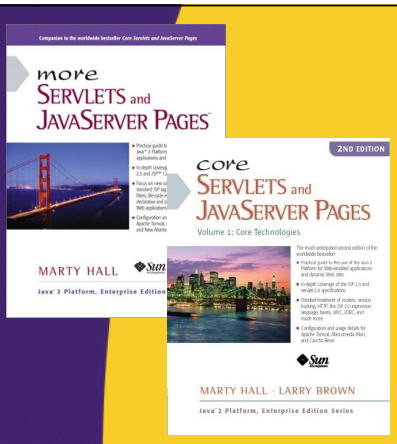
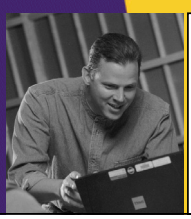
Using Spring in Web Applications

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/spring.html

© 2009 Marty Hall





Agenda

- **Servlet/JSP apps**
  - Creating a Dynamic Web Project in Eclipse
  - Adding Spring support
  - Adding Spring JAR files and bean definition file
  - Registering listeners in web.xml
  - Loading bean definition file
  - Getting bean instances
- **JSF apps**
  - Creating a JSF Project in Eclipse
  - Adding Spring support
  - Defining beans in applicationContext.xml
  - Defining beans in faces-config.xml

Overview

Why are Web Apps Different?

- **You need to access the bean definition file from many different places**
  - With desktop Java apps, you can have a single piece of code that instantiates the container and gets beans
    - I.e., driver class that calls instantiates ClassPathXmlApplicationContext and calls getBean
  - With Web apps, each servlet wants access to beans
    - But you want to instantiate container once only
- **You need additional bean scopes**
  - Standard Spring supports singleton and prototype
  - Web apps also want request, session, and application
- **Note**
  - We are not discussing the SpringMVC framework here, but rather how to use regular Spring beans in Web apps

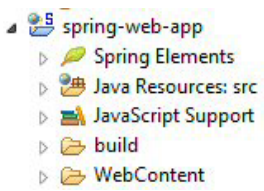
Summary of New Approaches

- **Regular Web Apps**
  - Put the two JAR files in WEB-INF/lib
  - Put bean definitions in WEB-INB/applicationContext.xml
    - request and session scope now supported
  - Declare two listeners in web.xml
    - Container will be instantiated when app is loaded
  - Get container reference by using static method in WebApplicationContextUtils
  - Get beans normally after that
- **JSF Apps**
  - Same approach for JAR files, bean defn file, and listeners
  - Declare variable-resolver in faces-config.xml
  - Can declare beans in applicationContext or faces-config

Using Spring in Regular Java-Based Web Apps

Creating the App in Eclipse

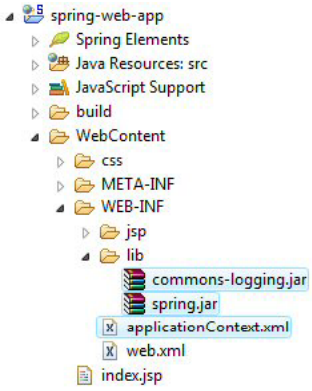
- **Create Dynamic Web App**
  - File → New → Project → Web → Dynamic Web Project
    - If you have made a Dynamic Web Project recently, you can just do File → New → Dynamic Web Project
- **Add Spring support**
  - R-click project, Spring Tools → Add Spring Project Nature



- **Note**
  - This tutorial assumes that you already know how to configure Eclipse for Tomcat (or another server) and are already familiar with servlets, JSP, and MVC
    - If not, see tutorials on Tomcat/Eclipse setup, servlets, JSP, and MVC at http://www.coreservlets.com/

Configuring the App for Spring: The Three Standard Files

- **Spring JAR files**
  - Put spring.jar and commons-logging.jar in WebContent/WEB-INF/lib
- **Starting-point applicationContext.xml**
  - Put "empty" file (with header and <beans..></beans> only) in WEB-INF
    - Unlike in desktop apps, the filename matters. The standard loading utility assumes that name and location
- **Non-Eclipse users**
  - The structure of the Web app is not tied to Eclipse. You still put JAR files in WEB-INF/lib and put applicationContext.xml in WEB-INF



Original Bean Definition File

- **Empty/Starting-point file**
  - /WEB-INF/applicationContext.xml
    - If you want to change this default name/location, set a context param called contextConfigLocation to override it

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

</beans>
```

Configuring the App for Spring: Defining Listeners in web.xml

- **ContextLoaderListener**
  - This listener runs when the app is first started. It instantiates the ApplicationContext (from WEB-INF/applicationContext.xml) and places a reference to it in the ServletContext
  - You can retrieve this reference with the static getRequiredWebApplicationContext method of WebApplicationContextUtils
- **RequestContextListener**
  - This listener is needed if you declare any of your beans to be request-scoped or session-scoped
    - I.e., Web scopes instead of the usual Spring scopes of singleton or prototype

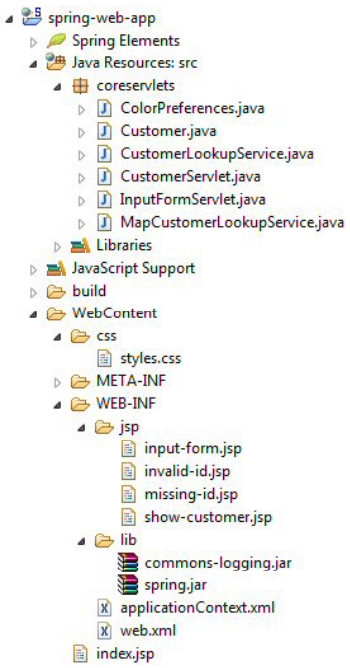
Defining Listeners: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
  …
</web-app>
```

Example Web App: Bank Balance Lookup

Overview

- **Input form**
  - Collects required user ID
    - Required
  - Collects preferred foreground and background colors
    - Optional
- **Results pages**
  - Shows name and balance of customer with given ID
  - Error if ID missing or unknown
  - Uses preferred colors



Overview (Continued)

- **Service interface**
  - CustomerLookupService
    - Maps customer IDs to Customers
- **Service implementation**
  - MapCustomerLookupService
    - Uses fixed HashMap of a few sample customers
- **applicationContext.xml**
  - Defines preferred foreground and background colors
    - In session scope
  - Defines customer lookup service as a Map
    - In singleton scope

web.xml: Defining Listeners

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

Loads /WEB-INF/applicationContext.xml and puts reference to it in servlet context. Can be accessed with WebApplicationContextUtils.getRequiredWebApplicationContext

Lets you give request or session scopes to beans in applicationContext.xml. If you don't use these scopes, this listener is not required. But you should probably have this entry commented out in web.xml just in case you want those scopes later.

web.xml: Servlet Mappings

```
<servlet>
   <servlet-name>Input Form Servlet</servlet-name>
   <servlet-class>coreservlets.InputFormServlet</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>Input Form Servlet</servlet-name>
   <url-pattern>/input-form</url-pattern>
 </servlet-mapping>

 <servlet>
   <servlet-name>Customer Lookup Servlet</servlet-name>
   <servlet-class>coreservlets.CustomerServlet</servlet-class>
 </servlet>
 <servlet-mapping>
   <servlet-name>Customer Lookup Servlet</servlet-name>
   <url-pattern>/get-customer</url-pattern>
 </servlet-mapping>
```


Color Preferences Bean

```
public class ColorPreferences implements Serializable {
  private String foreground, background;

  public String getForeground() {
    return(foreground);
  }
  public void setForeground(String foreground) {
    if (!isEmpty(foreground)) {
      this.foreground = foreground;
    }
  }

  // getBackground and setBackground

  private boolean isEmpty(String value) {
    return((value == null) || (value.trim().equals("")));
  }
}
```

In Web apps in general, session data should be Serializable. This is partly to support distributed apps, but the more important reason is that Tomcat and other servers will let session data live across server restarts if the data is Serializable.

Customer Lookup Service: Interface

```
public interface CustomerLookupService {

  public Customer getCustomer(String id);

  public Customer getRichestCustomer();
}
```


Customer Lookup Service: One Concrete Implementation

```
public class MapCustomerLookupService
       implements CustomerLookupService {
  private Map<String,Customer> sampleCustomers;

  public Map<String,Customer> getSampleCustomers() {
    return sampleCustomers;
  }
  public void setSampleCustomers(Map<String,Customer> sampleCustomers) {
    this.sampleCustomers = sampleCustomers;
  }

  public Customer getCustomer(String id) {
    if (id == null) {
      id = "unknown";
    }
    return(sampleCustomers.get(id.toLowerCase()));
  }

  public Customer getRichestCustomer() { … }
}
```

This will be set via <property name="sampleCustomers"> in applicationContext.xml

Customer Bean

```
public class Customer {
  private String customerID, firstName, lastName;
  private double balance;

  // Simple getters and setters

  public String getFormattedBalance() {
    return(String.format("$%,.2f", getBalance()));
  }
}
```


applicationContext.xml: Defining Color Preferences

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

  <bean id="colorPreferences" class="coreservlets.ColorPreferences"
        scope="session">
    <property name="foreground" value="black"/>
    <property name="background" value="#fdf5e6"/>
  </bean>
```

applicationContext.xml: Defining Lookup Service

```
<bean id="sampleLookupService"
      class="coreservlets.MapCustomerLookupService">
   <property name="sampleCustomers">
     <map>
       <entry key="a1234">
         <bean class="coreservlets.Customer">
           <property name="customerID" value="a1234"/>
           <property name="firstName" value="Rod"/>
           <property name="lastName" value="Johnson"/>
           <property name="balance" value="123.45"/>
         </bean>
       </entry>
       …
     </map>
   </property>
 </bean>

</beans>
```


index.jsp

```
<% response.sendRedirect("input-form"); %>
```

Input Form (Servlet)

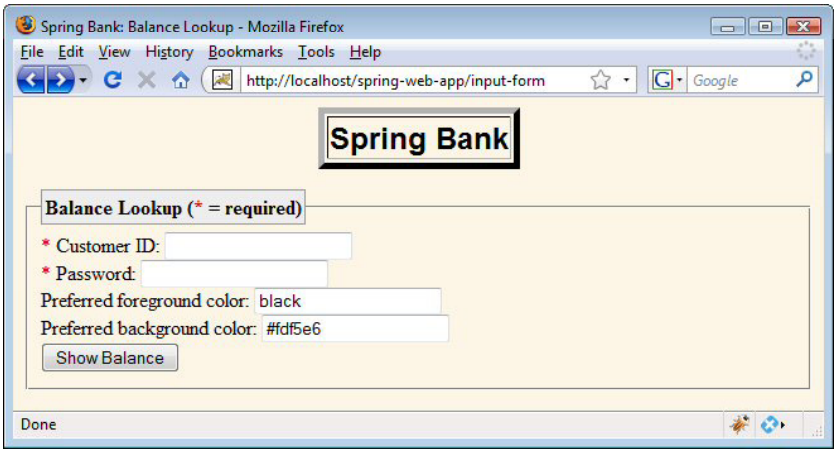
```
public class InputFormServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
    ApplicationContext context =
      WebApplicationContextUtils.getRequiredWebApplicationContext(
                                    (getServletContext()));
    context.getBean("colorPreferences");
    String address = "/WEB-INF/jsp/input-form.jsp";
    RequestDispatcher dispatcher =
      request.getRequestDispatcher(address);
    dispatcher.forward(request, response);
  }
}
```

Since bean is already specified with session scope, there is no need to do session.setAttribute here. But it is still necessary to call getBean.

Input Form (JSP)

```
<body bgcolor="${colorPreferences.background}"
      text="${colorPreferences.foreground}">
 …
<form action="get-customer" method="post">
   …
   <input type="text" name="cust-id"/>
   …
   <input type="password" name="cust-password"/>
   …
   <input type="text" name="fg"
          value="${colorPreferences.foreground}"/>
   …
   <input type="text" name="bg"
          value="${colorPreferences.background}"/>
   …
   <input type="submit" value="Show Balance"/>
</form>
```

Input Form (Result)



Customer Lookup (Servlet)

```
public class CustomerServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
      throws ServletException, IOException {
    ApplicationContext context =
      WebApplicationContextUtils.getRequiredWebApplicationContext(
                                    (getServletContext()));
    CustomerLookupService lookupService =
      (CustomerLookupService)context.getBean("sampleLookupService");
```

Customer Lookup (Servlet, Continued)

```
String id = request.getParameter("cust-id");
String address;
if (isEmpty(id)) {
  address = "missing-id.jsp";
} else {
  Customer customer = lookupService.getCustomer(id);
  if (customer == null) {
    request.setAttribute("id", id);
    address = "invalid-id.jsp";
  } else {
    request.setAttribute("customer", customer);
    address = "show-customer.jsp";
  }
}
address = "/WEB-INF/jsp/" + address;
```


Customer Lookup (Servlet, Continued)

```
  ColorPreferences colorPreferences =
    (ColorPreferences)context.getBean("colorPreferences");
  colorPreferences.setForeground(request.getParameter("fg"));
  colorPreferences.setBackground(request.getParameter("bg"));
  RequestDispatcher dispatcher =
    request.getRequestDispatcher(address);
  dispatcher.forward(request, response);
}

private boolean isEmpty(String value) {
  return((value == null) || (value.trim().equals("")));
}
}
```

Customer Lookup (/WEB-INF/show-customer.jsp)

```
…
<body bgcolor="${colorPreferences.background}"
      text="${colorPreferences.foreground}">
<table border="5" align="center">
  <tr><th class="title">Spring Bank: Your Balance</th></tr>
</table>
<p/>
<ul>
  <li>ID: ${customer.customerID}</li>
  <li>First name: ${customer.firstName}</li>
  <li>Last name: ${customer.lastName}</li>
  <li>Balance: ${customer.formattedBalance}</li>
</ul>
…
```

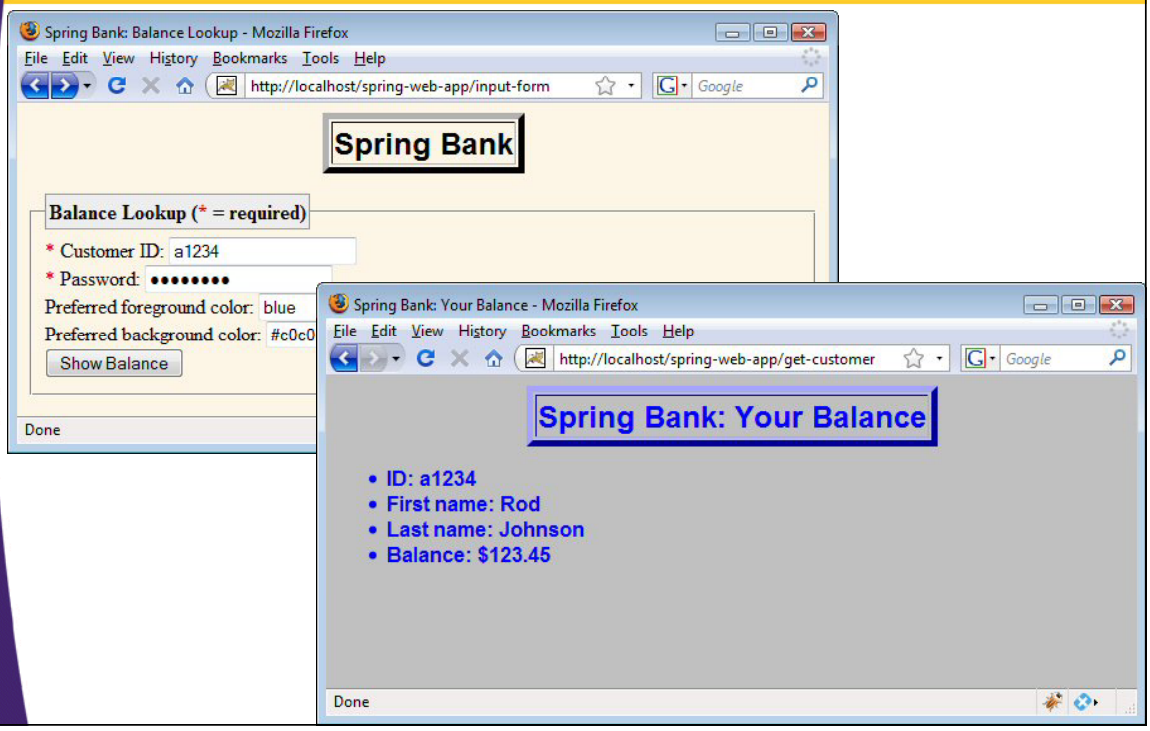

Customer Lookup (/WEB-INF/invalid-id.jsp)

```
…
<body bgcolor="${colorPreferences.background}"
      text="${colorPreferences.foreground}">
<table border="5" align="center">
  <tr><th class="title">Spring Bank: Error</th></tr>
</table>
<p/>
<h1 class="error">No customer with ID '${id}'.</h1>
</body></html>
…
```

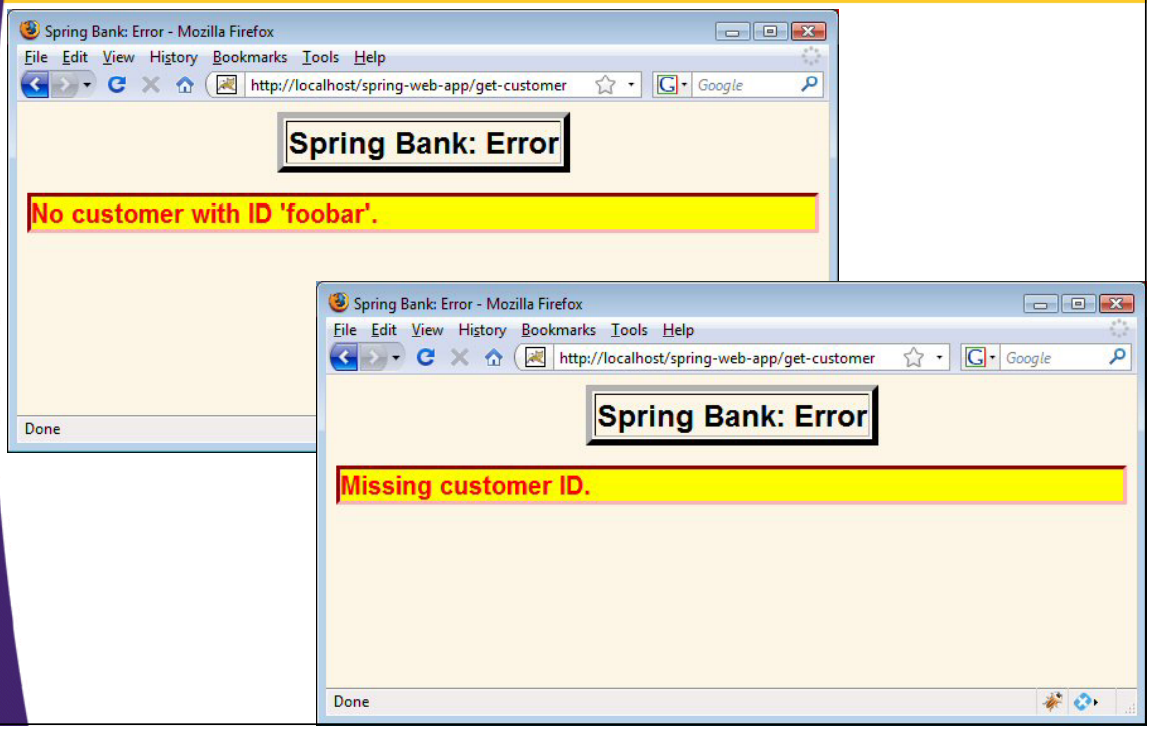
Customer Lookup (/WEB-INF/missing-id.jsp)

```
…
<body bgcolor="${colorPreferences.background}"
      text="${colorPreferences.foreground}">
<table border="5" align="center">
  <tr><th class="title">Spring Bank: Error</th></tr>
</table>
<p/>
<h1 class="error">Missing customer ID.</h1>
</body></html>
…
```


Customer Lookup (Results: Good Data)



Customer Lookup (Results: Bad Data)



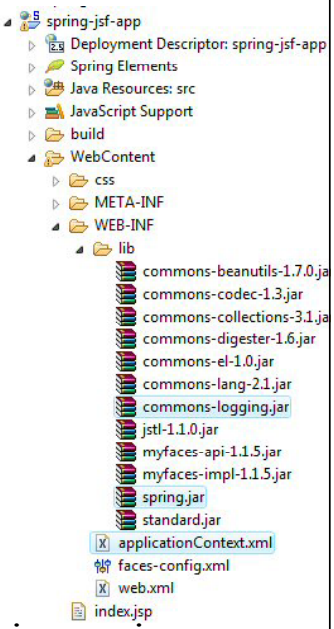
Using Spring in JSF Apps

Creating the App in Eclipse

- **Create Dynamic Web App**
  - File → New → Project → Web → Dynamic Web Project
- **Add Spring support**
  - R-click, Spring Tools → Add Spring Project Nature
- **Add JSF Support**
  - R-click project, Properties, Project Facets, JSF
    - You can also choose JSF under the configuration tab when first creating dynamic Web project
    - You can use JAR files from Eclipse or get your own (e.g., from myfaces.apache.org). I get my own.
  - Note
    - This section assumes that you already know JSF
      - If not, see http://www.coreservlets.com/JSF-Tutorial/

Configuring the App for Spring and JSF: The Standard Files

- **Spring JAR files**
  - Put spring.jar and commons-logging.jar in WebContent/WEB-INF/lib
- **Starting-point applicationContext.xml**
  - Put "empty" file (with header and <beans..></beans> only) in WEB-INF
- **Starting-point faces-config.xml**
  - Empty file with start/end tags only. Eclipse creates this automatically.
- **Note**
  - First two bullets are exactly the same as in previous section on using Spring in regular Web apps



Configuring the App for Spring and JSF: web.xml Settings

- **Two Spring listeners**
  - ContextLoaderListener and RequestContextListener
    - Same as in previous section on regular Web apps
- **Standard JSF settings**
  - At very least, FacesServlet mapped to some url-pattern like *.faces

web.xml Settings

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

Configuring JSF to Recognize Spring Beans: Idea

- **Not good to call getBean**
  - It would be technically legal to get the ApplicationContext and call getBean explicitly (probably from the backing bean's action controller method). But this is a bad idea since JSF is geared around declaring beans in config files only.
- **JSF already supports dependency injection**
  - The managed-property element lets you insert other beans inside newly created ones.
    - The only trick is to be able to refer to Spring beans
- **Use DelegatingVariableResolver**
  - Declare in faces-config.xml. Now, whenever JSF sees a bean name, it uses JSF rules first, then Spring rules next.

Configuring JSF to Recognize Spring Beans: faces-config.xml

```
…
<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
…
</faces-config>
```

JSF/Spring Example 1: Beans in Two Config Files

Overview

- **Functionality and appearance**
  - Exactly the same as in previous app
    - Given customer id, shows name and balance (or error)
- **Approach**
  - applicationContext.xml
    - Exactly the same as in previous example
      - Defines session-scoped ColorPreferences and singleton-scoped CustomerLookupService
  - faces-config.xml
    - Similar approach to standard JSF apps
      - Defines backing bean
      - Defines navigation rules
    - Access Spring beans with managed-property
      - Backing bean gets Spring beans as properties

Files not Shown

- **Java classes unchanged from last example**
  - ColorPreferences
    - Bean with foreground, background
  - Customer
    - Bean with customerID, firstName, lastName, balance
  - CustomerLookupService
    - Interface with getCustomer method
  - MapCustomerLookupService
    - Implementation with HashMap of some sample customers
- **web.xml**
  - Unchanged from version shown in previous section on general JSF configuration
    - Two Spring listeners, servlet mapping for FacesServlet

applicationContext.xml

- **Unchanged from previous example**
  - Defines session-scoped ColorPreferences

```
<bean id="colorPreferences" class="coreservlets.ColorPreferences"
      scope="session">
  <property name="foreground" value="black"/>
  <property name="background" value="#fdf5e6"/>
</bean>
```

  - Defines singleton-scoped CustomerLookupService

```
<bean id="sampleLookupService"
      class="coreservlets.MapCustomerLookupService">
  <property name="sampleCustomers">
    <map>…</map>
  </property>
</bean>
```

Backing Bean: Properties

```
public class CustomerBackingBean {
  private String inputID, password;
  private Customer customer;
  private ColorPreferences colorPreferences;
  private CustomerLookupService lookupService;


  // Getters and setters
  // for above 5 properties
```

Corresponding setters called by JSF when form submitted.

Filled in by action controller method (shown on next slide).

Setter methods called when bean created because of managed-bean-property in faces-config.xml. The incoming values are Spring beans.

Backing Bean: Action Controller Method

```
public String getBalance() {
   if (isEmpty(inputID)) {
     return("missing-id");
   } else {
     customer = lookupService.getCustomer(inputID);
     if (customer == null) {
       return("invalid-id");
     } else {
       return("show-balance");
     }
   }
}
```

faces-config: Variable Resolver

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE …>

<faces-config>
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>
```


faces-config: Backing Bean

```
<managed-bean>
   <managed-bean-name>formBean</managed-bean-name>
   <managed-bean-class>
     coreservlets.CustomerBackingBean
   </managed-bean-class>
   <managed-bean-scope>request</managed-bean-scope>
   <managed-property>
     <property-name>lookupService</property-name>
     <value>#{sampleLookupService}</value>
   </managed-property>
   <managed-property>
     <property-name>colorPreferences</property-name>
     <value>#{colorPreferences}</value>
   </managed-property>
 </managed-bean>
```

Gets the Spring bean called sampleLookupService and passes it to the setLookupService method of the JSF backing bean called formBean (i.e., injects it into the lookupService property).

Gets the Spring bean called colorPreferences and injects it into the colorPreferences property of the backing bean.

faces-config: Navigation Rules

```
<navigation-rule>
   <from-view-id>/welcome.jsp</from-view-id>
   <navigation-case>
     <from-outcome>show-balance</from-outcome>
     <to-view-id>/show-customer.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
     <from-outcome>missing-id</from-outcome>
     <to-view-id>/missing-id.jsp</to-view-id>
   </navigation-case>
   <navigation-case>
     <from-outcome>invalid-id</from-outcome>
     <to-view-id>/invalid-id.jsp</to-view-id>
   </navigation-case>
 </navigation-rule>
```


index.jsp

```
<% response.sendRedirect("welcome.faces"); %>
```

Input Form (welcome.jsp)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view> …
<body bgcolor="<h:outputText
                  value="#{formBean.colorPreferences.background}"/>"
      text="<h:outputText
               value="#{formBean.colorPreferences.foreground}"/>">
…
 <h:form>
    …
    <h:inputText value="#{formBean.inputID}"/>
    …
    <h:inputSecret value="#{formBean.password}"/>
    …
    <h:inputText value="#{formBean.colorPreferences.foreground}"/>
    …
    <h:inputText value="#{formBean.colorPreferences.background}"/>
    …
    <h:commandButton action="#{formBean.getBalance}"/>
  </h:form>
…
```


Main Results Page (show-customer.jsp)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view> …
<body bgcolor="<h:outputText
                  value="#{formBean.colorPreferences.background}"/>"
      text="<h:outputText
               value="#{formBean.colorPreferences.foreground}"/>">
…
<ul>
  <li>ID: <h:outputText value="#{formBean.customer.customerID}"/>
  </li>
  <li>First name:
      <h:outputText value="#{formBean.customer.firstName}"/>
  </li>
  <li>Last name:
      <h:outputText value="#{formBean.customer.lastName}"/>
  </li>
  <li>Balance:
      <h:outputText value="#{formBean.customer.formattedBalance}"/>
  </li>
</ul> …
```

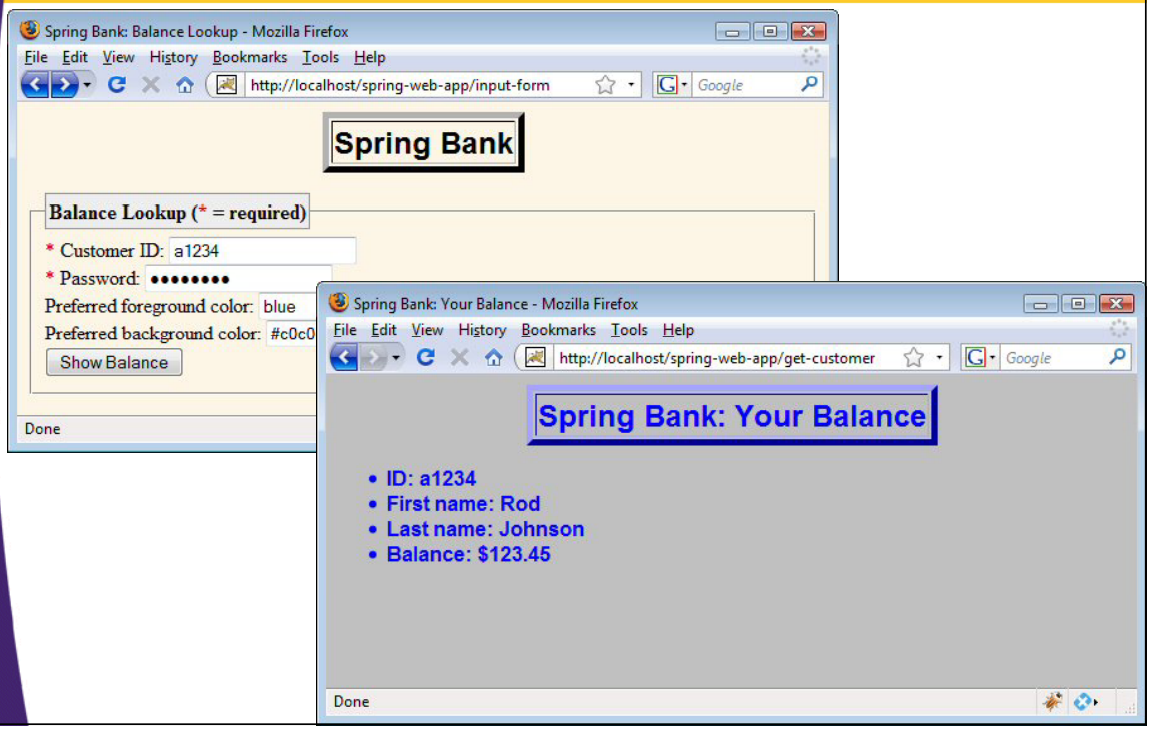
Error Page 1 (invalid-id.jsp)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view> …
<body bgcolor="<h:outputText
                  value="#{formBean.colorPreferences.background}"/>"
      text="<h:outputText
                value="#{formBean.colorPreferences.foreground}"/>">
…
<h1 class="error">
  No customer with ID
  '<h:outputText value="#{formBean.inputID}"/>'.
</h1>
…
```

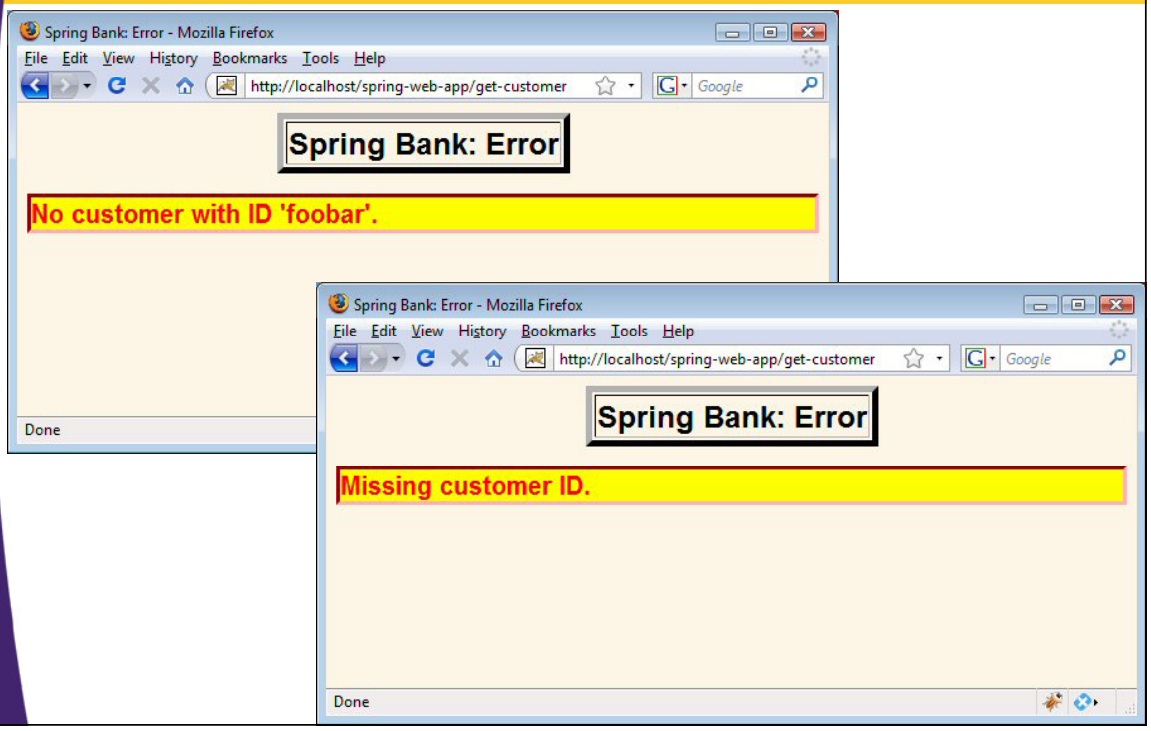

Error Page 1 (missing-id.jsp)

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view> …
<body bgcolor="<h:outputText
                  value="#{formBean.colorPreferences.background}"/>"
      text="<h:outputText
                value="#{formBean.colorPreferences.foreground}"/>">
…
<h1 class="error">Missing customer ID.</h1>
…
```

Results: Good Data (Same as Previous Example)



Results: Bad Data (Same as Previous Example)



JSF/Spring Example 2: Beans in One Config File

Overview

- **Issue with previous example**
  - Beans defined in two different files
    - Some using Spring syntax, some using JSF syntax
- **Approach**
  - applicationContext.xml
    - Defines the Spring beans as before
      - Defines session-scoped ColorPreferences and singleton-scoped CustomerLookupService
    - Also defines the backing bean
  - faces-config.xml
    - Defines navigation rules (and variable resolver) only
  - Functionality and appearance
    - Exactly the same as in previous two apps

Changes from Previous Example

- **faces-config.xml**
  - Deleted the entire <managed-bean> entry
- **applicationContext.xml**
  - Added the following simpler entry

```
<bean id="formBean"
      class="coreservlets.CustomerBackingBean"
      scope="request">
  <property name="lookupService" ref="sampleLookupService"/>
  <property name="colorPreferences" ref="colorPreferences"/>
</bean>
```

  - Two advantages
    - Spring dependency injection syntax is simpler and more powerful
    - All bean definitions in the same file

Wrap-up

Summary: Regular Web Apps

- **Two JAR files**
  - Put Spring JAR files in WEB-INF/lib
- **Bean definition file**
  - Put applicationContext.xml in WEB-INF
    - Scopes now include request and session in addition to singleton and prototype
- **Listeners**
  - Two listener definitions in web.xml
- **Getting beans**
  - Access ApplicationContext with static getRequiredWebApplicationContext method of WebApplicationContextUtils
  - Call getBean normally

Summary: JSF-Based Apps

- **Basic setup**
  - Start with same setup as regular Web apps
  - Use normal JSF definition of FacesServlet in web.xml
- **faces-config.xml**
  - Declare DelegatingVariableResolver
- **Option 1**
  - Declare Spring beans in applicationContext.xml
  - Declare backing beans in faces-config.xml
    - Refer to Spring beans with managed-bean-property
- **Option 2**
  - Declare all beans in applicationContext.xml
    - Refer to other beans with ref and normal Spring syntax

Questions?