Spring Web MVC Development

Contents

1. Overview of Spring Web
2. Spring Web MVC
3. Creating a Spring Web MVC application

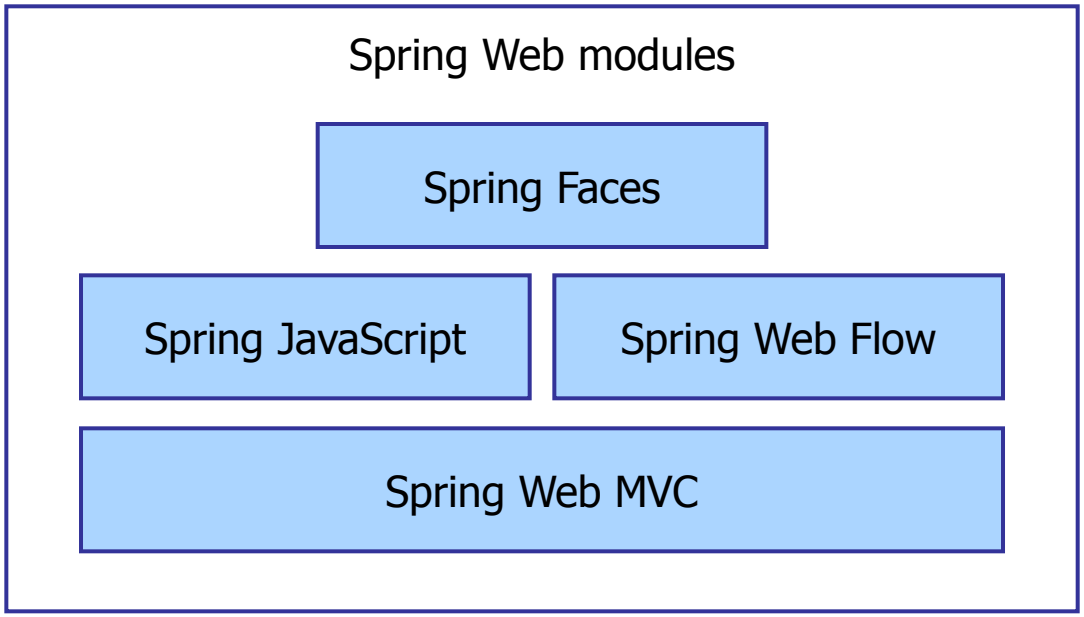
Demo project: DemoWebAppMvc

1. Overview of Spring Web

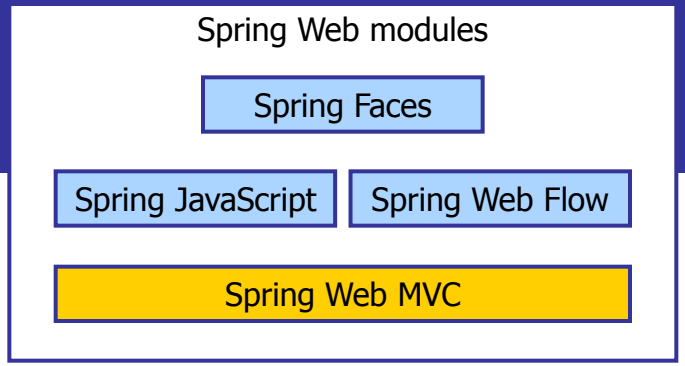
- Spring Web modules
- Spring MVC
- Spring JavaScript
- Spring Web Flow
- Spring Faces

Spring Web Modules

- Spring provides a rich model to simplify and decouple Web applications
- There are 4 parts to the Spring Web model
  - We've called them "Spring Web modules"

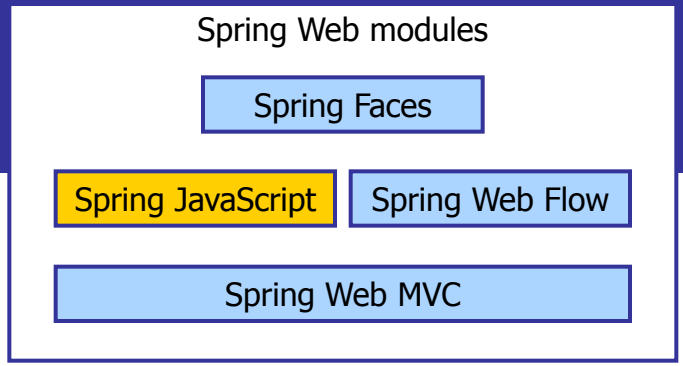


Spring Web MVC



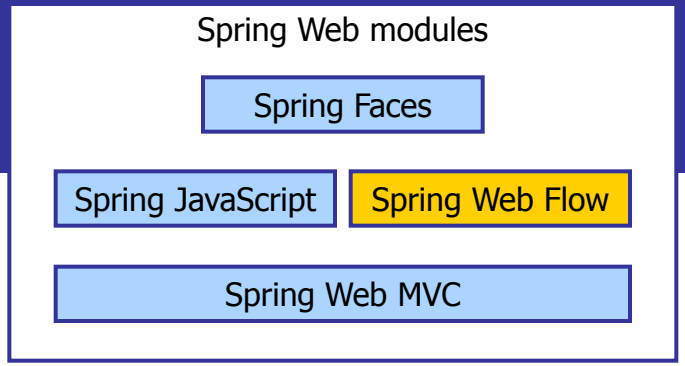
- Spring's Web framework
  - The core platform for developing Web apps with Spring
  - All higher-level modules build on it
- Core concepts:
  - Web requests are handled by controllers
  - They interact with business logic (the model)
  - They forward to views (usually JSPs)
- Supported view technologies
  - JSP / JSTL
  - Apache Velocity
  - Adobe PDF
  - Microsoft Excel
  - XML / XSLT

Spring JavaScript



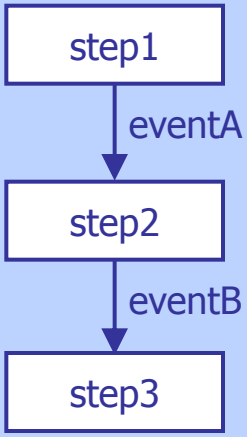
- JavaScript abstraction layer
  - Simplifies JavaScript usage
  - Avoid proliferation of JavaScript
- Features:
  - Decorate form input fields
  - Client-side validation
  - Ajax events
  - Fragment rendering
  - Ajax modal dialogs (pop-up)
  - CSS framework

Spring Web Flow

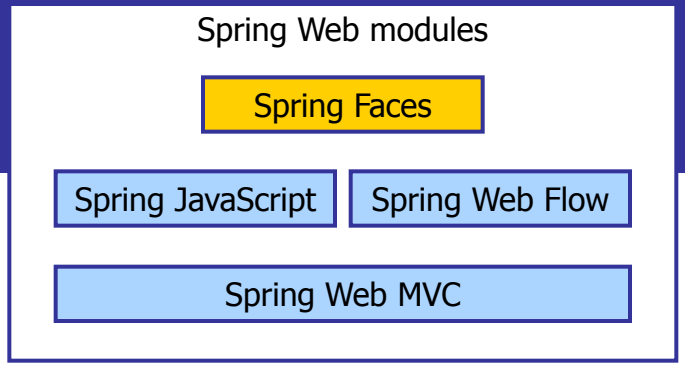


- Implements stateful flows
  - Plugs into Spring MVC as a controller technology
  - Allows you to define a flow to encapsulate a reusable sequence of steps
- Core concepts:
  - Flow definition language
  - For example:

```
<flow>

  <view-state id="step1">
    <transition on="eventA" to="step2"/>
  </view-state>

  <view-state id="step2">
    <transition on="eventB" to="step3"/>
  </view-state>

  <end-state id="step3"/>

</flow>
```



Spring Faces



- Combines JSF UI component model with Web Flow navigation/state
  - All in a native Spring MVC environment
- Features:
  - Lightweight JSF component library (~80% subset of JSF)
  - Includes Ajax support, client-side form validation
  - Built on Spring JavaScript
  - Compliant with JSF 1.1 and 1.2
- Major JSF component libraries are all usable in a Spring Faces environment
  - JBoss RichFaces
  - Apache Trinidad
  - IceSoft IceFaces

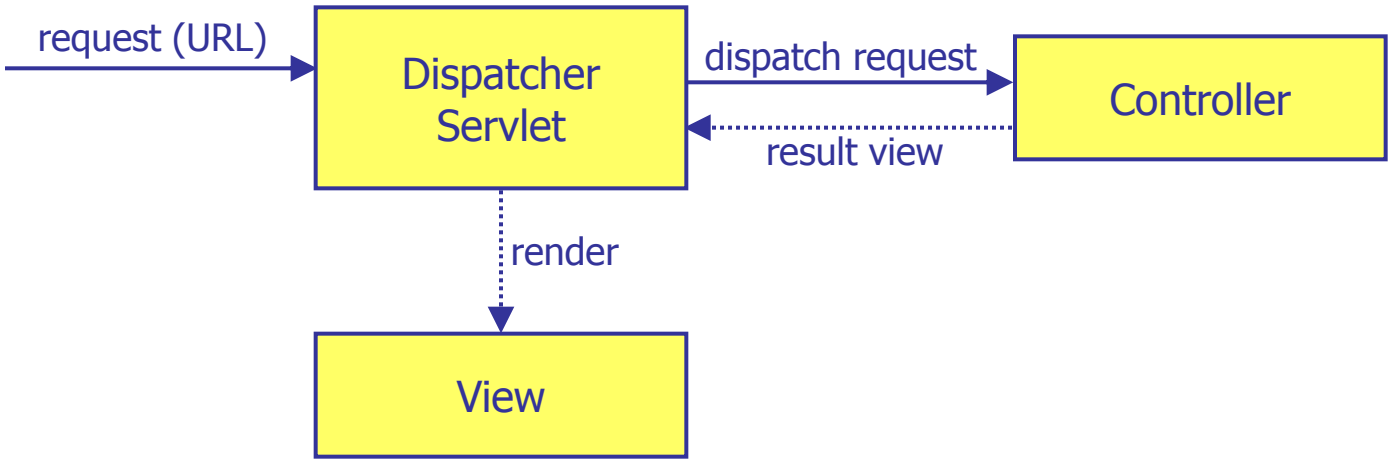
2. Spring Web MVC

- Web request handling overview
- DispatcherServlet
- Summarizing config files
- Controllers
- Views

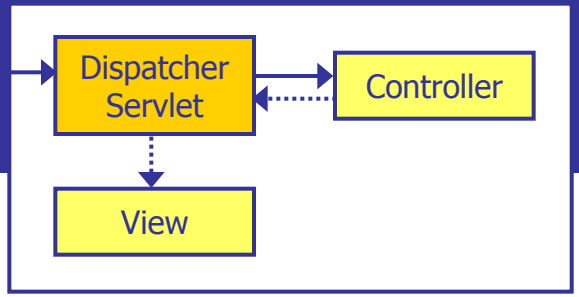
Request Processing Lifecycle

- Web request handling is quite simple
  - Based on an incoming URL…
  - … we need to call a method…
  - … after which the return value (if any)…
  - … needs to be rendered using a view

- Spring MVC request processing lifecycle:

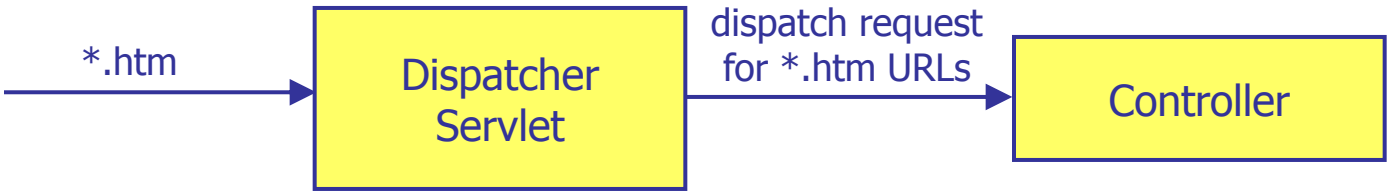


DispatcherServlet (1 of 2)

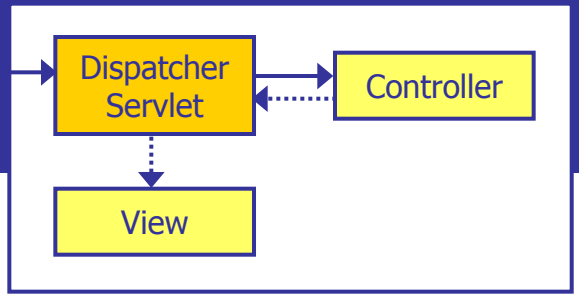


- `DispatcherServlet` is at the heart of Spring Web MVC
  - Front controller
  - Coordinates all request-handling activities
  - Analogous to Struts ActionServlet / JSF FacesServlet

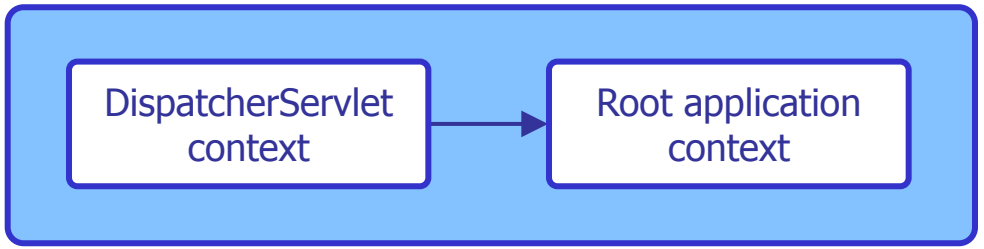
- Defined in `web.xml`
  - Servlet class: org.springframework.web.servlet.DispatcherServlet
  - You associate it with a URL pattern, e.g. *.htm



DispatcherServlet (2 of 2)

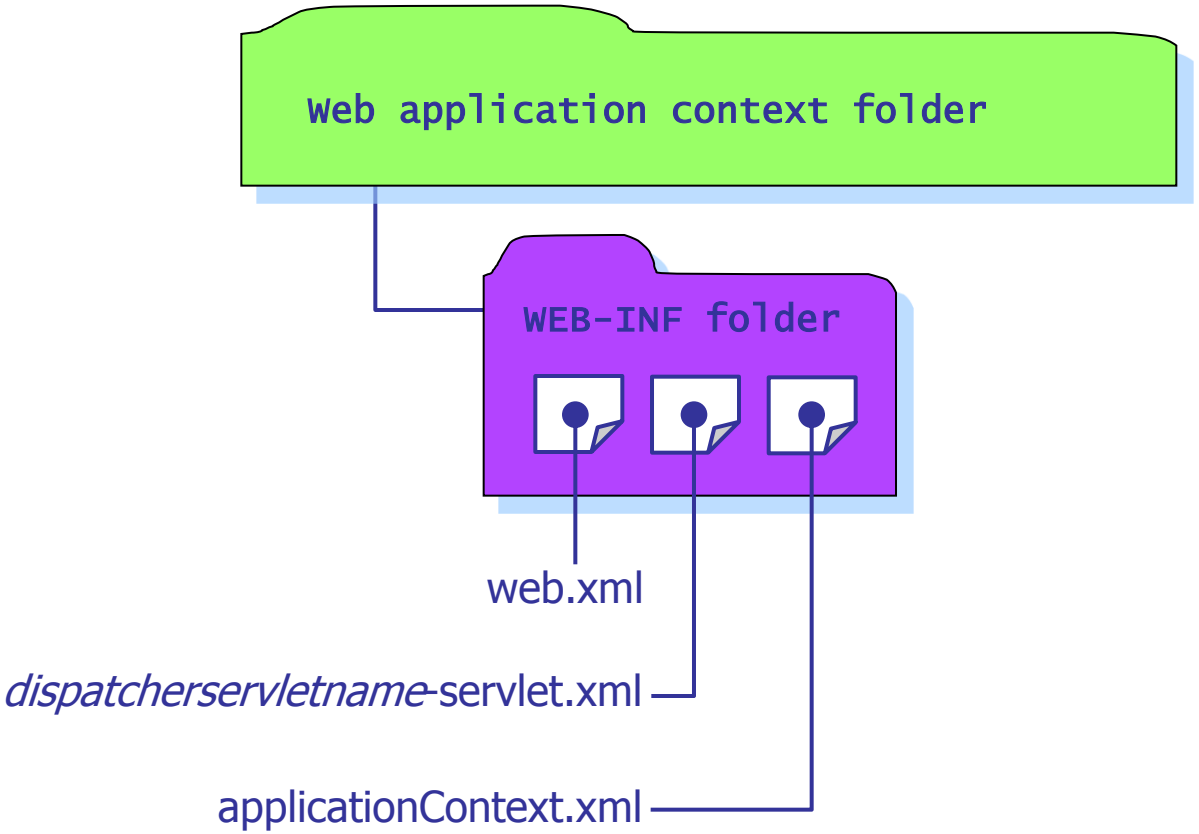


- `DispatcherServlet` has its own "servlet" application context
  - Filename: *`dispatcherservletname`*`-servlet.xml`
  - Defines beans needed by your `DispatcherServlet`
  - E.g. controllers, views, resolvers
- Still have access to the "root" application context
  - Filename: `applicationContext.xml`
  - Defines application-level beans
  - E.g. business services, repositories, etc.
- Servlet container after starting up:

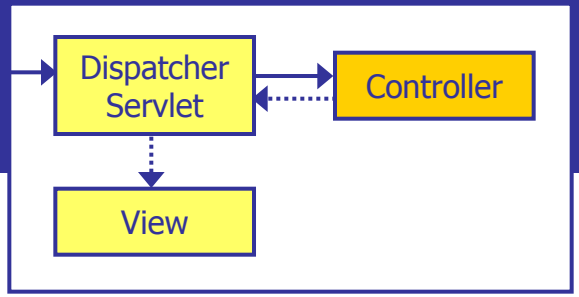


Summarizing Config Files

- As we've seen on the last few slides, a Spring Web MVC application will have (at least) 3 config files:



Controllers (1 of 4)

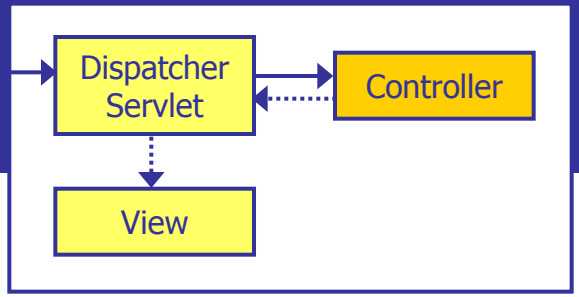


- Incoming requests are dispatched to controllers

- How to define a controller before Spring 2.5:
  - For each URL, define a class that implements `Controller`
  - Implement `handleRequest()`
  - Define URL→controller mapping rules in config file

- Current way to define a controller :
  - Define a bean annotated with `@Controller`
  - For each URL, define method annotated with `@RequestMapping`
  - (No need to define URL→controller mapping rules in config file)

Controllers (2 of 4)



- Traditional way to define a controller

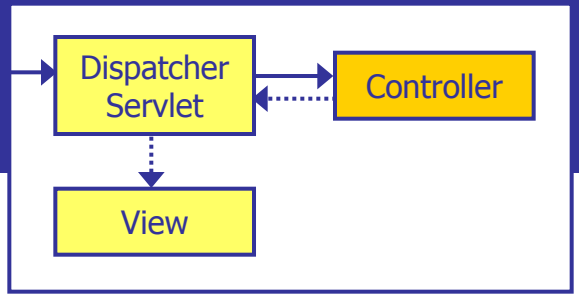
```
public class MyEmployeeListController implements Controller {
  public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) {…}
}
```

```
public class MyEmployeeDetailController implements Controller {
  public ModelAndView handleRequest(HttpServletRequest req, HttpServletResponse resp) {…}
}
```

```
<bean id="listController"   class="mypackage.MyEmployeeListController"  />
<bean id="detailController" class="mypackage.MyEmployeeDetailController"/>

<bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="urlMap">
    <map>
      <entry key="/employeeList.htm">
        <ref bean="listController" />
      </entry>
      <entry key="/employeeDetail.htm">
        <ref bean="detailController" />
      </entry>
    </map>
  </property>
</bean>
```

dispatcherservletname-servlet.xml

Controllers (3 of 4)



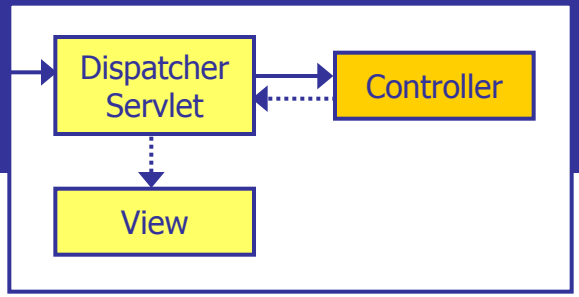
- Current way to define a controller

```
@Controller
public class MyController {

  @RequestMapping("/employeeList.htm")
  public ModelAndView listEmployees(HttpServletRequest req, HttpServletResponse resp) {…}

  @RequestMapping("/employeeDetail.htm")
  public ModelAndView detailForEmployee(HttpServletRequest req, HttpServletResponse resp) {…}
  …
}
```

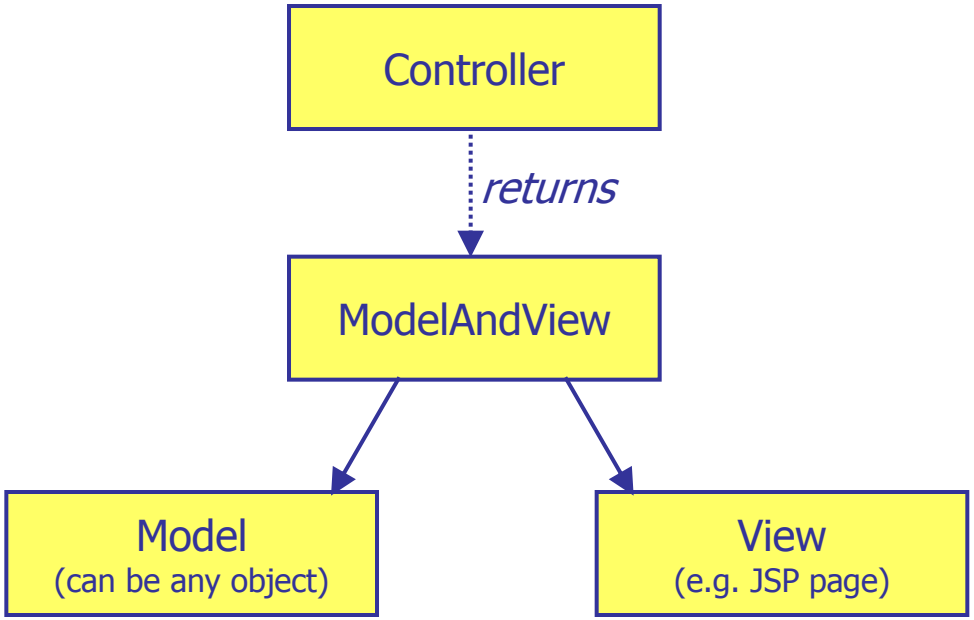
```
<!-- Just need to enable annotations ☺ -->
<context:component-scan base-package="mypackage" />
```

dispatcherservletname-servlet.xml

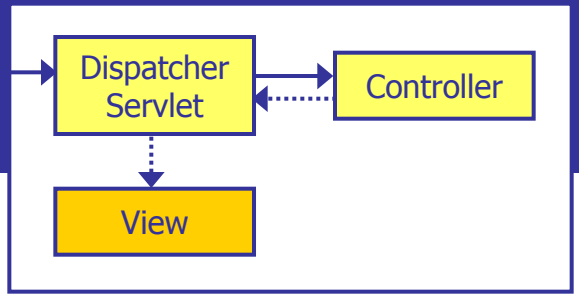
Controllers (4 of 4)



- After request handling, controllers return a `ModelAndView` result object
  - Selects the view to render the response
  - Contains the data needed for rendering



Views

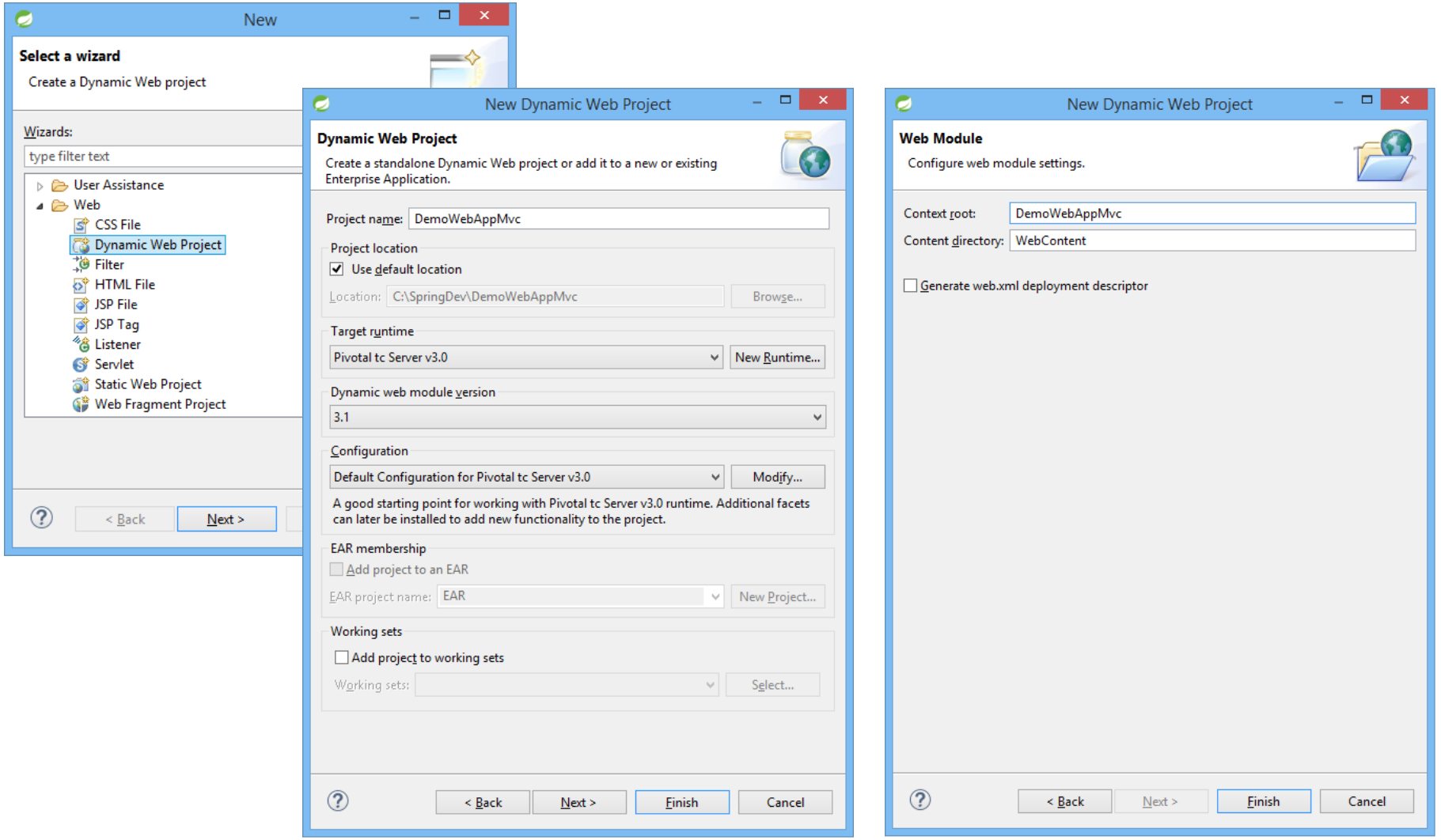


- The controller returns a view to render the model
  - The view generates the HTTP response
- A controller can return a concrete view implementation
  - E.g. `new JstlView("/WEB-INF/displayEmployees.jsp")`
  - E.g. `new EmployeeListingPdf()`
- … or a view name
  - E.g. `"displayEmployees"`
  - `DispatcherServlet` resolves name to a view implementation, by delegating to a `ViewResolver`
  - The default `ViewResolver` treats the view name as a file path relative to the Web application root folder
  - You can override this default by registering a `ViewResolver` bean with `DispatcherServlet`

3. Creating a Spring Web MVC App

- Creating a Web project
- Configuring the project
- Specifying a DispatcherServlet
- Defining a controller
- Defining a view
- Running the application
- Resolving views

Creating a Web Project



Configuring the Project

- Add necessary JAR files to project
  - Including the Spring Web jar file
  - E.g. in our examples, we added a reference to our *User Library* that included all the JARs
- You must also make sure the Spring JARs are available at run-time
  - You can either add these JARs to your Web server
  - Or add them to your `WEB-INF\lib` folder
- Also, enable Spring tooling support for the project
  - Right-click the project, then select Spring Tools | Add Spring Project Nature

Specifying a DispatcherServlet (1 of 2)

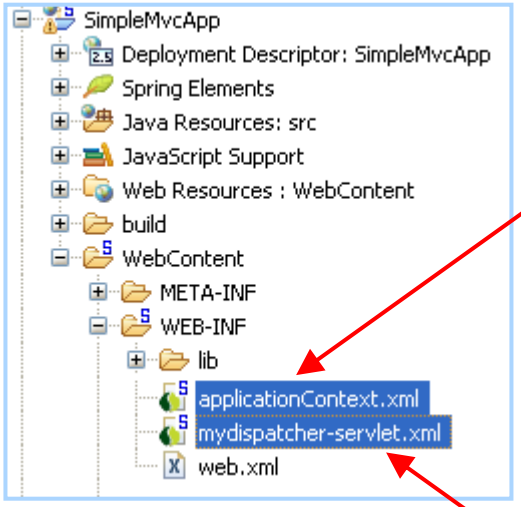
1. `<context-param>` - Optional, specify Spring context file location
2. `<listener>` - Load Spring app context automatically
3. `<servlet>` - Specify Spring `DispatcherServlet` class
4. `<servlet-mapping>` - Map URL pattern to `DispatcherServlet`

```
<web-app … >

1 <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext.xml</param-value>   <!-- Default ☺ -->
  </context-param>

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

3 <servlet>
    <servlet-name>mydispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

4 <servlet-mapping>
    <servlet-name>mydispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>
  …
```

web.xml

Specifying a DispatcherServlet (2 of 2)

- Add 2 Spring bean config files to `WEB-INF` folder




```
<beans … >

  <!-- Define application-level beans here. -->
  <!-- E.g. services, repositories, etc.    -->

</beans>
```

applicationContext.xml

```
<beans … >

  <context:component-scan base-package="mypackage" />

  <!-- Define dispatcher-level beans here.    -->
  <!-- E.g. controllers, view resolvers, etc. -->

</beans>
```

mydispatcher-servlet.xml

Defining a Controller

- Define a regular Java class
  - Annotate with `@Controller`
  - Define methods annotated with `@RequestMapping`

```
package mypackage;
…
@Controller
public class MyController {

  @RequestMapping("/page.htm")
  public ModelAndView show(HttpServletRequest request) {

    String language = request.getParameter("language");

    ModelAndView result = new ModelAndView();
    if (language != null && language.equals("French"))
      result.addObject("greeting", "Bonjour!");
    else
      result.addObject("greeting", "Hello!");

    result.setViewName("/WEB-INF/views/displayGreeting.jsp");
    return result;
  }
  …
}
```

MyController.java

Defining a View

- Typically a JSP page
  - But it could be XML/XSLT, Excel, PDF, etc

- Example:
  - JSP page
  - Accesses items in model, using JSP Expression Language (EL)

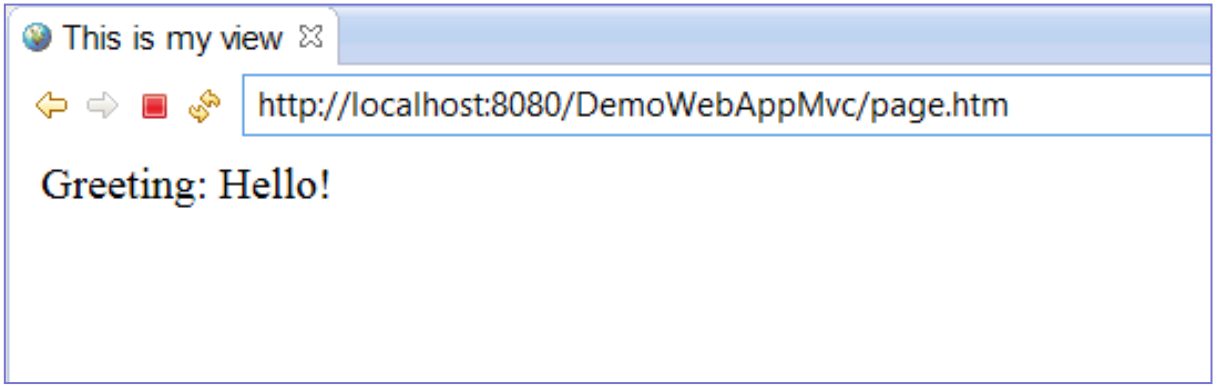
```
<html>

  <head>
    <title>This is my view</title>
  </head>

  <body>
    Greeting: ${greeting}
  </body>

</html>
```

views/displayGreeting.jsp

Running the Application

- Run the project within STS
  - Creates a WAR file
  - Deploys WAR file to application server (starts server if needed)
  - Launches a browser

- Specify a URL that the DispatcherServlet will dispatch
  - E.g. `http://localhost:8080/DemoWebAppMvc/page.htm`



Resolving Views

- Currently the controller specifies a complete URL for view
  - Verbose, error-prone, not very flexible ☹

```
result.setViewName("/WEB-INF/views/displayFullName.jsp");
```

- A better approach is to just specify the "name" of the view
  - Concise, simple, flexible ☺

```
result.setViewName("displayFullName");
```

  - For this to work, you must define a `ViewResolver` in the dispatcher servlet's config file
  - Specify the prefix/suffix parts for view URLs

```
<beans … >

  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
  </bean>

</beans>
```

mydispatcher-servlet.xml

Any Questions?