparently
  - Converts checked exceptions to runtime equivalents
  - Provides convenience methods and callbacks

NOTE: The *AmqpTemplate* (used with RabbitMQ) has an almost identical API to the *JmsTemplate* – they offer similar abstractions over very different products

Exception Handling

- Exceptions in JMS are checked by default
- *JmsTemplate* converts checked exceptions to runtime equivalents

JmsTemplate configuration

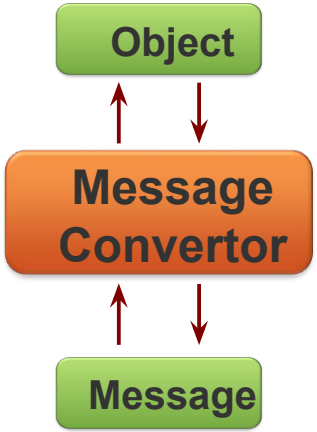
- *Must* provide reference to *ConnectionFactory*
  - via either constructor or setter injection
- Optionally provide other facilities
  - *setMessageConverter* (1)
  - *setDestinationResolver* (2)
  - *setDefaultDestination* or *setDefaultDestinationName* (3)

```
@Bean
public JmsTemplate jmsTemplate () {
    JmsTemplate template = new JmsTemplate( connectionFactory() );
    template.setMessageConverter ( … );
    template.setDestinationResolver ( … );
    return template;
}
```

(1), (2), (3) – see next few slides

(1) MessageConverter

- The *JmsTemplate* uses a MessageConverter to convert between objects and messages
  - You only send and receive objects
- The default *SimpleMessageConverter* handles basic types
  - String to TextMessage
  - Map to MapMessage
  - byte[] to BytesMessage
  - Serializable to ObjectMessage

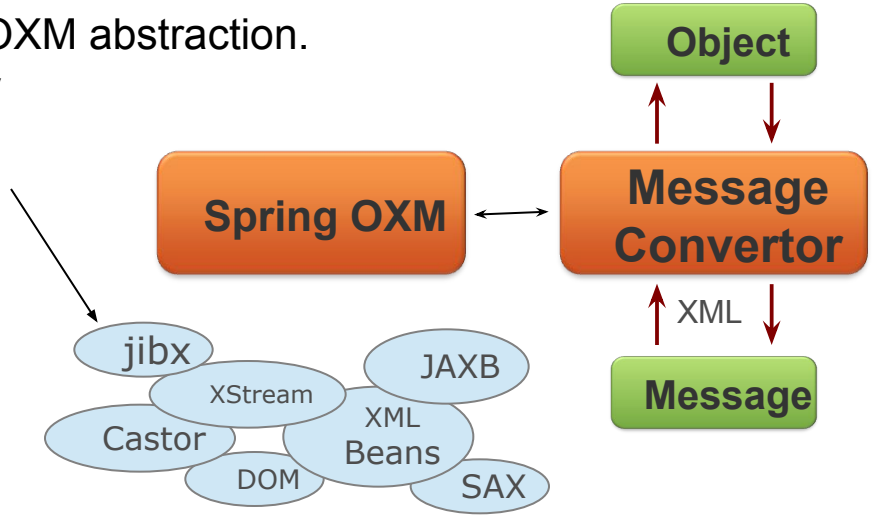


NOTE: It is possible to implement custom converters by implementing the *MessageConverter* interface

XML MessageConverter

- XML is a common message payload
  - ...but there is no “XmlMessage” in JMS
  - Use *TextMessage* instead

- MarshallingMessageConverter
  - Plugs into Spring's OXM abstraction.
  - You choose strategy



MarshallingMessageConverter Example

```
@Bean public JmsTemplate jmsTemplate () {
    JmsTemplate template = new JmsTemplate( connectionFactory() );
    template.setMessageConverter ( msgConverter() );
    return template;
}

@Bean public MessageConverter msgConverter() {
    MessageConverter converter = new MarshallingMessageConverter();
    converter.setMarshaller ( marshaller() );
    return converter;
}

@Bean  public Marshaller marshaller() {
    Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
    marshaller.setContextPath ( "example.app.schema" );
    return marshaller;
}
```

JAXB2 Illustrated here, other strategies available.

(2) DestinationResolver

- Convenient to use destination names at runtime
- DynamicDestinationResolver used by default
  - Resolves topic and queue names
  - *Not* their Spring bean names
- JndiDestinationResolver also available


```
Destination resolveDestinationName(Session session,
        String destinationName,
        boolean pubSubDomain)
throws JMSException;
```

publish-subscribe?
true → Topic
false → Queue

(3) Default Destination

- Used by default when sending *or* receiving messages

Specify by Object

```
@Bean
public JmsTemplate orderTemplate () {
      JmsTemplate template = new JmsTemplate ( connectionFactory() );
      template.setDefaultDestination ( orderQueue() );
      return template;
}
```

Specify by Name

```
@Bean public JmsTemplate orderTemplate () {
      JmsTemplate template = new JmsTemplate ( connectionFactory() );
      template.setDefaultDestinationName ("order.queue");
      return template;
}
```

Agenda

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- **Sending Messages**
- Receiving Messages

Sending Messages

- The template provides options
  - Simple methods to send a JMS message
  - One line methods that leverage the template's MessageConverter
  - Callback-accepting methods that reveal more of the JMS API
- Use the simplest option for the task at hand

Sending POJO

- A message can be sent in one single line

```
public class JmsOrderManager implements OrderManager {
  @Autowired JmsTemplate jmsTemplate;
  @Autowired Destination orderQueue;

  public void placeOrder(Order order) {
    String stringMessage = "New order " + order.getNumber();
    jmsTemplate.convertAndSend("message.queue", stringMessage );
                                  // use destination resolver and message converter

    jmsTemplate.convertAndSend(orderQueue, order);   // use message converter

    jmsTemplate.convertAndSend(order);   // use converter and default destination
  }
}
```

No @Qualifier so Destination is wired by *name*

Sending JMS Messages

- Useful when you need to access JMS API
  - eg. set expiration, redelivery mode, reply-to ...

```
public void sendMessage(final String msg) {

    this.jmsTemplate.send( (session) -> {
        TextMessage message = session.createTextMessage(msg);
        message.setJMSExpiration(2000);    // 2 seconds
        return message;
      });
}
```

Lambda syntax

```
public interface MessageCreator {
  public Message createMessage(Session session)
                  throws JMSException;
}
```

Agenda

- Introduction to JMS
- Apache ActiveMQ
- Configuring JMS Resources with Spring
- Spring's JmsTemplate
- Sending Messages
- **Receiving Messages**

Receiving Objects

- JmsTemplate can also *receive* data
  - Automatically converted using MessageConverter
  - Underlying messages hidden

```
public void receiveData() {

  // use message converter and destination resolver
  String s =  (String) jmsTemplate.receiveAndConvert("message.queue");
  // use message converter
  Order order1 = (Order) jmsTemplate.receiveAndConvert(orderQueue);
  // use message converter and default destination
  Order order2 = (Order) jmsTemplate.receiveAndConvert(\);
}
```

Receiving Messages

- Or you may access the underlying message
  - Gives you access to message properties

```
public void receiveMessages() {

  // handle JMS native message from default destination
  ObjectMessage orderMessage =  (ObjectMessage) jmsTemplate.receive();
  Order order2 = (Order) orderMessage.getObject();

  // receive(destination) and receive(destinationName) also available
}
```

Synchronous Message Exchange

- JmsTemplate also implements a request/reply pattern
  - Using sendAndReceive()
  - Sending a message and blocking until a reply has been received (also uses receiveTimeout)
  - Manage a temporary reply queue automatically by default

```
public void processMessage(String msg) {

  Message reply =  jmsTemplate.sendAndReceive("message.queue",
      (session) -> {
          return session.createTextMessage(msg);
      });
  // handle reply
}
```

Asynchronous or Synchronous

- Sending messages is asynchronous
  - The send methods return immediately
    - Even if the message takes time to be delivered
    - Recall the acknowledgement modes in createSession()
- But receive() and receiveAndConvert() are blocking
  - Synchronous – will wait for ever for a new message
    - optional timeout: setReceiveTimeout()
- How can we receive data asynchronously?
  - JMS defines *Message Driven Beans*
  - But you normally need a full JEE container to use them

Spring's MessageListener Containers

- Spring provides containers for asynchronous JMS reception
  - *SimpleMessageListenerContainer*
    - Uses plain JMS client API
    - Creates a fixed number of Sessions
  - *DefaultMessageListenerContainer*
    - Adds transactional capability
- Many configuration options available for each container type

Quick Start

Steps for Asynchronous Message Handling

(1) Define POJO / Bean to process Message
(2) Define JmsListenerContainerFactory / Enable Annotations
(3) Annotate POJO to be message-driven

Spring 4.1

Step (1)
Define POJO / Bean to Process Message

- Define a POJO to process message
  - Note: No references to JMS

```
public class OrderServiceImpl {
        @JmsListener(destination="queue.order")
        @SendTo("queue.confirmation")
        public OrderConfirmation order(Order o) { … }
}
```

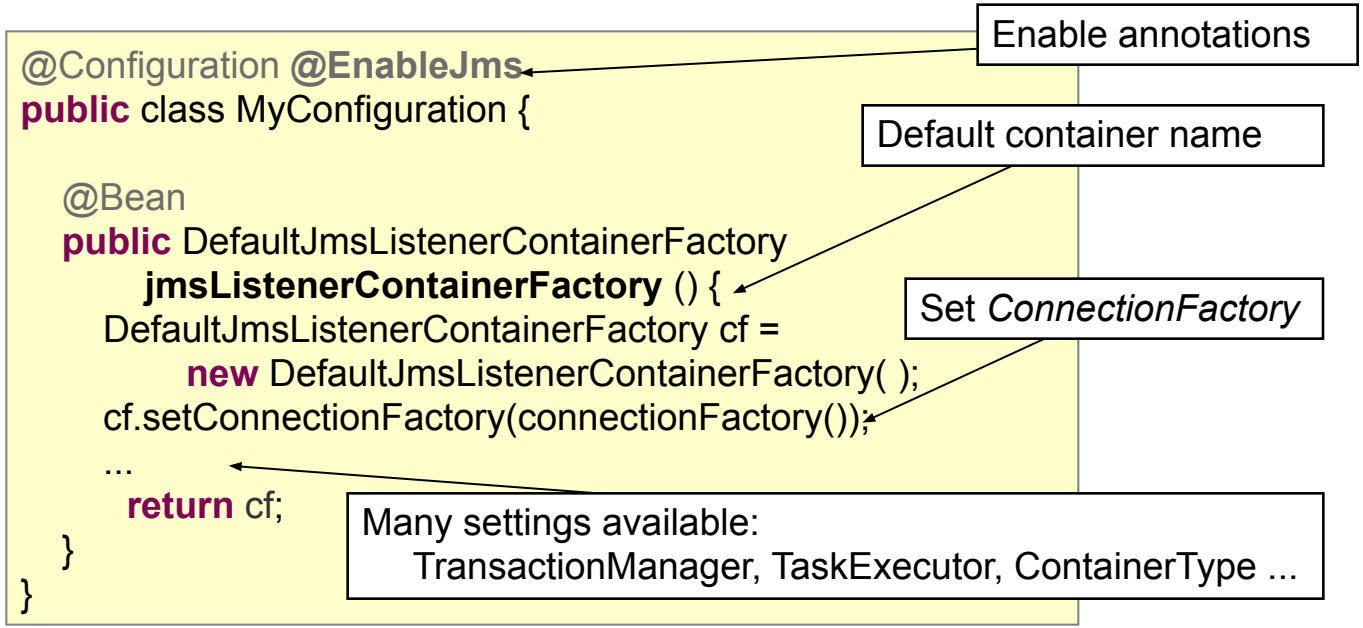
  - Define as a Spring bean using XML, JavaConfig, or annotations as preferred
  - @JmsListener enables a JMS message consumer for the method
  - @SendTo defines response destination (optional)

Step (2)

Define JmsListenerContainerFactory to use

Spring 4.1

- JmsListenerContainerFactory
  - Separates JMS API from your POJO:



Spring 4.1

Step (3)

Define Receiving Method with @JmsListener

- Container with name **jmsListenerContainerFactory** is used by default

```
public class OrderServiceImpl {
    @JmsListener(containerFactory="myFactory",
        destination="orderConfirmation")
    public void process(OrderConfirmation o) { … }
}
```

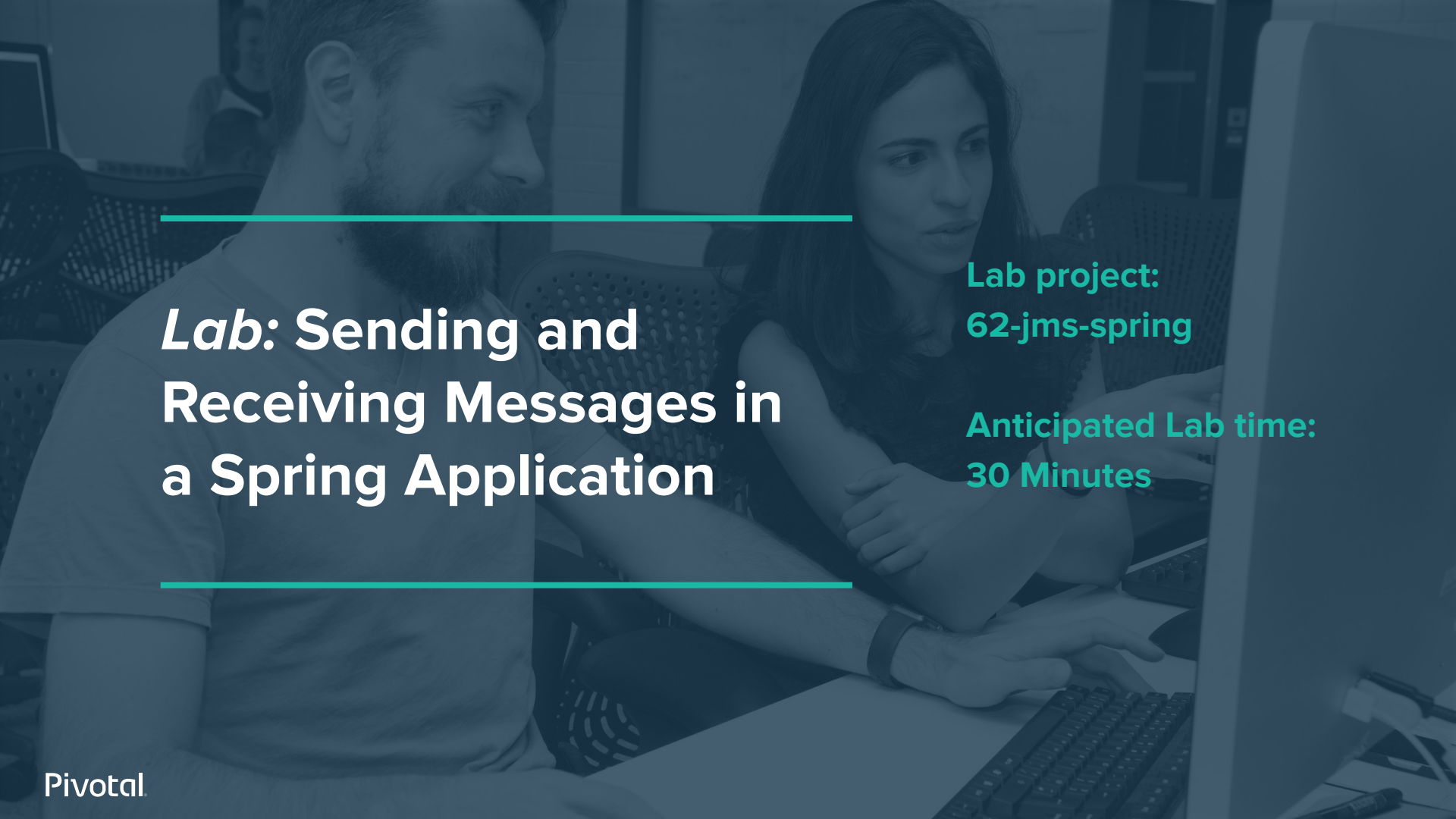
- Can also set a custom concurrency or a payload selector

```
public class OrderServiceImpl {
    @JmsListener(selector="type = 'Order'",
        concurrency="2-10", destination = "order")
    public OrderConfirmation order(Order o) { … }
}
```

Using JMS: Pros and Cons

- Advantages
  - Application freed from messaging concerns
    - Resilience, guaranteed delivery (compare to REST)
  - Asynchronous support built-in
  - Interoperable – languages, environments

- Disadvantages
  - Requires additional third-party software
    - Can be expensive to install and maintain
  - More complex to use – *but not with JmsTemplate!*



Lab: Sending and Receiving Messages in a Spring Application

Lab project:
62-jms-spring

Anticipated Lab time:
30 Minutes