Spring Framework(Part 1)
Appendices

Appendix A: HSQLDB

3

Notes on Using HSQL ...

- Stop the database using CTRL+C in the command window
- Start with a new, empty database:
  - Delete the contents of the data directory in the *HSQLDB* project after shutting down the server
- To bring up the *GUI DB* Manager
  - From within Eclipse
    - Expand the HSQLDB project and double-click ***RunDbMgr.bat***
  - From a command-prompt:
    - From within the <workspace>/HSQLDB/data directory, type:

```
java -classpath <path_to_jar>/hsqldb.jar
org.hsqldb.util.DatabaseManagerSwing
```

5

Notes:

When connecting to the DB using the GUI Database Mgr, select the type: "HSQL Database Engine Server". By default the user is SA and the password is left blank.

Appendix B: Web Tier Integration and JSF

Spring JSF Support

- Spring and JSF have similar bean models
- Spring provides a VariableResolver class to help JSF beans access Spring beans
  - This results in a JSF container and a Spring IOC Container managing their own respective beans

9

Notes:

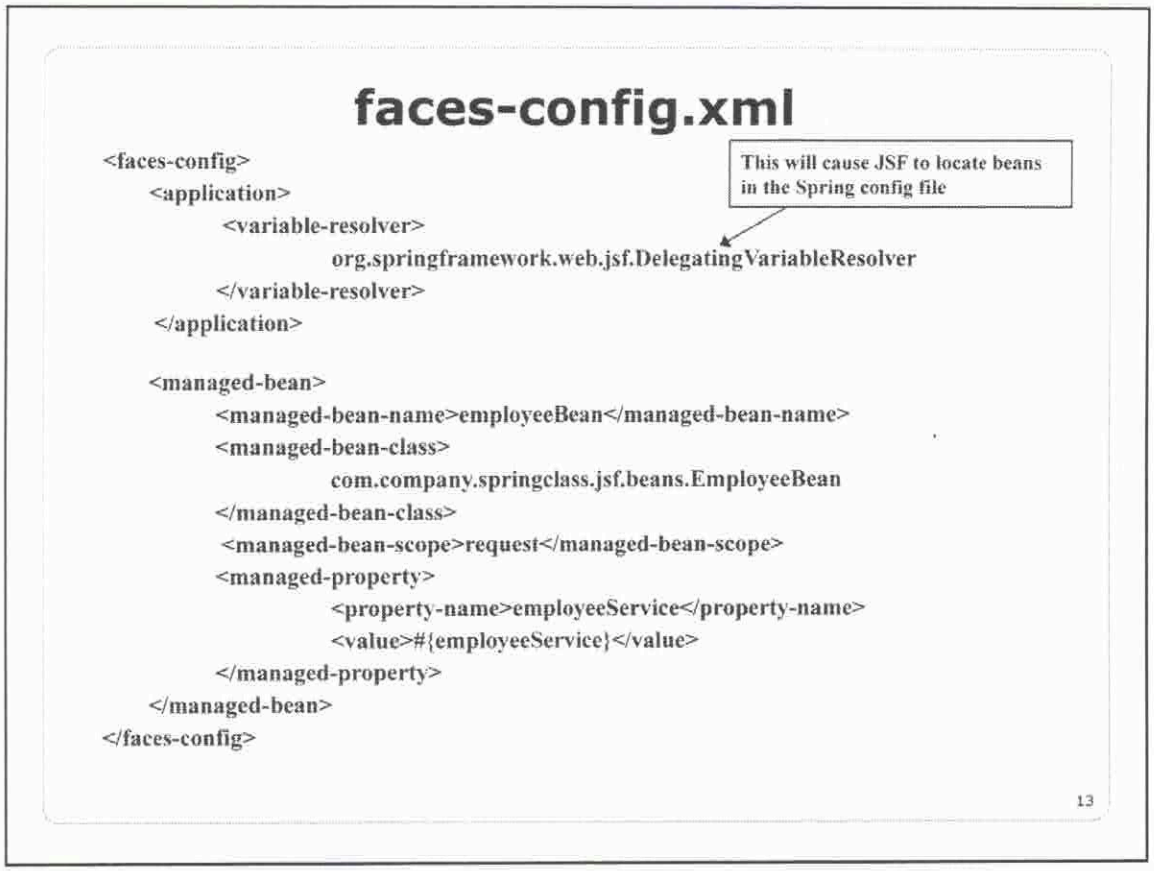
Setting Up web.xml

```
<web-app...>
    <listner>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>Faces Servlet</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
        <welcome-file>page1.faces</welcome-file>
    </welcome-file-list>
</web-app>
```

11

Notes:

As discussed earlier, integrate Spring by using a listener. The same is true for JSF apps. Set up the faces servlet as you normally would.



Notes:

With the <variable-resolve> declaration, JSF will search its own config file for the desired bean first, then it will delegate requests to search the Spring config file.

The JSPs

page1.jsp

```
<f:view>
    <h:form>
        <h:outputLabel for="id">Enter Employee ID: </h:outputLabel>
        <h:inputText id="id" value="#{employeeBean.id}" />
        <h:commandButton value="Find" action="#{employeeBean.find}"/>
    </h:form>
</f:view>
```

page2.jsp

```
<f:view>
    <h1>Employee Info</h1>
    <ul>
      <li> FirstName: <h:outputText
                value="#{employeeBean.employee.firstName}"/> </li>
      <li> LastName: <h:outputText
                value="#{employeeBean.employee.lastName}"/> </li>
      <li> Dept Id: <h:outputText
                value="#{employeeBean.employee.deptId}"/> </li>
    </ul>
</f:view>
```

15

Notes:

The JSPs for our solution contain the input form and output pages respectively. Our employee data was stored as a reference in the EmployeeBean class, so it is easy to retrieve.

JSF to Spring Injection

- The previous two examples inject Spring beans into a JSF web app
  - This results in two containers being used
- A cleaner approach is to inject JSF beans into Spring
  - This results in only a single (Spring) container managing beans

17

Notes:

web.xml

```
<web-app …>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>

<listener>
    <listener-class>
        org.springframework.web.context.request.RequestContextListener
    </listener-class>
</listener>

… (same servlet and mapping for JSF as before) …

</web-app>
```

The ***RequestContextListener*** requires Spring 2.0 or later

19

Notes:

applicationContext.xml


```
<beans…>

    <bean id="employeeBean"
          class="com.company.springclass.jsf.beans.EmployeeBean"
          scope="request">
            <property name="employeeService" ref="employeeService"/>
    </bean>

    <bean id="employeeService"
          class="com.company.springclass.services.EmployeeServiceImpl">
        <property name="dataSource">
            <ref local="dataSource"/>
        </property>
    </bean>
</beans>
```

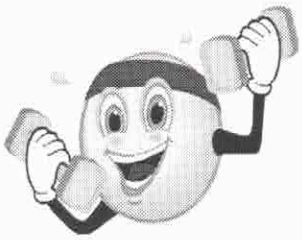
21

Notes:

The dataSource bean definition was omitted for brevity.

Exercise 6

- Complete the ***JSF-Spring*** integration exercise detailed in *SpringLab06*
- Follow the additional instructions in *SpringLab06*



23

Notes:

Appendix C: Spring Remoting and Distributed Services

Remote Objects

- Historically, creating remotely accessed objects is a complex process
- Solutions require stubs and skeletons to be generated
- Client communicates with a stub
  - Client "appears" to be working with remote object

27

Notes:

Spring Remoting

- Spring Simplifies the remoting tasks by handling much of the "communications coding"
- Our example uses the DayOfWeek service
- DayOfWeek will be exposed by Spring
  - Spring will require some simple configuration

29

Notes:

The Host Server

```
public class DayOfWeekHost {

    public static void main (String[] args) throws Exception {

        new FileSystemXmlApplicationContext (
                                        "applicationContext.xml");
        System.out.println("Server Started...");
    }
}
```

The server only needs to instantiate the needed services

Because the RMIServiceExporter will be started, the application remains alive

Running this Java class starts the RMI application on the server side

31

Notes:

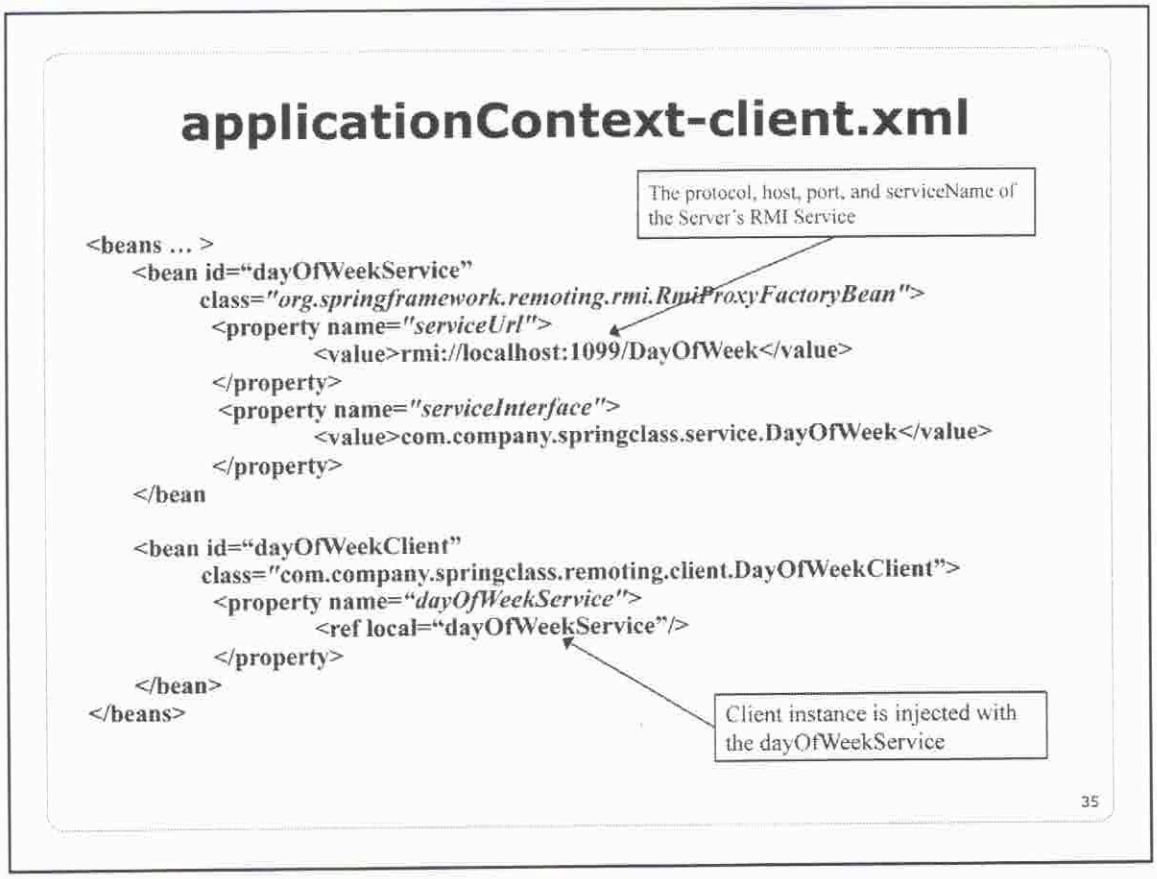
A remoting application has two parts: the server and the client. These are separate (hence the name remote) applications running. The server is shown here. This code merely instantiates an IOC container. The IOC container then instantiates the needed services as defined in the config file.

The RMI Client

- The client application is independent of the server
  - It will have its own Spring configuration and IOC container
  - The client must be given the service interface (often termed the remote interface)
  - This is the DayOfWeek class in our example

33

Notes:

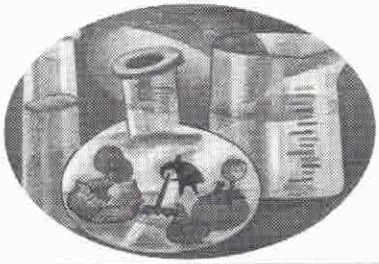


Notes:

The RmiProxyFactorybean has the task of making a remote object "appear" to be local to a client. It does this by generating a proxy component that implements the same interface as the service.

Lab 7: Spring Remoting

- Complete the Employee Service application following the additional instructions in **SpringLab07a** and **SpringLab07b**
- ***SpringLab07a*** *is a Host Application*
- ***SpringLab07b*** *is a Client Project*



Start with Lab 07a...

37

Notes:

Appendix D: Spring and Web 2.0

What is DWR?

- Direct Web Remoting (DWR)
  - A framework for making Java calls from JavaScript
    - Passing objects back and forth from JavaScript to a Java-based App Server
    - Automatic JavaScript code generation
    - Makes the browser seem like a remote client to the server

41

Notes:

DWR is a Client-Server solution in which a JavaScript code can invoke Java functions very easily without creating a lot of 'hookup' code. In fact, DWR will manage and create much of the needed communications code for you.

Setting Up DWR

- Down the JAR

http://directwebremoting.org/

- Drop *dwr.jar* (and commons-logging.jar) into the WEB_INF/lib directory

43

Notes:

To begin using DWR in an application, simply download the JAR from the provided web site and place it in the **lib** directory of your web application, along with Jakarta commons logging JAR.

Setting Up DWR ...

- Set up the meta-data file *dwr.xml*:

```
<!DOCTYPE dwr PUBLIC
        "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
        "http://getahead.org/dwr/dwr20.dtd">
<dwr>
 <allow>
  <create creator="new" javascript="HelloWorld">
   <param name="class"
              value="com.company.springclass.dwr.beans.HelloWorld"/>
  </create>
 </allow>
</dwr>
```

45

Notes:

Create the dwr.xml metadata file that the DWR servlet reads. This file is placed in the WEB-INF directory and should be called dwr.xml. This file will tell DWR what calls to allow on certain classes as well as what JavaScript code to create. That's it for the setup.

Create the Java Class

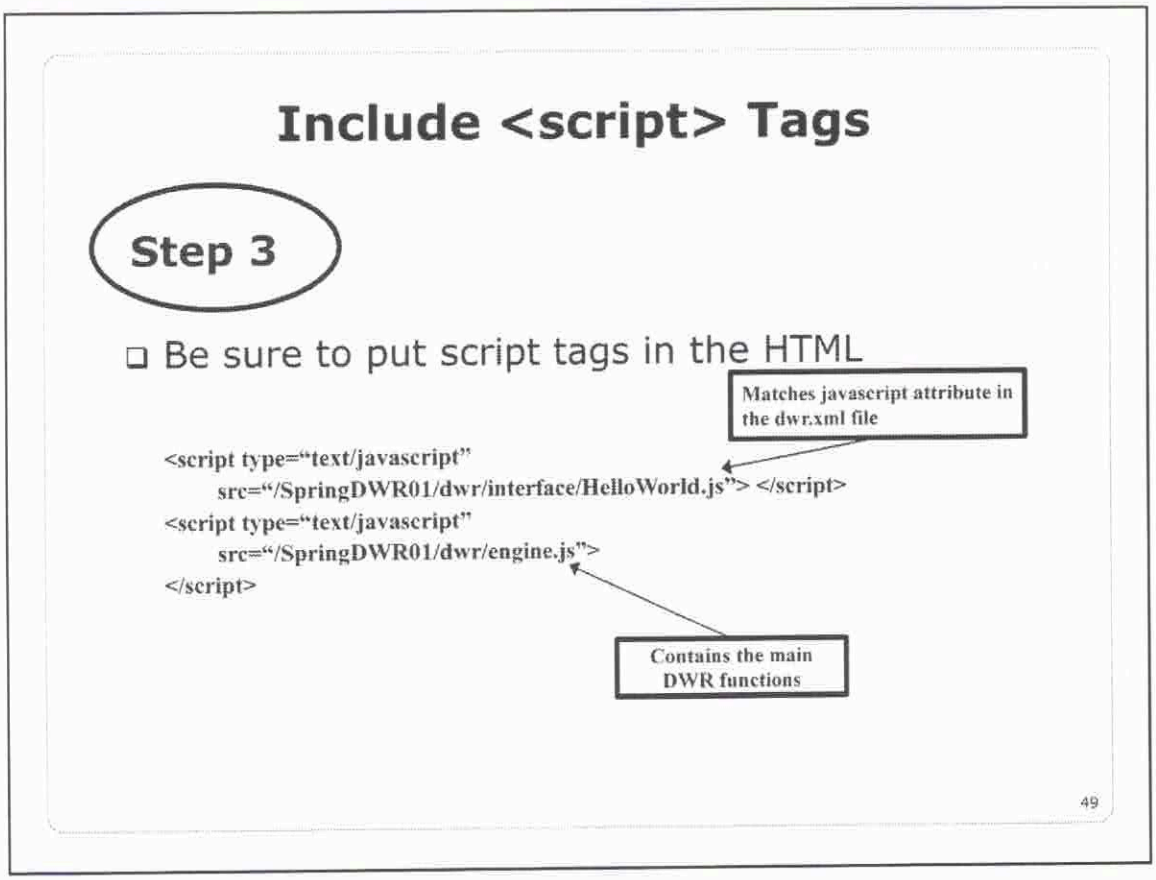
Step 1

```
package com.company.springclass.dwr.beans;

public class HelloWorld {

        public String greeting(String name) {
                if (name == null) name = "friend";

                return "Hello, " + name;
        }
}
```

47

Notes:

This will be our class that is invoked on the server-side. We will invoke it from our JavaScript code. The method *greeting()* accepts a value and returns a value back to the client.



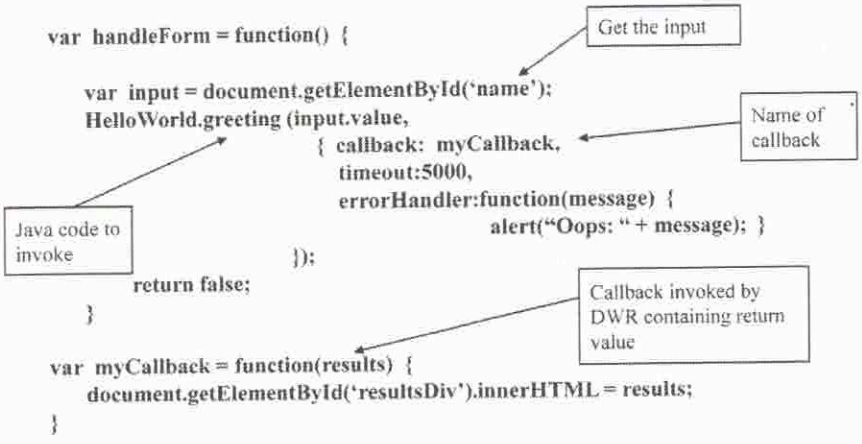
Notes:

You do not create HelloWorld.js. This file is generated by DWR based on your value in the dwr.xml file shown on the previous slide. This file will be referenced from the /<APPNAME>/dwr/interface directory structure as shown.

The JavaScript (the Client)

Step 5

- The JavaScript invokes the DWR JavaScript engine (which in turn invokes the servlet)



51

Notes:

handleForm() is called when the form is submitted. The input value is retrieved and passed to the Java method by writing JavaScript code. The JavaScript code actually invokes the DWR JavaScript engine, which in turn talks to the DWR servlet. The response is sent back to the page and your callback function (in this case, called myCallback) is invoked.

web.xml

```
<servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>
        org.directwebremoting.spring.DwrSpringServlet
    </servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <param-name>allowsScriptTagRemoting</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

A Spring-aware DWR Servlet

53

Notes:

applicationContext.xml ...

```
<bean id="employeeService"
class="com.company.springclass.services.EmployeeServiceImpl">
    <property name="dataSource">
        <ref local="dataSource"/>
    </property>
    <dwr:remote javascript="EmployeeService" />
</bean>

<dwr:configuration>
    <dwr:convert type="bean"
        class="com.company.springclass.beans.Employee" />
</dwr:configuration>

</beans>
```

Allow DWR to create JavaScript for this service

Define how to use the data returned from the service, in this case as properties from an Employee object

55

Notes:

Summary

- DWR can make remote calls to a Spring-based service easy
- Ajax-based Spring service solutions bring powerful new Web 2.0 capabilities to web applications
- When a service returns a business object, a converter configuration needs to be set up

57

Notes:

Appendix E: Managing Applications Using JMX

59

JMX

- JMX not JMS
- Java Management eXtension
- Mature Standard (JSR-3)
- Provides Managed Bean (MBean) Server
  - Bean registration / discovery / query
  - Exposes beans information
- Built-in Event / Notification Model
- Built-in Monitoring / Timer Services
- Part of Java 5

61

Notes:

JMX Managed Beans (MBeans)

- Standardized API for Instrumentation
- Exposes Properties (MBean Attributes)
- Exposes Operations
- Emit Event Notifications
- Hosted within MBean Server
- Accessible via Management Tools
  - Query MBean Server for attribute values
  - Access MBean operations
  - Register for Mbean event notifications

63

Notes:

Spring and JMX

- Spring makes JMX development easier
- MBean Exporter Component
  - Generates/registers dynamic MBean component
- Exposes regular POJO as managed objects
  - No special interfaces or naming patterns required
- Declarative MBean Registration
- Flexible Management API
  - Java annotations / Common Attributes
  - Expose via interfaces, method names, POJO's

65

Notes:

Summary

- Spring, Spring AOP, and JMX (included in Java 5) are potent combinations for creating highly manageable applications.
- The Spring Framework provides a set of API that makes it easy to develop manageable applications using a declarative XML configuration file to expose POJO as managed beans.
- JMX and related tools can be used to provide runtime application monitoring, control, and configuration services.
- Most common case is application monitoring. Use Spring AOP to cut through your application execution sequence to collect data and push that data onto dedicated managed beans.
- Use dedicated POJOs to expose management information and controls. This allows you to separate manageability from the core business logic of your application.

67

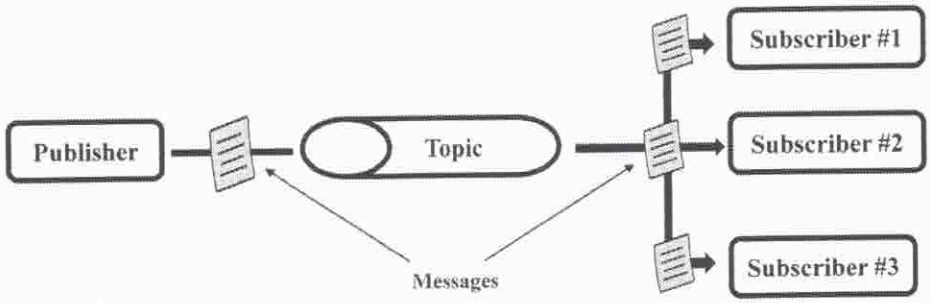
Notes:

Appendix F: Java Message Service

69

Topics

- Publish / Subscribe
  - All receivers receive a copy of the message



71

Notes:

Steps

- Set-up a Message broker → *ActiveMQ*
- Add ***activemq.jar*** to the classpath
- Run **activemq.bat**
- Create a connection factory

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

73

Notes:

Working with JMS

- JMS code is similar to JDBC
- JMS Template
  - Connection
  - Obtain a session
  - Send/receive messages
  - JMSException handling
  - Configuration

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

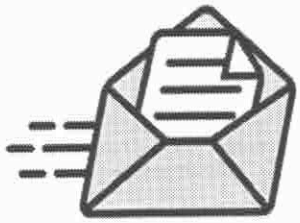
75

Notes:

Sending Messages ...

- The Code

```
package com.roadrantz.marketing;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.Session;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import com.roadrantz.domain.Mortorist;

public class RantzMarketingGatewayImpl
    implements RantzMarketingGateway {
  public RantzMarketingGatewayImpl () {}
```



77

Notes:

Sending Messages ...

- Additional Wiring

```
<bean id="marketingGateway"
      class="com.roadrantz.marketing.RantzMarketingGatewayImpl">
    <property name="jmsTemplate" ref="jmsTemplate" />
    <property name="destination" ref="rantzDestination" />
</bean>
```

79

Notes:

Consuming a Message ...

- The Code

```
package com.roadrantz.marketing;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.JmsUtils;

public class MarketingReceiverGatewayImpl  {
  public MarketingReceiverGatewayImpl () {}

  public SpammedMotorist receiveSpammedMotorist ()  {
    MapMessage message = (MapMessage)  jmsTemplate.receive();   <-- Receives message

    SpammedMotorist motorist = new SpammedMotorist();
    try  {
      motorist.setFirstName (message.getString("firstName"));
       motorist.setLastName (message.getString("lastName"));
      motorist.setEmail (message.getString ("email"));
```

Creates object from message

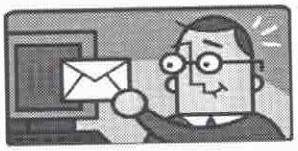
81

Notes:

Consuming a Message ...

- Configuration

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



83

Notes: