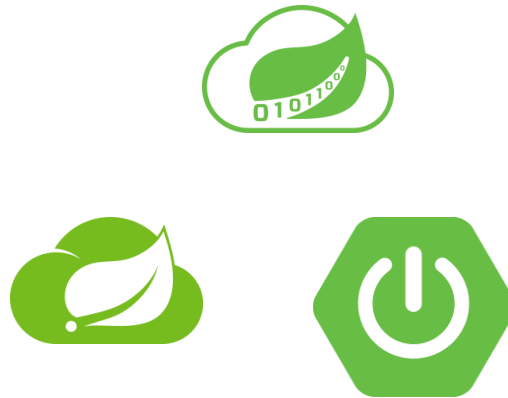
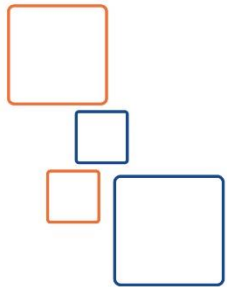




# Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND



Java™ Education  
and Technology Services



Invest In Yourself,  
**Develop** Your Career



# Course Outline

- **Lesson 1:** Spring MVC Introduction
  - (DispatcherServlet, Application and Web Application Context)
- **Lesson 2:** Spring MVC Request Lifecycle
- **Lesson 3:** Spring MVC Hello World Example
- **Lesson 4:** Handler Mappings
- **Lesson 5:** Controllers
- **Lesson 6:** View Resolvers
- **Lesson 7:** Views
- **Lesson 8:** Using Annotations
- **\*\*\* References & Recommended Reading**

## Lesson 1

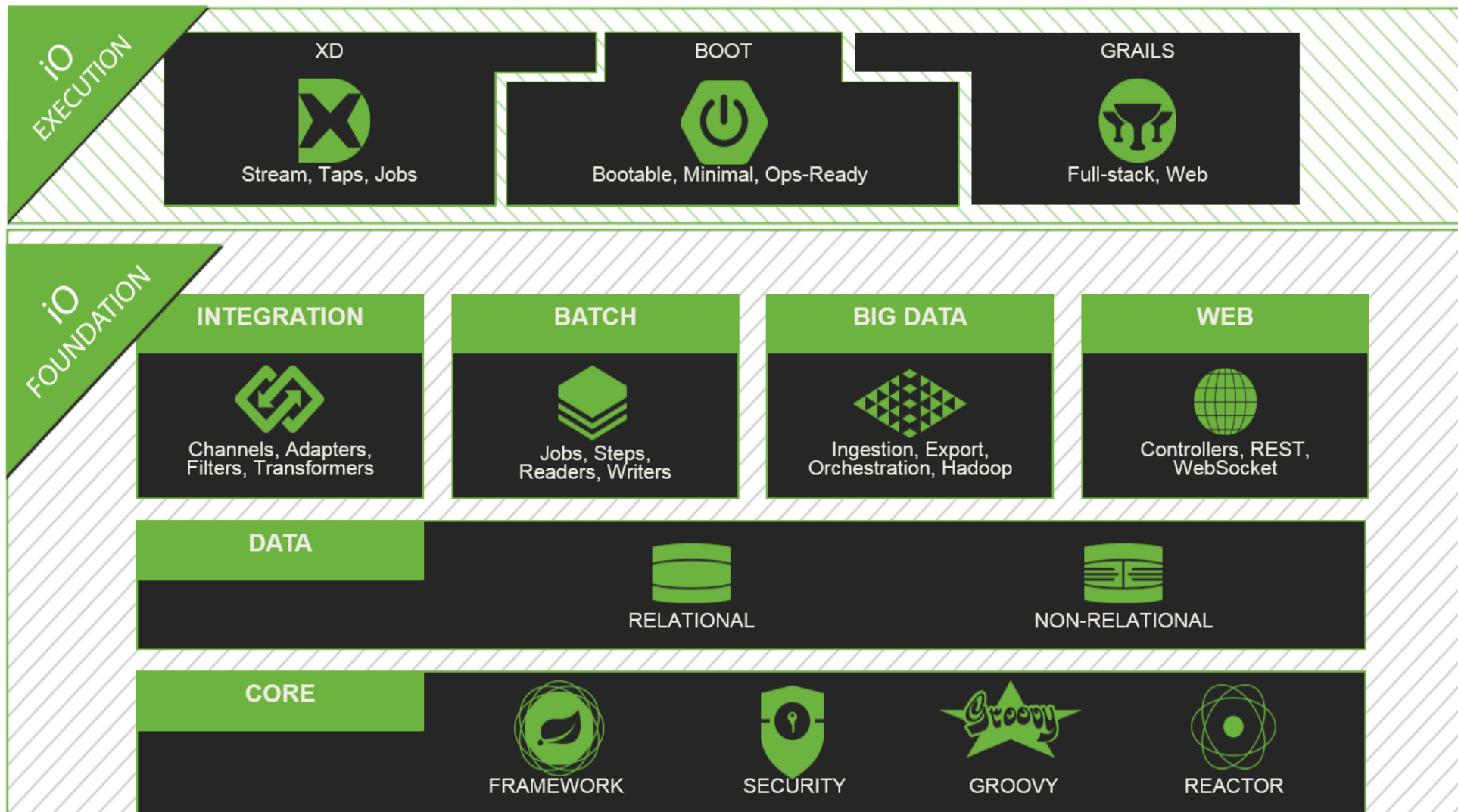
# Spring MVC Introduction

(DispatcherServlet, Application and Web  
Application Context)





# Spring Framework Modules





# Spring Framework Modules





# Spring Web MVC

- Spring Web MVC is the original web framework.
- Built on the Servlet API and has been included in the Spring Framework from the very beginning.
- The formal name "Spring Web MVC" comes from the name of its **source module** (**spring-webmvc**), but it is more commonly known as "Spring MVC".
- Parallel to Spring Web MVC, Spring Framework 5.0 introduced a reactive-stack web framework whose name "Spring WebFlux" is also based on its **source module** (**spring-webflux**).



# Spring Web MVC (Ex.)

- Spring MVC like any web frameworks is designed around the front controller pattern.
  - Where a central Servlet called **DispatcherServlet**, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components.
  - This model is flexible and supports diverse workflows.
- The **DispatcherServlet** as any Servlet, needs to be declared and mapped according to the Servlet specification by using **Java configuration** or in **web.xml**.
- Then, The **DispatcherServlet** uses Spring configuration to discover the delegate components it needs for request mapping, view resolution, exception handling, and more.



# Spring **Application** and **Web Application** Context

- In Spring Web Applications, there are two types of container, each of which is configured and initialized differently.
  - The "Application Context".
  - The "Web Application Context".





# Spring **Application** and Web Application Context

- **Application Context** is the container defined in the **web.xml** and initialized:
  - Either by a **ContextLoaderListener** Or **ContextLoaderServlet**.
- This context might, for instance, contain components such as middle-tier transactional services, data access objects, or other objects that you might want to use (and re-use) across the application.
- By default it is looking up for file called "**applicationContext.xml**" in **WEB-INF** that contains your definitions.
- There will be one application context per application.



# Spring **Application** and Web Application Context

- The configuration would look something like this:

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:*-context.xml,
    /WEB-INF/spring/applicationContext.xml
  </param-value>
</context-param>
```

- I am asking spring to load all files from the classpath that match **\*-context.xml** and **/WEB-INF/spring/ApplicationContext.xml** and create an **Application Context** from it.



# Spring **Application** and Web Application Context

- The purpose of the ContextLoaderListener is two function:
  1. To tie the lifecycle of the ApplicationContext to the lifecycle of the ServletContext.
  2. To automate the creation of the ApplicationContext, so you don't have to write explicit code to do create it - it's a convenience function.



# Spring Application and Web Application Context

- **Web Application Context** is the child context of the application context.
- Each DispatcherServlet defined in a Spring web application will have an associated `WebApplicationContext`.
- You can define more than DispatcherServlet in the same web application at same time.
- All Created DispatcherServlet(s) share the same application context but with different web application context.
- By default it is looking up for file called "<servletname>-servlet.xml" in **WEB-INF** that contains your springmvc definition.



# Spring Application and Web Application Context

- The configuration would look something like this:

```
<servlet>
  <servlet-name>mvc-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:*-servlet.xml,
      /WEB-INF/spring/springmvc-servlet.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- I am asking spring to load all files from the classpath that match `*-servlet.xml` and `/WEB-INF/spring/springmvc-servlet.xml` and create an **Web Application Context** from it.



# Spring Application and Web Application Context

- Whatever beans are available in the ApplicationContext can be referred to from each WebApplicationContext.
- It is **a best practice** to keep a clear separation between middle-tier services such as business logic components and data access classes (that are typically defined in the ApplicationContext) and web-related components such as controllers and view resolvers (that are defined in the WebApplicationContext per Dispatcher Servlet).

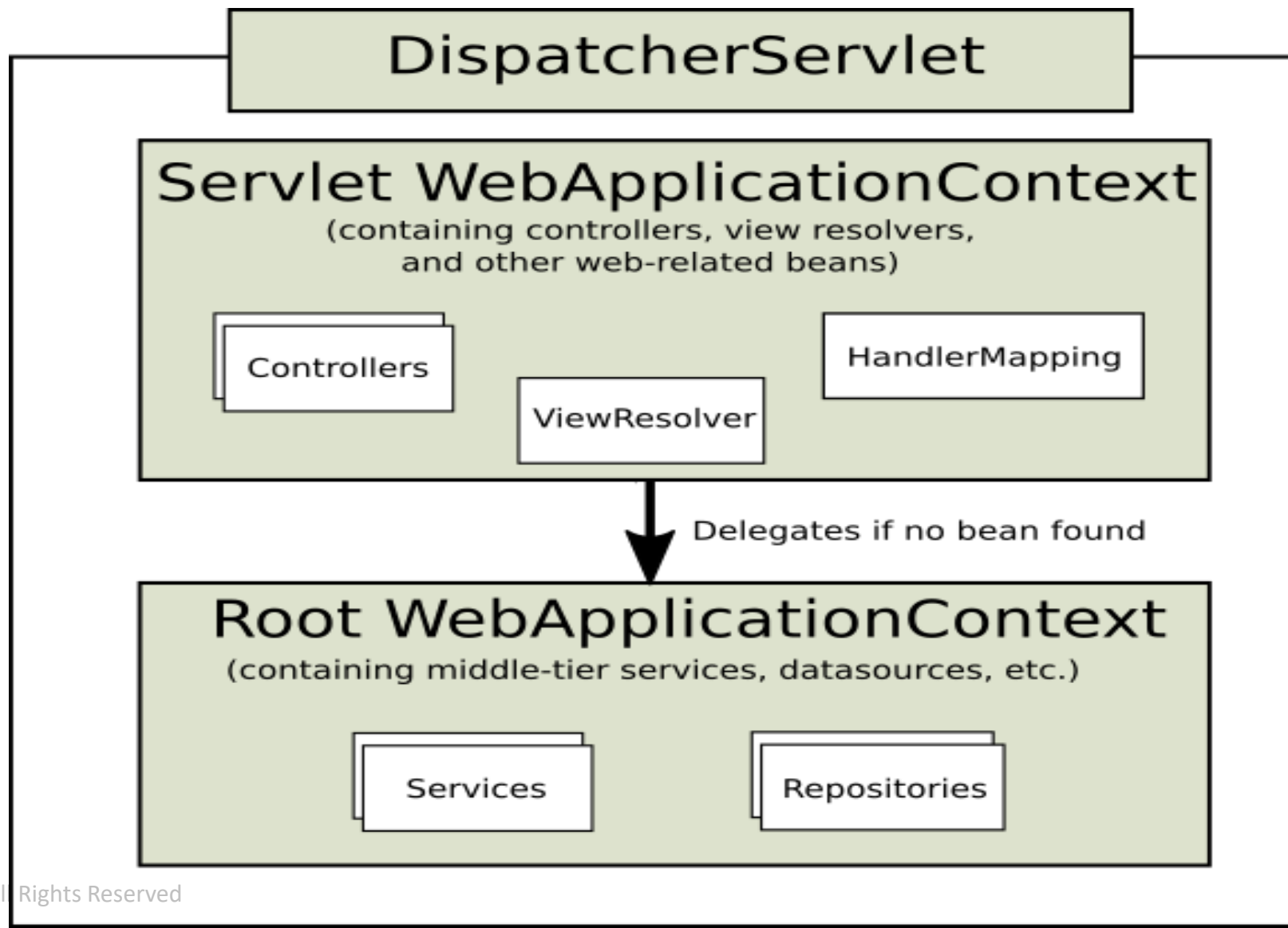


# Spring Application and Web Application Context

- There is a list to understand Application Contexts and Web Application Contexts
  - Application-Contexts are hierarchical and so are WebApplicationContexts.
  - ContextLoaderListener creates a root web-application-context for the web-application and puts it in the ServletContext.
    - This context can be used to load and unload the spring-managed beans irrespective of what technology is being used in the controller layer (Struts or Spring MVC).
  - DispatcherServlet creates its own WebApplicationContext and the handlers/controllers/view-resolvers are managed by this context.
  - When ContextLoaderListener is used in same configuration with DispatcherServlet, a root web-application-context is created first as said earlier and a child-context is also created by DispatcherServlet and is attached to the root application-context.



# Spring Application and Web Application Context







# Spring **Application** and **Web Application** Context

- You can configure your DispatcherServlet by:
  - Either by Servlet Mapping for DispatcherServlet.
  - Or by implement Custom WebApplicationInitializer
  - Or by extend AbstractAnnotationConfigDispatcherServletInitializer



# Using DispatcherServlet

- You can configure your DispatcherServlet by:
  - Either by Servlet Mapping for DispatcherServlet.
  - You made it by define your DispatcherServlet and It's mapping as follows:

```
<servlet>
  <servlet-name>mvc-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- In the previous example we define the dispatcher servlet with default configuration settings so they lookup for file with standard name (**<servlet-name>-servlet.xml**) called **"mvc-dispatcher-servlet.xml"** that contains beans definition for Spring MVC components.



# Using DispatcherServlet (Ex.)

- You also could override the default configuration to your own as follows:

```
<servlet>
  <servlet-name>mvc-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/mvc.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- In the previous example we define the dispatcher servlet with custom configuration settings in **(WEB-INF/spring/mvc.xml)**.



# Using DispatcherServlet (Ex.)

- You also could override the default configuration to your own as follows:

```
<servlet>
  <servlet-name>mvc-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      classpath:*-servlet.xml,
      /WEB-INF/spring/springmvc-servlet.xml
    </param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

- In the previous example we define the dispatcher servlet with custom configuration settings in all files from the classpath that match **\*-servlet.xml** and **/WEB-INF/spring/springmvc-servlet.xml**.



# Using DispatcherServlet (Ex.)

- For All previous declared syntax for **DispatcherServlet**, you must specify the servlet-mapping for your dispatcher servlet to match your request patterns as follows for example:

```
<servlet-mapping>  
    <servlet-name>mvc-dispatcher</servlet-name>  
    <url-pattern>*.htm</url-pattern>  
</servlet-mapping>
```

- For spring MVC convention we use this pattern for URLs ".htm" but you could use any pattern you want



# Using WebApplicationInitializer

- You can configure your DispatcherServlet by:
  - Or by implement Custom WebApplicationInitializer
- You made it by
  - Clearing your web.xml from any spring configuration.
  - Define Class that implement `org.springframework.web.WebApplicationInitializer` and override `onStartup` method to define spring context with your custom configuration.
- Here we define our application context programmatically and pass it to DispatcherServlet.



# Using WebApplicationInitializer (Ex.)

- If your configuration it's XML Based, you could use **XmlWebApplicationContext** to create context as root configuration instead of @Configuration:

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {  
  
    @Override  
    public void onStartup(ServletContext servletContext) {  
        // Load Spring MVC configuration using XmlWebApplicationContext  
        XmlWebApplicationContext applicationContext  
            = new XmlWebApplicationContext();  
        applicationContext.setConfigLocation("/WEB-INF/spring/mvc.xml");  
        // Create and register the DispatcherServlet  
        DispatcherServlet servlet = new DispatcherServlet(applicationContext);  
        ServletRegistration.Dynamic registration  
            = servletContext.addServlet("mvc-dispatcher", servlet);  
        registration.setLoadOnStartup(1);  
        registration.addMapping("*.htm");  
    }  
}
```



# Using WebApplicationInitializer (Ex.)

- If your configuration it's XML Based, you could user `AnnotationConfigWebApplicationContext` to create context as root configuration using `@Configuration`:

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {  
  
    @Override  
    public void onStartup(ServletContext servletContext) {  
        // Load Spring MVC using AnnotationConfigWebApplicationContext  
        AnnotationConfigWebApplicationContext applicationContext  
            = new AnnotationConfigWebApplicationContext();  
        applicationContext.register(ApplicationConfiguration.class);  
        // Create and register the DispatcherServlet  
        DispatcherServlet servlet = new DispatcherServlet(applicationContext);  
        ServletRegistration.Dynamic registration  
            = servletContext.addServlet("mvc-dispatcher", servlet);  
        registration.setLoadOnStartup(1);  
        registration.addMapping("*.htm");  
    }  
}
```





# Using WebApplicationInitializer (Ex.)

- In Your Configuration file as the previous example called "**ApplicationConfiguration**" you could specify the linking to some resources or declared the spring MVC components inside it.

```
@Configuration
//Either in class-path
//@ImportResource("classpath:/com/jediver/spring/cfg/mvc.xml")
//or in class-path
@ImportResource("WEB-INF/spring/mvc.xml")
//or in both
//@ImportResource({"WEB-INF/spring/mvc.xml",
//      "classpath:/com/jediver/spring/cfg/mvc.xml"})
public class ApplicationConfiguration {
    ...
}
```



# Using AbstractAnnotationConfigDispatcherServletInitializer

- You can configure your DispatcherServlet by:
  - **extending AbstractAnnotationConfigDispatcherServletInitializer**
- You made it by
  - Clearing your web.xml from any spring configuration.
  - Define Class that extend **org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer** and override `getRootConfigClasses()`, `getServletConfigClasses()` and `getServletMappings()` methods to define spring context with your configuration.



# Using WebApplicationInitializer (Ex.)

```
public class MyWebApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{};
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{ApplicationConfiguration.class};
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"*.htm"};
    }
}
```



# Using AbstractAnnotationConfigDispatcherServletInitializer

- It also provide abstract way to add filter instances and have them be automatically mapped to the DispatcherServlet :

```
public class MyWebApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Filter[] getServletFilters() {
        return new Filter[]{
            new HiddenHttpMethodFilter(), new CharacterEncodingFilter();
        }
    }
}
```



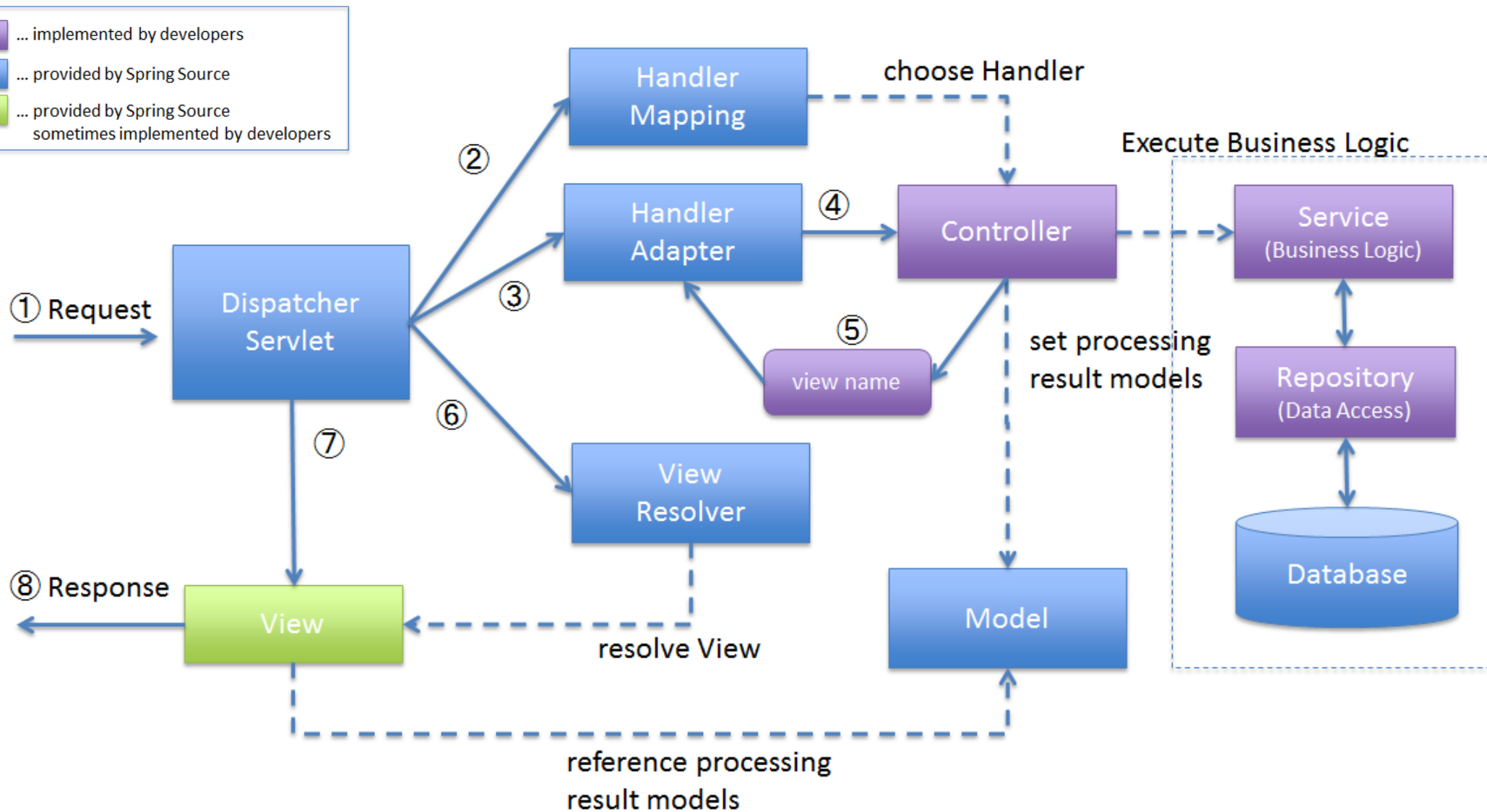
## Lesson 2

# Spring MVC Request Lifecycle





# Request Life Cycle





# Request Life Cycle (Ex.)

- When the **request** leaves the browser, it carries information (e.g. the requested URL, the information submitted in a form ,etc..)
1. The first step in the request's life cycle is Spring's DispatcherServlet receives the request
    - It act as façade class for user request handling.
  2. DispatcherServlet dispatches (delegate) the task of selecting an appropriate controller to **HandlerMapping**.
    - **HandlerMapping** selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.



# Request Life Cycle (Ex.)

3. DispatcherServlet dispatches (delegate) the task of executing of business logic of Controller to **HandlerAdapter**.
4. HandlerAdapter calls the business logic process of Controller.
  - Also responsible for data binding and conversion between the row request and the controller data binding.
5. Controller
  - Executes the business logic.
  - Sets the processing result in Model.
  - Returns the logical name of view to HandlerAdapter.





# Request Life Cycle (Ex.)

6. DispatcherServlet dispatches (delegate) the task of resolving the View corresponding to the View name to **ViewResolver**.
  - **ViewResolver** returns the View that mapped to this View name.
7. DispatcherServlet dispatches (delegate) the rendering process to returned View to generate this type of view.
8. View renders Model data and returns the final response to client.



## Lesson 3

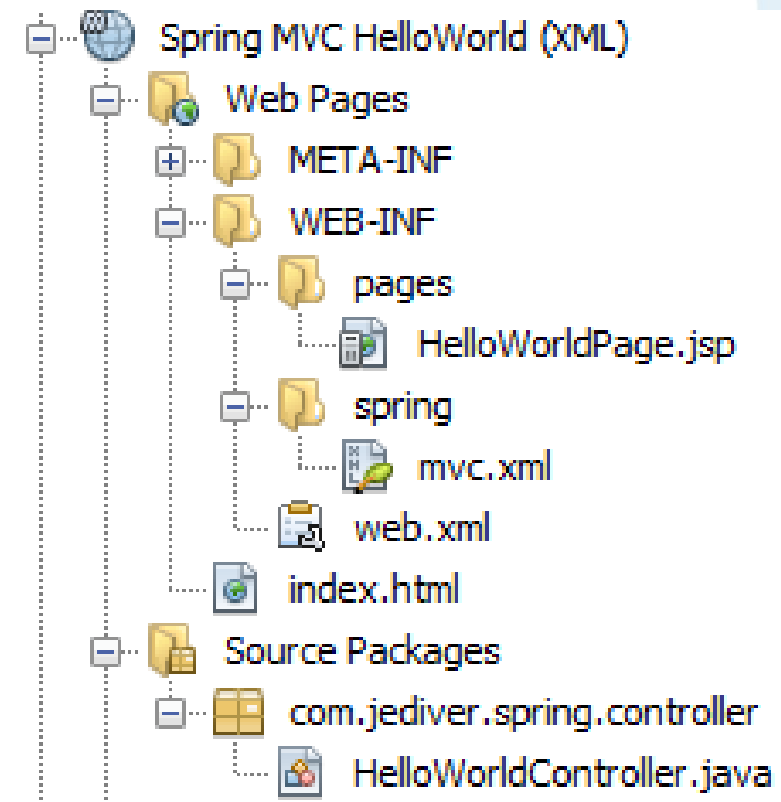
# Spring MVC Hello World Example





# Hello World Example

- Let us make our first Hello World example:
  1. First define the web application context in web.xml as described before.
  2. Then define your controller class to receive the request and handle it.
  3. Then define your application context file and/or class to define your bean definitions.
  4. Then define your pages as defined with this hierarchy for example.





# Hello World Example (Ex.)

1. First define the web application context in web.xml as described before for example as follows:

```
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
    http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <servlet>
    <servlet-name>mvc-dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring/mvc.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>mvc-dispatcher</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>
</web-app>
```



# Hello World Example (Ex.)

2. Then define your controller class to receive the request and handle it.
  - One of the techniques to declare **Controller** as in lifecycle is to declare class that extend AbstractController and override handleRequestInternal() that return ModelAndView object to Handler Adapter.

```
public class HelloWorldController extends AbstractController {  
  
    @Override  
    protected ModelAndView handleRequestInternal(  
        HttpServletRequest request,  
        HttpServletResponse response) throws Exception {  
        ModelAndView model = new ModelAndView("HelloWorldPage");  
        model.addObject("msg", "Hello From JEDiver");  
        return model;  
    }  
}
```



# Hello World Example (Ex.)

3. Then define your application context file and/or class to define your bean definitions.
  - In the following definitions we declare a bean definition for HelloWorldController.

```
<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 name="helloController"  
        class="com.jediver.spring.controller.HelloWorldController" />
```



# Hello World Example (Ex.)

3. Then define your application context file and/or class to define your bean definitions.
  - Then we declare one of the types of **Handler Mapping** called "**SimpleUrlHandlerMapping**".
  - **SimpleUrlHandlerMapping** enable you to define the URLs and reference to the controllers id or name.

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/welcome.htm">helloController</prop>
      </props>
    </property>
  </bean>
```



# Hello World Example (Ex.)

3. Then define your application context file and/or class to define your bean definitions.
  - Finally we declare one of the types of **View Resolver** called "InternalResourceViewResolver".
  - It enable you to map view name with physical page with prefix and suffix, e.g. if the logical view name is "HelloWorldPage" so the physical page name is "/WEB-INF/pages/HelloWorldPage.jsp".

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
    <property name="prefix">
      <value>/WEB-INF/pages/</value>
    </property>
    <property name="suffix">
      <value>.jsp</value>
    </property>
  </bean>
</beans>
```





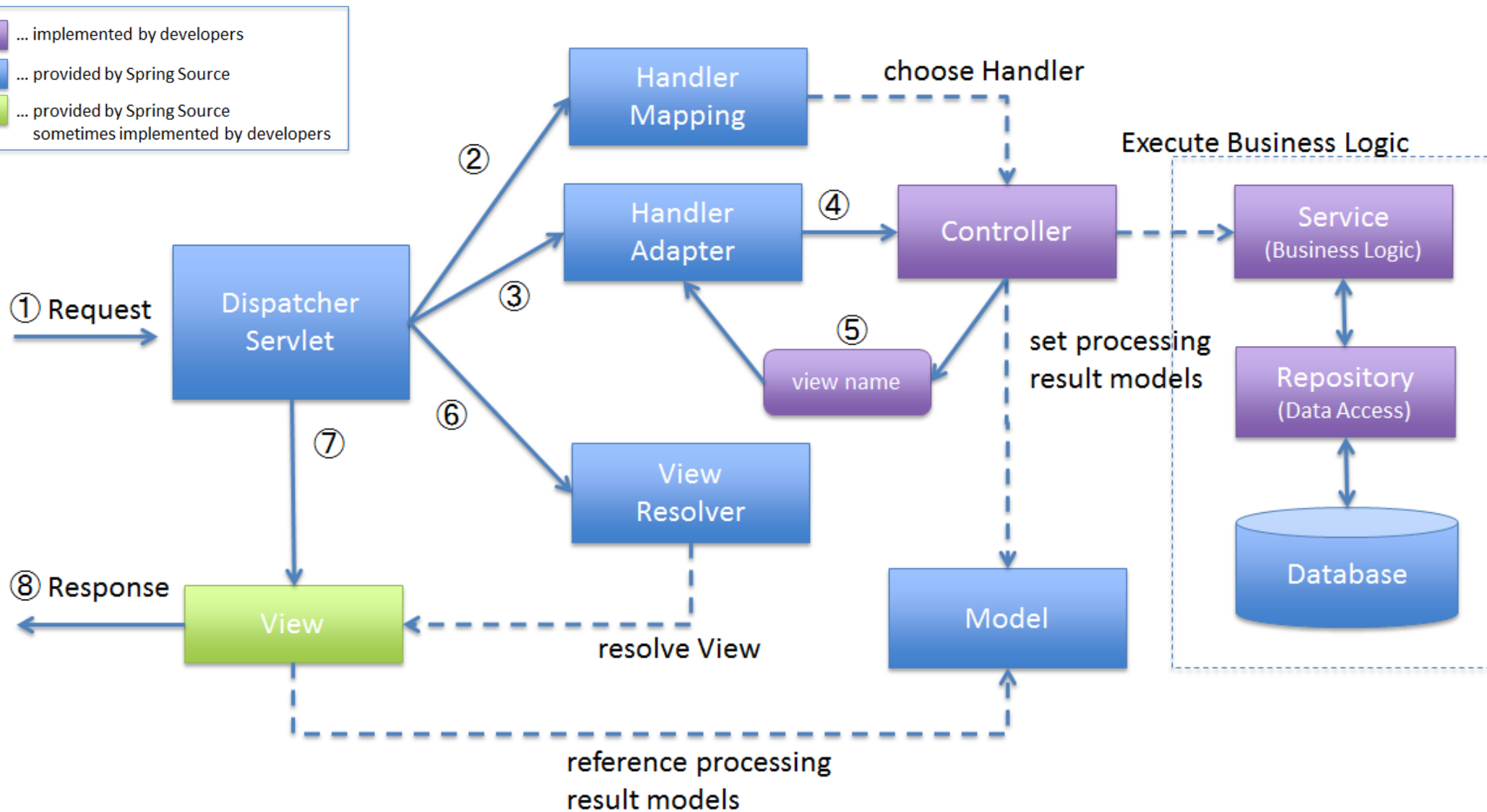
# Hello World Example (Ex.)

4. Then define your pages as defined with this hierarchy for example.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
  <body>
    <h1>Spring MVC Hello World Example</h1>
    <h2>${msg}</h2>
  </body>
</html>
```



# Request Life Cycle





# Discuss Request Life cycle in Hello World Example

- First The user initiate the request by request the URL  
(<http://localhost:8080/HelloWorldXML/welcome.htm>)
1. DispatcherServlet receives your request **because it mapped with url "welcome.htm"**.
  2. DispatcherServlet asks **SimpleUrlHandlerMapping** for this url, then  
**SimpleURLHandlerMapping** respond with an instance from the controller (based on scope rules) that called "**helloController**" and his own handler adapter to the DispatcherServlet.
  3. DispatcherServlet hand over the request to HandlerAdapter to execute the Controller.
  4. HandlerAdapter calls the Controller in **handleRequestInternal()** method and send it the request and the response.



# Discuss Request Life cycle in Hello World Example

5. Controller executes the business logic and return **ModelAndView** object that represent:
  - Model that contains your data which passed to view that cached until the view is rendered.
  - Contains "**msg**" ==> "**Hello From JEDiver**"
  - View that contains the logical name "**HelloWorldPage**" of the view to **HandlerAdapter**.
6. DispatcherServlet ask your **InternalResourceViewResolver** for this logic name so it respond with the physical name of page "**/WEB-INF/pages/HelloWorldPage.jsp**".
7. DispatcherServlet send the model data to the view "**/WEB-INF/pages/HelloWorldPage.jsp**".
8. Our jsp (Servlet in fact) take this data and define response body and send it to the client.



# Discuss Request Life cycle in Hello World Example

- If you didn't define your own bean definition of any of the classes in the life cycle, spring will provide ones by default defined in properties file named

"`org/springframework/web/servlet/DispatcherServlet.properties`" as below:

- `org.springframework.web.servlet.LocaleResolver`  
    =`org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver`
- `org.springframework.web.servlet.ThemeResolver`  
    =`org.springframework.web.servlet.theme.FixedThemeResolver`
- `org.springframework.web.servlet.HandlerMapping`  
    =`org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,`  
    `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping`



# Discuss Request Life cycle in Hello World Example

- `org.springframework.web.servlet.HandlerAdapter`
  - =`org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,`
  - `org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,`
  - `org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter`
- `org.springframework.web.servlet.HandlerExceptionResolver`
  - =`org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionResolver,`
  - `org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandler,`
  - `org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver`



# Discuss Request Life cycle in Hello World Example

- `org.springframework.web.servlet.RequestToViewNameTranslator`  
`=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator`
- `org.springframework.web.servlet.ViewResolver`  
`=org.springframework.web.servlet.view.InternalResourceViewResolver`
- `org.springframework.web.servlet.FlashMapManager`  
`=org.springframework.web.servlet.support.SessionFlashMapManager`



## Lesson 4

# Handler Mappings







# Handler Mappings

- Types of **HandlerMapping** Interface:
  - Interface to be implemented by objects that define a mapping between **requests** and **handler objects**.
- This class can be implemented by application developers.
- By Default if you didn't define any **HandlerMapping**, Spring will provide:
  - **BeanNameUrlHandlerMapping**
  - and **RequestMappingHandlerMapping** as default Handler.

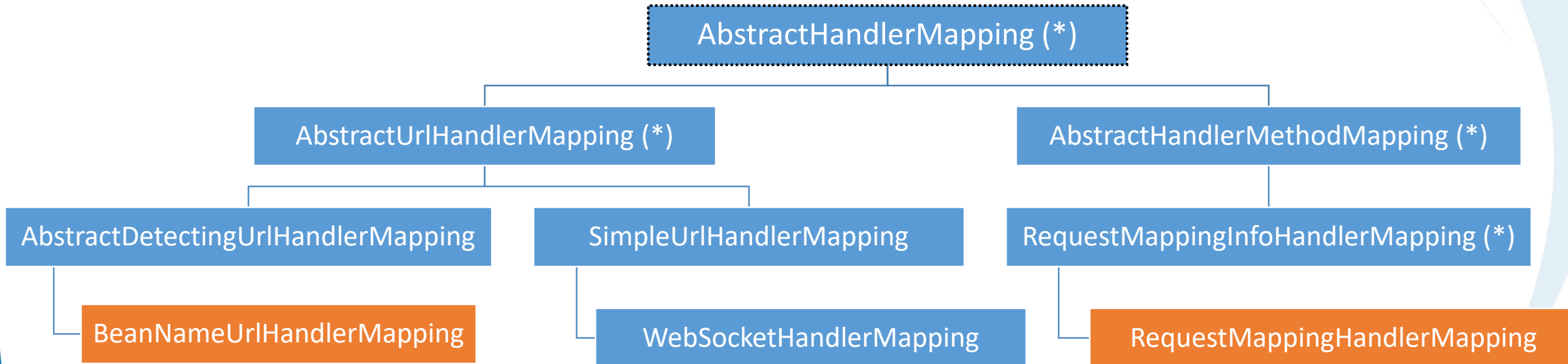


# Handler Mappings (Ex.)

- The ability to parameterize this mapping is a powerful and unusual capability of this MVC framework.
  - For example, it is possible to write a custom mapping based on session state, cookie state or many other variables. No other MVC framework seems to be equally flexible.
- **Note:** Implementations can implement the **Ordered** interface to be able to specify a sorting order and thus a priority for getting applied by DispatcherServlet.
  - Non-Ordered instances get treated as lowest priority.



# Handler Mappings Types



- All marked with (\*) you have to implement the full functionality of handler mapping you could use it if you want to override the logic of normal handler mapping.



# Handler Mappings Types (Ex.)

- So we can use:
  - AbstractDetectingUrlHandlerMapping
  - BeanNameUrlHandlerMapping
  - SimpleUrlHandlerMapping
  - WebSocketHandlerMapping
  - RequestMappingHandlerMapping
- Also if you are using older version of **spring before 5** you can find:
  - ControllerClassNameHandlerMapping



# Handler Mappings Types (Ex.)

- **AbstractDetectingUrlHandlerMapping**
- This is abstract parent that define mapping between url pattern to the controller identity.
- You can use it by:
  - Extend the **AbstractDetectingUrlHandlerMapping** and
  - Override `protected String[] determineUrlsForHandler(String controllerId);`
- Which takes the controller id and return array of string represent the keys map for this controller's URLs.



# Handler Mappings Types (Ex.)

- **AbstractDetectingUrlHandlerMapping**
- As the following example:

```
public class MyHandlerMapping extends AbstractDetectingUrlHandlerMapping {  
  
    @Override  
    protected String[] determineUrlsForHandler(String controllerId) {  
        String[] mappings = null;  
        if (controllerId.equalsIgnoreCase("helloController")) {  
            mappings = new String[1];  
            mappings[0] = "/welcome.htm";  
        }  
        return mappings;  
    }  
}
```

- **Note:** Don't forget to create bean from this class.



# Handler Mappings Types (Ex.)

- **BeanNameUrlHandlerMapping**
- One of the two default handler mapping if you didn't specify another one.
- Also you could define it explicitly as follows:

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>
```

- Controller mapping depends on its given name, as this name will be considered as its corresponding URL pattern.
- And you could define your controller as follows:

```
<bean name="/hello.htm"  
      class="com.jediver.spring.controller.HelloWorldController" />
```



# Handler Mappings Types (Ex.)

- **SimpleUrlHandlerMapping**

- Controllers are mapped to URLs using a property collection defined in the Spring application context.
- Using this SimpleUrlHandlerMapping, you don't have to give your controller a special name.
- You can directly assign a URL pattern to your controller.
- Mappings to bean names can be set via the "mappings" property, in a form accepted by the `java.util.Properties` class, like as follows:
  - `/welcome.html=ticketController`                      `/show.html=ticketController`
- The syntax is `PATH=HANDLER_BEAN_NAME`. If the path doesn't begin with a slash, one is prepended.





# Handler Mappings Types (Ex.)

- SimpleUrlHandlerMapping
- As the following example:

```
<bean name="helloController"  
      class="com.jediver.spring.controller.HelloWorldController" />  
<bean id="urlMapping"  
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">  
  <property name="mappings">  
    <props>  
      <prop key="/welcome.htm">helloController</prop>  
    </props>  
  </property>  
</bean>
```



# Handler Mappings Types (Ex.)

- **WebSocketHandlerMapping**
- An extension of SimpleUrlHandlerMapping.
- That is also a SmartLifecycle container and propagates start and stop calls to any handlers that implement Lifecycle.
- The handlers are typically expected to be **WebSocketHttpRequestHandler** or **SockJsHttpRequestHandler**.



# Handler Mappings Types (Ex.)

- **RequestMappingHandlerMapping**
- Applied only when using annotation configuration only in mapping controllers.
- Creates **RequestMappingInfo** instances from type and method-level **@RequestMapping** annotations in **@Controller** classes



# Handler Mappings Types (Ex.)

- **ControllerClassNameHandlerMapping**
- The **URL pattern** will be the same as the **controller class name**.
- Spring will automatically map controllers to URL pattern using controllers' class names.
- It Creates URL based on:
  - Drops the Controller portion (if it exists).
  - Lowercase the remaining text.
  - Add slash '/' to the beginning
  - Add .htm to the end.



# Handler Mappings Types (Ex.)

- **ControllerClassNameHandlerMapping**
- As the following example:

```
<bean id="urlMapping"  
      class="org.springframework.web.servlet.mvc.ControllerClassNameHandlerMapping"/>  
  
<bean name="helloController"  
      class="com.jediver.spring.controller.HelloWorldController" />
```

- The previous example define this controller **HelloWorldController** will response to the URL  
**/helloworld.htm**



# Including Multiple Handler Mappings

- Including Multiple Handler Mappings In The Same Application:
- All Spring Handler Mappings implement Ordered interface.
- You can declare more than one handler mapping in the same application and set its precedence using the order property.
- You could specify the order property for HandlerMapping.
- Note:
- The **lower the value** of the order property, the **higher the priority**.



# Including Multiple Handler Mappings (Ex.)

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
    <property name="order" value="2"/>
</bean>

<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="order" value="1"/>
    <property name="mappings">
        <props>
            <prop key="/welcome.htm">helloController</prop>
        </props>
    </property>
</bean>
```



## Lesson 5 Controllers





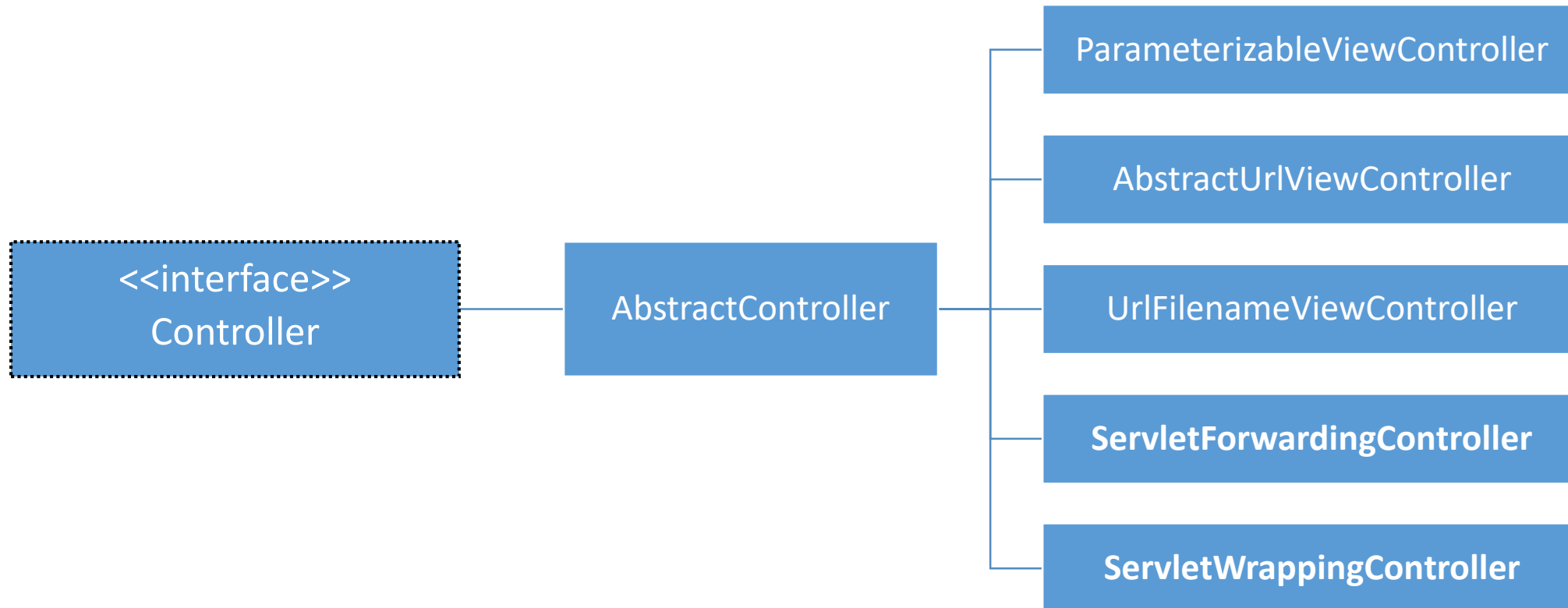


# What is Controllers?

- Controllers are responsible for processing HTTP requests, composing the response objects, and passing control back to the Dispatcher Servlet.
- It is preferable that controller delegate the responsibility for business logic to the service layer.
- Almost all controllers are singletons, so they should be stateless as they handle concurrent requests.
- Spring MVC has a rich hierarchy of Controllers.

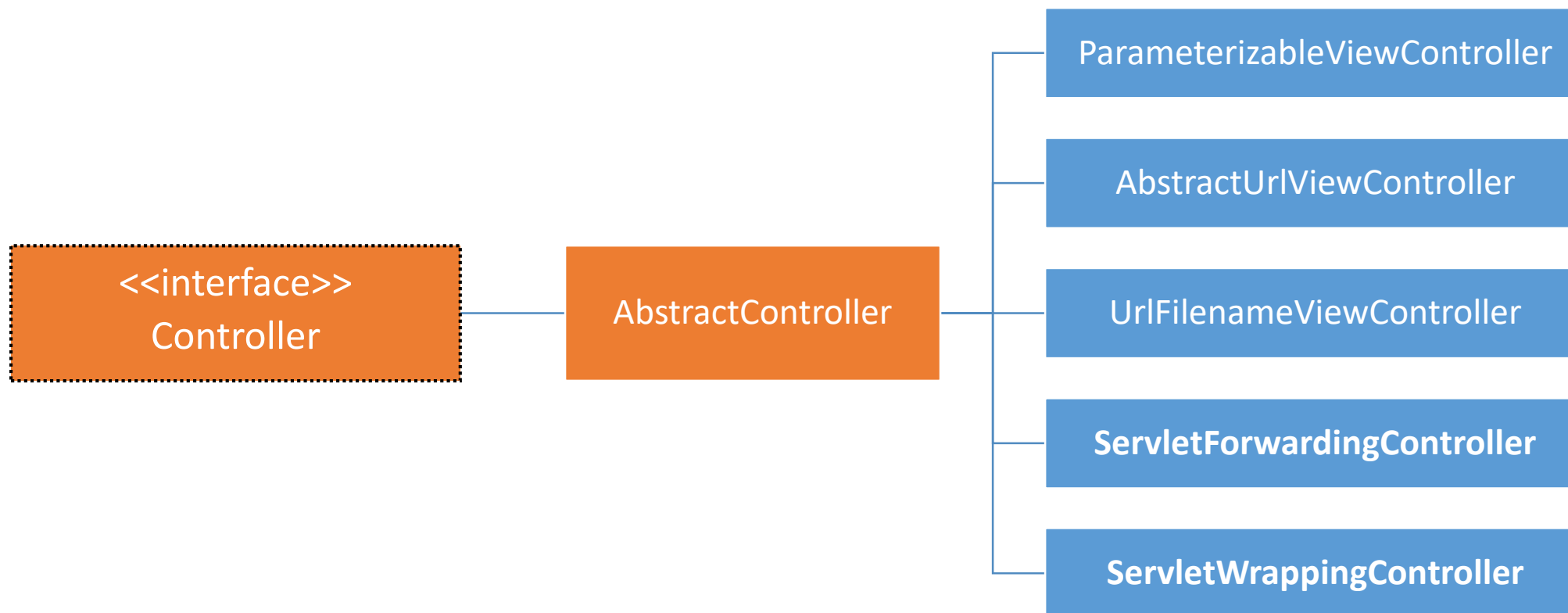


# Controllers Hierarchy





# Controllers Hierarchy





# AbstractController

- Convenient superclass for controller implementations.
- Using the Template Method design pattern.
- If your controller require little more functionality than is afforded by basic java servlets, then you'll need a simple controller.



# AbstractController (Ex.)

- Workflow:
  1. `handleRequest()` will be called by the DispatcherServlet (through HandlerAdapter).
  2. Inspection of `supported methods (GET, POST, .. etc)`
    - (ServletException if request method is not support)
  3. If `session` is required, try to get it.
    - (ServletException if not found)
  4. Set caching headers if needed according to the `cacheSeconds` property
  5. Call abstract method `handleRequestInternal()`
    - (synchronization on HttpSession is optional)



# AbstractController (Ex.)

- Exposed configuration properties:

name	default	description
supportedMethods	GET,POST	<ul style="list-style-type: none"><li>Comma-separated (CSV) list of methods supported by this controller. such as GET, POST and PUT</li></ul>
requireSession	false	<ul style="list-style-type: none"><li>Whether a session should be required for requests to be able to be handled by this controller.</li><li>This ensures that derived controller can call <code>request.getSession()</code> to retrieve a session without fear of null pointers.</li><li>If no session can be found while processing the request, a <code>ServletException</code> will be thrown</li></ul>



# AbstractController (Ex.)

- Exposed configuration properties:

name	default	description
cacheSeconds	-1	<ul style="list-style-type: none"><li>Indicates the amount of seconds to include in the cache header for the response following on this request.</li><li>0 (zero) will include headers for no caching at all.</li><li>-1 (the default) will not generate <i>any headers</i>.</li><li>Any positive number will generate headers that state the amount indicated as seconds to cache the content</li></ul>
synchronizeOnSession	false	<ul style="list-style-type: none"><li>Whether the call to <code>handleRequestInternal</code> should be <b>synchronized around the <code>HttpSession</code></b>, to serialize invocations from the same client.</li><li>No effect if there is no <code>HttpSession</code>.</li></ul>



# AbstractController (Ex.)

- There are two ways to make an extremely simple controller:
  - Implement `org.springframework.web.servlet.mvc.Controller` interface
  - Extend `org.springframework.web.servlet.mvc.AbstractController` class
- You can write your request handling code by overriding the method on interface and class :
  - `public ModelAndView handleRequest (HttpServletRequest hsr, HttpServletResponse hsr1);`
- You can write your request handling code by overriding the method on class only:
  - `public ModelAndView handleRequestInternal(HttpServletRequest hsr, HttpServletResponse hsr1);`





# AbstractController (Ex.)

```
public class HelloWorldController implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest hsr,
        HttpServletResponse hsr1) throws Exception {
        ModelAndView model = new ModelAndView("HelloWorldPage");
        model.addObject("msg", "Hello From JEDiver using Controller");
        return model;
    }
}

public class HelloWorldController1 extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(
        HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        ModelAndView model = new ModelAndView("HelloWorldPage");
        model.addObject("msg", "Hello From JEDiver using AbstractController");
        return model;
    }
}
```



# What is ModelAndView?

- Holder for both Model and View in the web MVC framework.
- Note that **these are entirely distinct**.
  - This class merely holds both to make it possible for a controller to return both model and view in a single return value.
- Represents a model and view returned by a handler adapter, to be resolved by a DispatcherServlet.
- The controller builds up the objects that make up the response (the Model) and chooses which View display next.
- Class **org.springframework.web.servlet.ModelAndView**



# What is ModelAndView? (Ex.)

- The view can take the form of:
  - A String view name which will need to be resolved by a ViewResolver object.
  - A View object can be specified directly.
  - The model is a Map, allowing the use of multiple objects keyed by name.
- The Model is a Map, allowing the use of multiple objects keyed by name.



# What is ModelAndView? (Ex.)

- Useful Constructors:
- ModelAndView()
  - Default constructor for bean-style usage: populating bean properties instead of passing in constructor arguments.
- ModelAndView (String viewName)
  - Convenient constructor when there is no model data to expose.
- ModelAndView (String viewName, HttpStatus status)
  - Create a new ModelAndView given a view name and HTTP status.
- ModelAndView (String viewName, Map<String,?> model)
  - Create a new ModelAndView given a view name and a model.



# What is ModelAndView? (Ex.)

- Useful Constructors:
- `ModelAndView (String viewName, Map<String,?> model, HttpStatus status)`
  - Create a new ModelAndView given a view name, model, and HTTP status.
- `ModelAndView (String viewName, String modelName, Object modelObject)`
  - Convenient constructor to take a single model object.
- `ModelAndView (View view)`
  - Convenient constructor when there is no model data to expose.
- `ModelAndView (View view, Map<String,?> model)`
  - Create a new ModelAndView given a View object and a model.

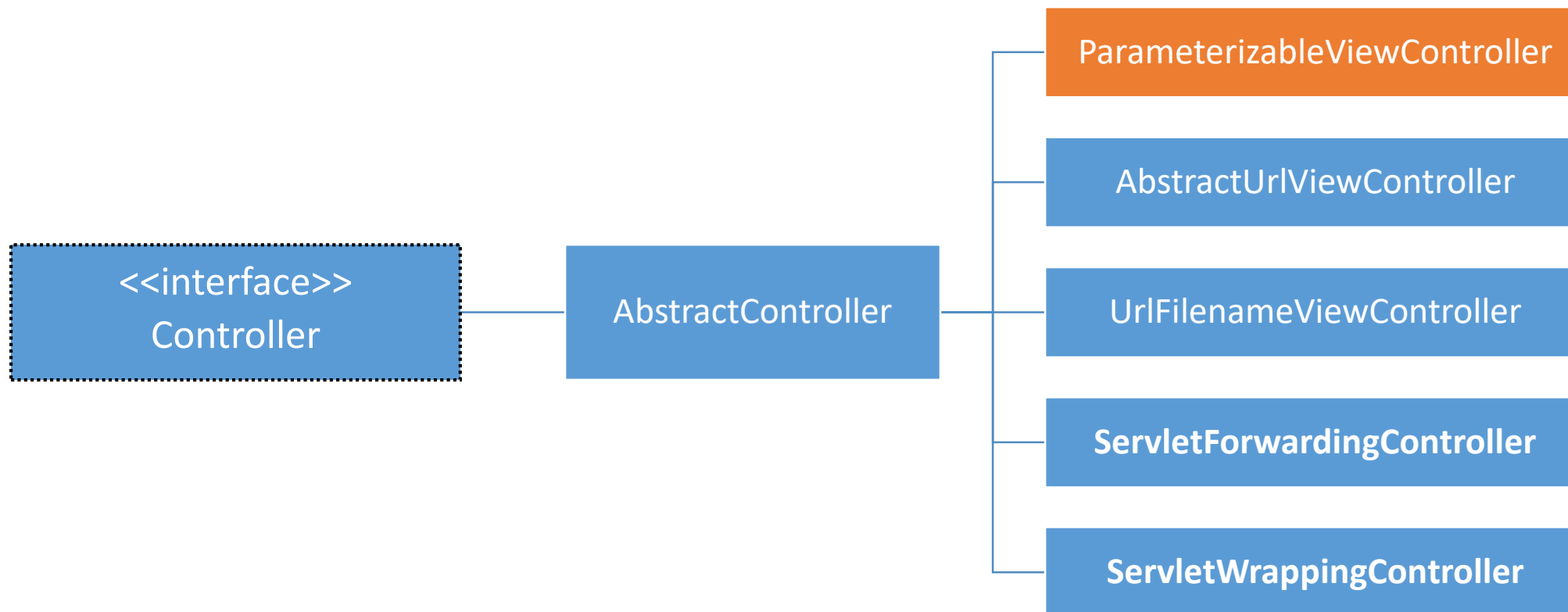


# Static Resources

- We have a challenge to deal with static resources as:
  - Based on the **request life cycle** we have to **submit our request to controller** then **controller provide the logical view name** for me.
- We will have **an overhead of defining controller** classes to just provide me with **the logical name**.
- So Spring provide us an abstracted ways to do that using:
  - ParameterizableViewController
  - AbstractUrlViewController
  - UrlFilenameViewController



# Controllers Hierarchy





# ParameterizableViewController

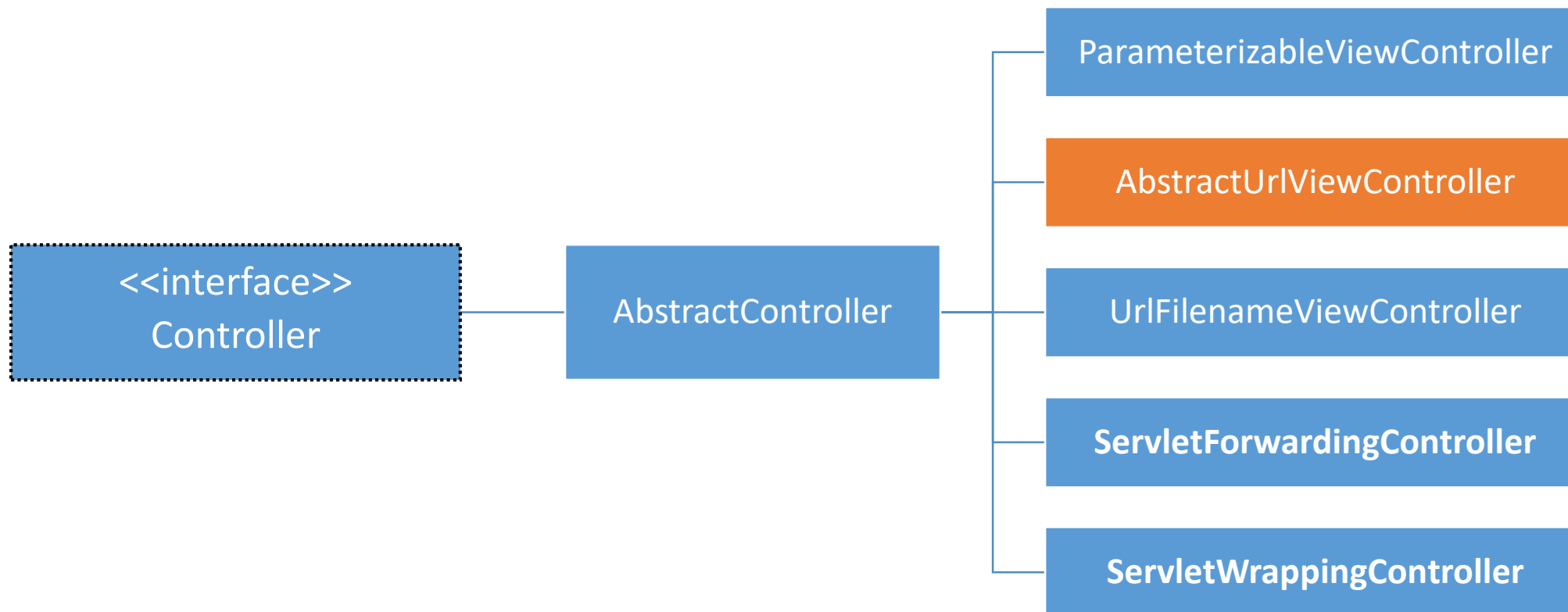
- Trivial controller that always returns
  - A pre-configured view.
  - Optionally sets the response status code.
- The view and status can be configured using the **provided configuration properties**.
- As following:

```
<!-- ParameterizableViewController -->  
<bean name="contactUsController"  
      class="org.springframework.web.servlet.mvc.ParameterizableViewController"  
      p:viewName="contactUs"  
      p:statusCode="ACCEPTED"/>
```





# Controllers Hierarchy





# AbstractUrlViewController

- Abstract base class for Controllers that return a **view name** based on **the request URL**.
- Provides infrastructure for determining view names from URLs and configurable URL lookup.
- We can use it by define class that extend AbstractUrlViewController and override
  - **protected String getViewNameForRequest(HttpServletRequest httpServletRequest);** Method
  - That takes your request and provide the logical name of view.
- Also don't forget to create your bean definition in application context.

```
<!-- AbstractUrlViewController -->  
<bean name="helloController2"  
      class="com.jediver.spring.controller.HelloWorldController2" />
```



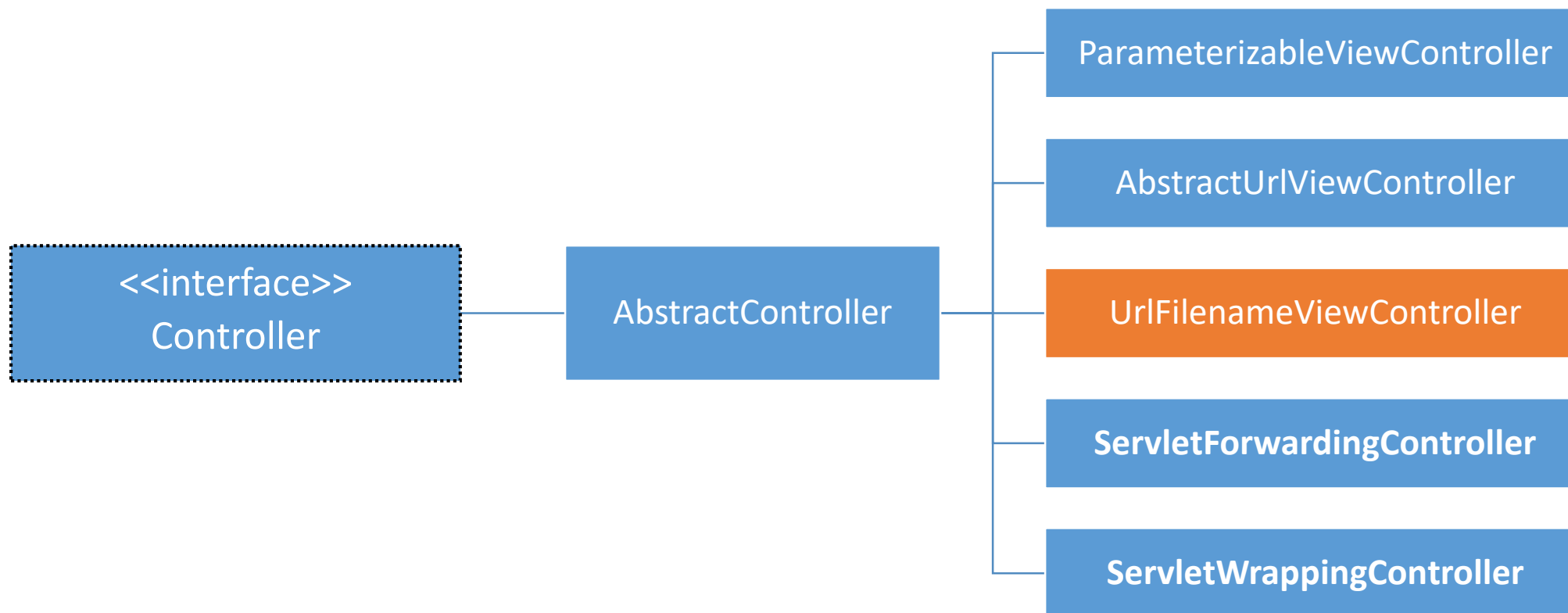
# AbstractUrlViewController (Ex.)

```
public class HelloWorldController2 extends AbstractUrlViewController {

    @Override
    protected String getViewNameForRequest(HttpServletRequest httpServletRequest) {
        String viewName = null;
        //      String requestPath = httpServletRequest.getRequestURI()
        //          .substring(httpServletRequest.getContextPath().length());
        String requestPath = new UrlPathHelper()
            .getPathWithinApplication(httpServletRequest);
        System.out.println(requestPath);
        if (requestPath.equals("/hello2.htm")) {
            viewName = "HelloWorldPage2";
        }
        return viewName;
    }
}
```



# Controllers Hierarchy





# UrlFilenameViewController

- `public class UrlFilenameViewController extends AbstractUrlViewController`
- Simple Controller implementation that transforms **the virtual path of a URL** into **a view name** and returns that view.
- Also we Can **optionally** prepend a prefix **and/or** append a suffix to build the **viewname**.
- Find some examples below:
- `"/index"`  $\rightarrow$  `"index"`
- `"/index.html"`  $\rightarrow$  `"index"`
- `"/products/view.html"`  $\rightarrow$  `"products/view"`
- `"/index.html" + prefix "pre_" + suffix "_suf"`  $\rightarrow$  `"pre_index_suf"`



# UrlFilenameViewController (Ex.)

- First We declare a bean definition from `UrlFilenameViewController`.

```
<!-- UrlFilenameViewController -->
<bean name="aboutUsController"
      class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>
```

- Then don't forget to map this controller with an url:
- **Note:** We could map the same bean definition with many mapped URLs as needed.

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="order" value="1"/>
  <property name="mappings">
    <props>
      <prop key="/contactUs.htm">aboutUsController</prop>
      <prop key="/aboutUs.htm">aboutUsController</prop>
    </props>
  </property>
</bean>
```



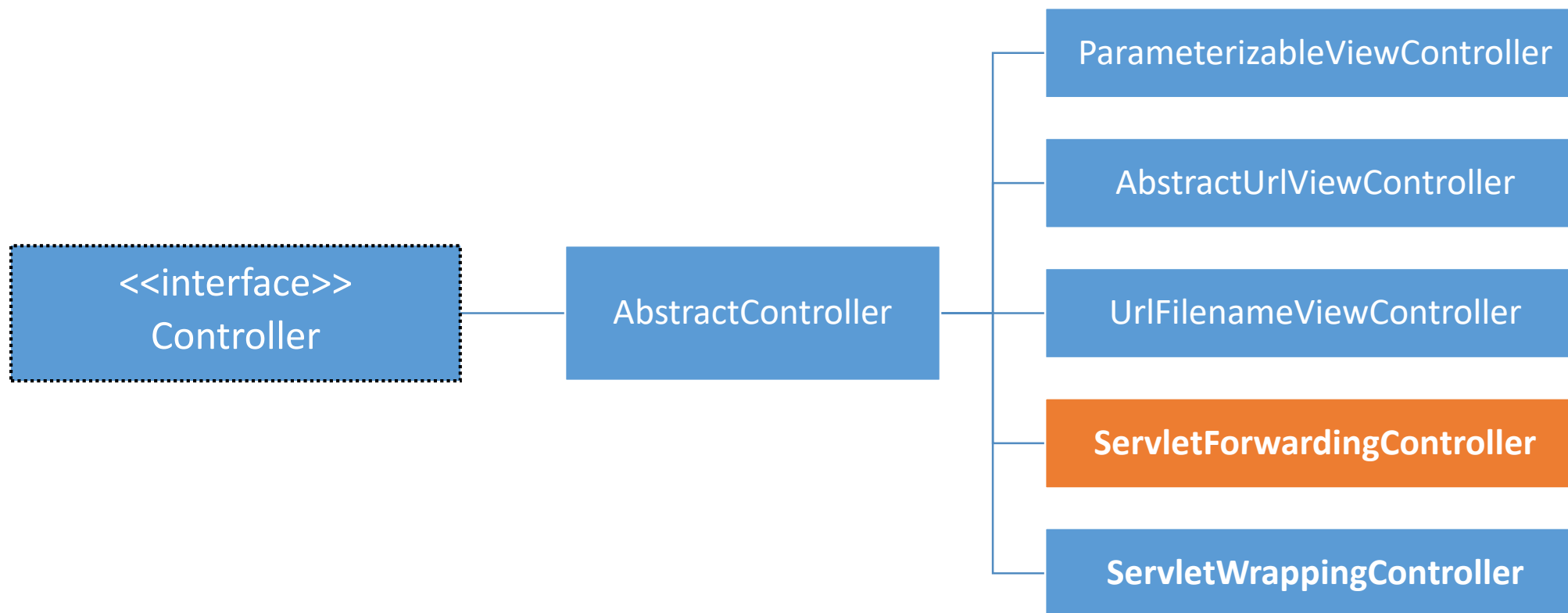
# UrlFilenameViewController (Ex.)

- Another example using Prefix:

```
<bean name="adminLinks"
      class="org.springframework.web.servlet.mvc.UrlFilenameViewController"
      p:prefix="/admin/" />
<bean name="userLinks"
      class="org.springframework.web.servlet.mvc.UrlFilenameViewController"
      p:prefix="/user/" />
      <bean id="urlMapping"
            class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="order" value="1" />
        <property name="mappings">
          <props>
            <prop key="/adminHome.htm">adminLinks</prop>
            <prop key="/adminContact.htm">adminLinks</prop>
            <prop key="/userHome.htm">userLinks</prop>
            <prop key="/userContact.htm">userLinks</prop>
          </props>
        </property>
      </bean>
```



# Controllers Hierarchy







# ServletForwardingController

- `public class ServletForwardingController` extends `AbstractController` implements `BeanNameAware`
- Spring Controller implementation that forwards to a named servlet.
  - i.e. the "servlet-name" in web.xml rather than a URL path mapping.
- A target servlet **doesn't** even **need a "servlet-mapping"** in **web.xml** in the first place: A "servlet" declaration is sufficient.
- Useful to invoke an existing servlet via Spring's dispatching infrastructure.
- For example to apply **Spring HandlerInterceptors** to its requests. This will work even in a **minimal Servlet container** that **does not support Servlet filters**.



# ServletForwardingController (Ex.)

- We need only to declare servlet instance in web.xml without any mapping (optional).

```
<servlet>
    <servlet-name>ProfileServlet</servlet-name>
    <servlet-class>com.jediver.spring.servlet.ProfileServlet</servlet-class>
</servlet>
```

- Then you define the spring controller that is linked with this servlet.

```
<bean id="myServletForwardingController"
    class="org.springframework.web.servlet.mvc.ServletForwardingController">
    <property name="servletName">
        <value>ProfileServlet</value>
    </property>
</bean>
```



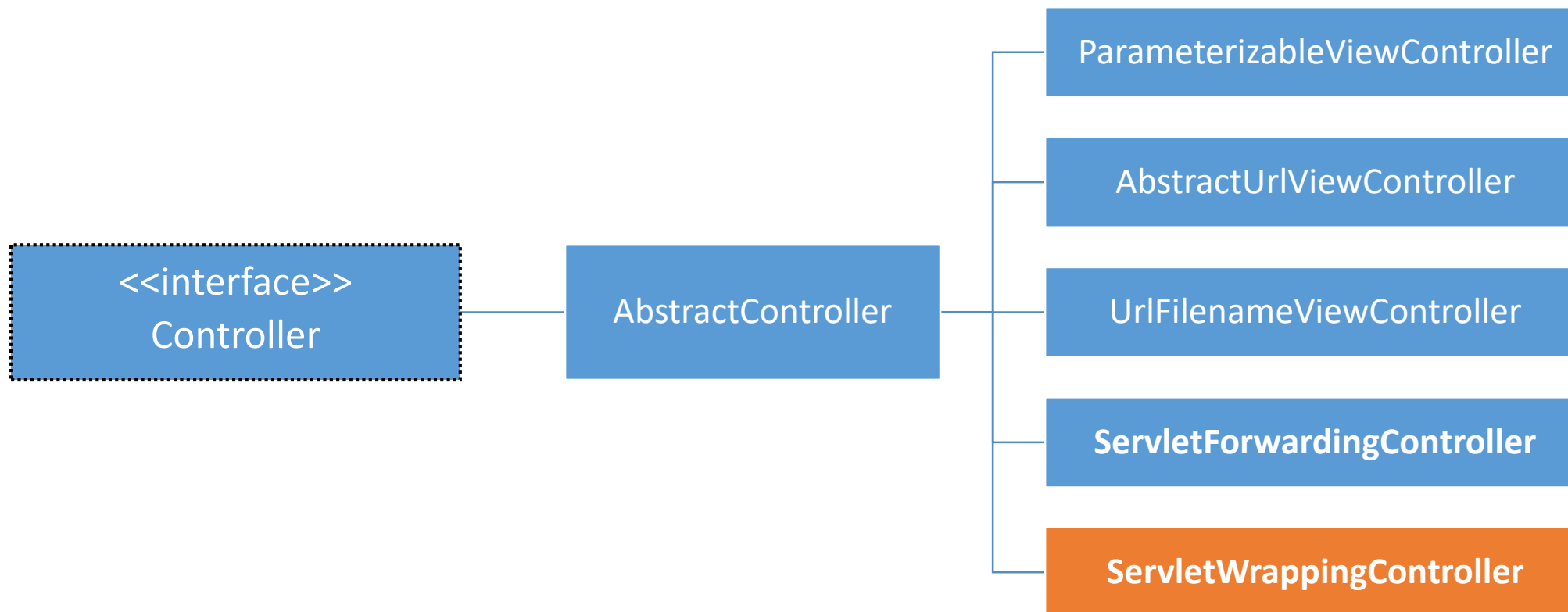
# ServletForwardingController (Ex.)

- Finally map your controller through your handler mapping.

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="order" value="1"/>
    <property name="mappings">
        <props>
            <prop key="/profileservlet.htm">myServletForwardingController</prop>
        </props>
    </property>
</bean>
```



# Controllers Hierarchy





# ServletWrappingController

- `public class ServletWrappingController` extends `AbstractController`  
implements `BeanNameAware`, `InitializingBean`, `DisposableBean`
- Spring Controller implementation that wraps a servlet instance which it manages internally.
- Such a **wrapped servlet** is **not known** outside of **this controller**.
- Its entire lifecycle is covered here (in **contrast** to `ServletForwardingController`).
- Useful to invoke an existing servlet via Spring's dispatching infrastructure, for example to apply Spring HandlerInterceptors to its requests.



# ServletWrappingController (Ex.)

- **Note:**

- For Example Struts has a special requirement in that it parses web.xml to find its servlet mapping.
- Therefore, you need to specify the DispatcherServlet's servlet name as "servletName" on this controller, so that Struts finds the DispatcherServlet's mapping (thinking that it refers to the ActionServlet).



# ServletWrappingController (Ex.)

- We need only to declare servlet instance in beans definition in spring context.

```
<bean id="myServletWrappingController"
      class="org.springframework.web.servlet.mvc.ServletWrappingController">
  <property name="servletClass">
    <value>com.jediver.spring.servlet.ProfileServlet</value>
  </property>
  <property name="servletName">
    <value>ProfileServlet2</value>
  </property>
  <property name="initParameters">
    <props>
      <prop key="name">JEDiver</prop>
    </props>
  </property>
</bean>
```



# ServletWrappingController (Ex.)

- Finally map your controller through your handler mapping.

```
<bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="order" value="1"/>
    <property name="mappings">
        <props>
            <prop key="/profileservlet2.htm">myServletWrappingController</prop>
        </props>
    </property>
</bean>
```





# Controllers Hierarchy

- We have another controllers supported in older versions:
- Removed from version 4:
  - MultiActionController
- Removed from version 3:
  - BaseCommandController
  - AbstractCommandController
  - AbstractFormController
  - SimpleFormController
  - AbstractWizardFormController



# SimpleFormController

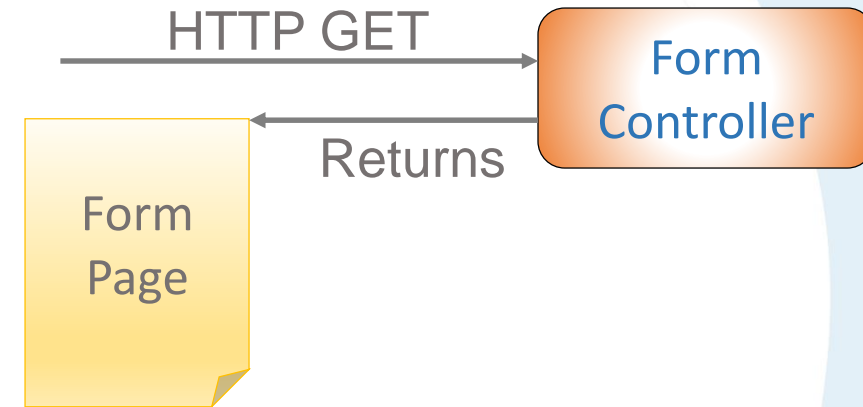
- Concrete FormController implementation that provides:
  - Configurable form.
  - Success views.
  - An **onSubmit** chain for convenient overriding.
- **Automatically resubmits** to the form view in case of validation errors, and renders the success view in case of a valid submission.



# SimpleFormController (Ex.)

- Workflow of **AbstractFormController**:

1. The controller receives a **request for a new form** (typically a GET).
2. Call to **formBackingObject()** which by default, returns an instance of the **commandClass** that has been configured either default or user defined objects.
3. Call to **initBinder()** which allows you to register custom editors for certain fields (often properties of non-primitive or non-String types) of the command class.
  - This will render appropriate Strings for those property values, e.g. locale-specific date strings.





# SimpleFormController (Ex.)

- Workflow:
  4. Only if `bindOnNewForm` is set to `true`, then `ServletRequestDataBinder` gets applied to populate the new form object with initial request parameters and the `onBindOnNewForm(HttpServletRequest, Object, BindException)` callback method is called.
  5. Call to `showForm()` to return a View that should be rendered (your form).
  6. The `showForm()` implementation will call `referenceData()`, which you can implement to provide any relevant reference data you might need when editing a form.
  7. Model gets exposed and view gets rendered, to let the user fill in the form.



# SimpleFormController (Ex.)

- Workflow:

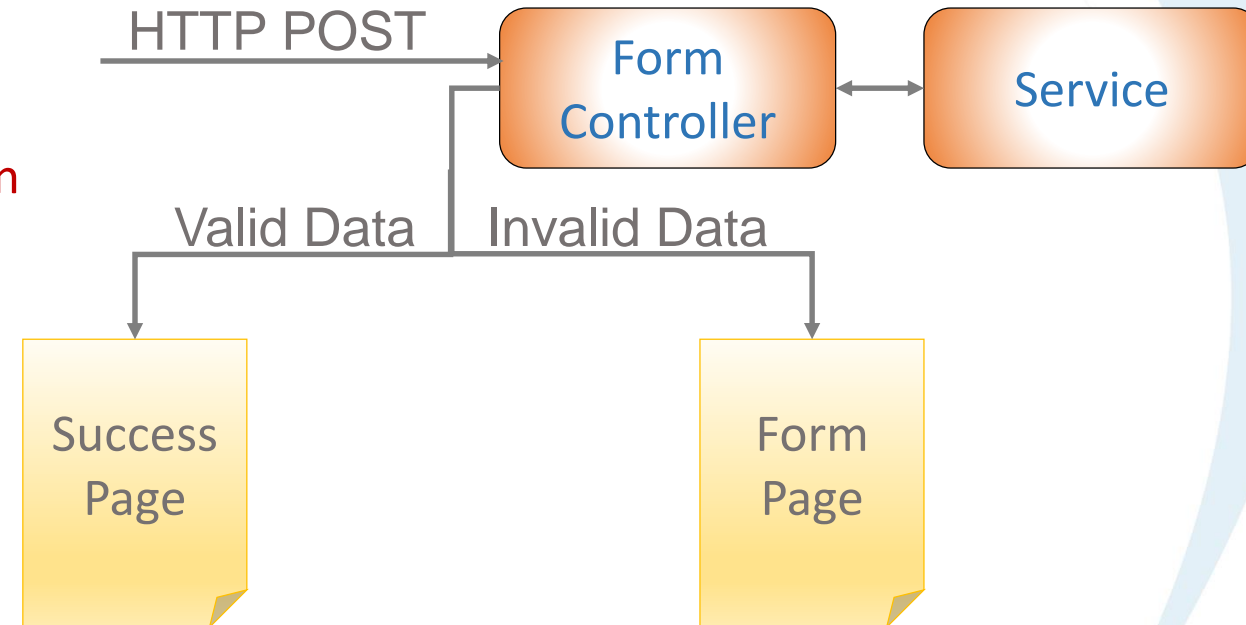
8. The controller receives a **form submission** (typically a **POST**).

- To use a different way of detecting a form submission, override **isFormSubmission()**

9. If **sessionForm** is

- Not configured, **formBackingObject()** is called to retrieve a form object.
- Configured, the controller tries to find the command object which is already bound in the session.

10. The **ServletRequestDataBinder** gets applied to populate the form object with current request parameters.





# SimpleFormController (Ex.)

- Workflow:
  11. Call to `onBind(HttpServletRequest, Object, Errors)` which allows you to do custom processing after binding but **before validation**.
    - e.g. to manually bind request parameters to bean properties, to be seen by the Validator.
  12. If `validateOnBinding` is **set**, a registered Validator will be invoked.
    - The Validator will check the form object properties, and register corresponding errors via the given Errors object.
  13. Call to `onBindAndValidate()` which allows you to do custom processing after binding and validation.
  14. Call `processFormSubmission()` to process the submission, with or without binding errors.



# SimpleFormController (Ex.)

- Workflow of **SimpleFormController**:
  1. Call to **processFormSubmission** which inspects the Errors object to see if any errors have occurred during binding and validation.
  2. **If errors occurred**, the controller will return the configured **formView**, showing the form again.
  3. **If no errors occurred**, the controller will call **onSubmit** using all parameters, which in case of the default implementation delegates to **onSubmit** with just the command object.
    - The **default implementation** of the latter method will return the configured **successView**.



# SimpleFormController (Ex.)

- Exposed configuration properties from **AbstractFormController**:

name	default	description
bindOnNewForm	false	<ul style="list-style-type: none"><li>Indicates whether to bind servlet request parameters when creating a new form. Otherwise, the parameters will only be bound on form submission.</li></ul>
sessionForm	false	<ul style="list-style-type: none"><li>Indicates whether the form object should be kept in the session when a user asks for a new form. This allows you e.g. to retrieve an object from the database, let the user edit it, and then persist it again.</li><li>Otherwise, a new command object will be created for each request (even when showing the form again after validation errors).</li></ul>





# SimpleFormController (Ex.)

- Exposed configuration properties from **SimpleFormController**:

name	default	description
formView	<i>null</i>	<ul style="list-style-type: none"><li>Indicates what view to use when the user asks for a new form</li><li>or when validation errors have occurred on form submission.</li></ul>
successView	<i>null</i>	<ul style="list-style-type: none"><li>Indicates what view to use when successful form submissions have occurred. Such a success view could e.g. display a submission summary.</li></ul>



# SimpleFormController Example

1. We declare our controller class by extending **SimpleFormController**.
- Also we define the constructor to set the commandClass and commandName to use it for rendering the gui and form submit binding

```
public class UserController extends SimpleFormController {  
  
    public UserController() {  
        setCommandClass(User.class);  
        setCommandName("user");  
    }  
}
```

- You can Also wire with any other classes as you want.

```
private UserService userService;  
  
public void setUserService(UserService userService) {  
    this.userService = userService;  
}
```



# SimpleFormController Example (Ex.)

- And override `onSubmit()` method to define our code.

```
@Override
protected ModelAndView onSubmit(Object command,
    BindException bindException) throws Exception {
    User user = (User) command;
    userService.addUser(user);
    ModelAndView mv = new ModelAndView(getSuccessView());
    mv.addObject("user", command);
    return mv;
}
```

- The HandlerAdapter will call `onSubmit()` and follow the workflow of the form controller.



# SimpleFormController Example (Ex.)

2. Then we define bean definition for this controller in spring context:

```
<bean id="userController" class="com.jediver.spring.controller.UserController">
    <property name="formView" value="addUser" />
    <property name="successView" value="userAdded" />
    <property name="userService" ref="userService" />
</bean>
```

- In this definition we define:
  - formView the view logical name which will go in case of GET request and form submission with errors.
  - successView the view logical name which will go in case of success after POST request.
  - userService As example you can inject any other bean.



# SimpleFormController Example (Ex.)

3. Then map this controller to a URL:

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="order" value="1"/>
    <property name="mappings">
      <props>
        <prop key="/register.htm">UserController</prop>
      </props>
    </property>
  </bean>
```

- In this Mapping we Have two URLs:
  - **/register.htm** with **GET** Method: that represent user request instance to fill the form.
  - **/register.htm** with **POST** Method: that represent user form submission.



# Form Validation

- Spring MVC supports validation through annotation based or custom implementation of `org.springframework.validation.Validator`
- A validator for application-specific objects.
- This interface is totally divorced from any infrastructure or context; that is to say it is not coupled to validating only objects in the **web tier**, the **data-access tier**, or the **whatever-tier**.
- All you need to validate your form fields is to make your own class that implements **Validator**.
  - Override `validate()` method to make your custom validation as you need.
  - Override `supports()` method to determine on which class to apply this validator.
- Spring MVC also offers a utility class called **ValidationUtils** which helps you in validation through static methods.



# Form Validation (Ex.)

```
public class UserValidator implements Validator {  
  
    @Override  
    public void validate(Object object, Errors errors) {  
        User user = (User) object;  
        if (user.getSalary() < 1200) {  
            errors.rejectValue("salary", "invalid.salary", "Invalid Salary");  
        }  
        ValidationUtils.rejectIfEmpty(errors, "name", "required.name",  
            "Name is required.");  
    }  
  
    @Override  
    public boolean supports(Class<?> clazz) {  
        return clazz.equals(User.class);  
    }  
}
```



# Form Validation (Ex.)

- `ValidationUtils.rejectIfEmpty(errors, "name", "required.name", "Name is required.");`
- **errors:**
  - the name of the object of Errors that contains any raised error from form binding.
- **name:**
  - is the name of the variable (field) we are validating in the command object
- **required.name:**
  - error code if you want to use properties file to retrieve the error message
- **"Name is required.":**
  - the default error message that will appear to the user





# Form Validation

- Then we define the definition of this validator in spring context by:

```
<bean id="userValidator" class="com.jediver.spring.validator.UserValidator"/>
```

- After we create our custom validator we need to use it with our controller by wiring it as follows:

```
<bean id="userController" class="com.jediver.spring.controller.UserController">  
    <property name="formView" value="addUser" />  
    <property name="successView" value="userAdded" />  
    <property name="userService" ref="userService" />  
    <property name="validator" ref="userValidator" />  
</bean>
```



# Exception Handling

- Spring MVC provides support to handle exceptions using **SimpleMappingExceptionHandler** which allows mapping exception class names to view names.
- **SimpleMappingExceptionHandler** handles any **java.lang.Exceptions** thrown from Spring MVC controllers.

```
<bean id="exceptionResolver"  
    class="org.springframework.web.servlet.handler.SimpleMappingExceptionHandler">  
    <property name="exceptionMappings">  
        <props>  
            <prop key="java.lang.Exception">  
                errorView  
            </prop>  
        </props>  
    </property>  
</bean>
```

## Lesson 6

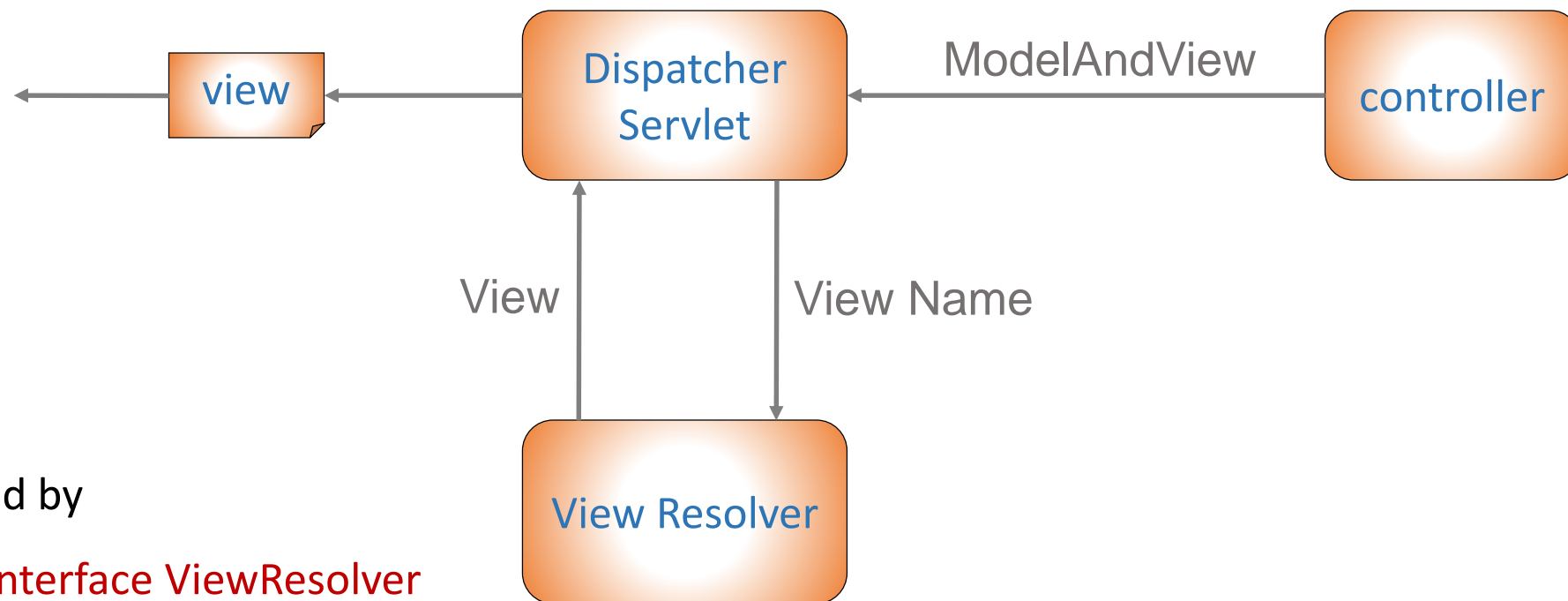
# View Resolvers





# View Resolvers

- Are used to know how the logical view name is used to determine which view will render the results to the user.



- Provided by  
**public interface ViewResolver**

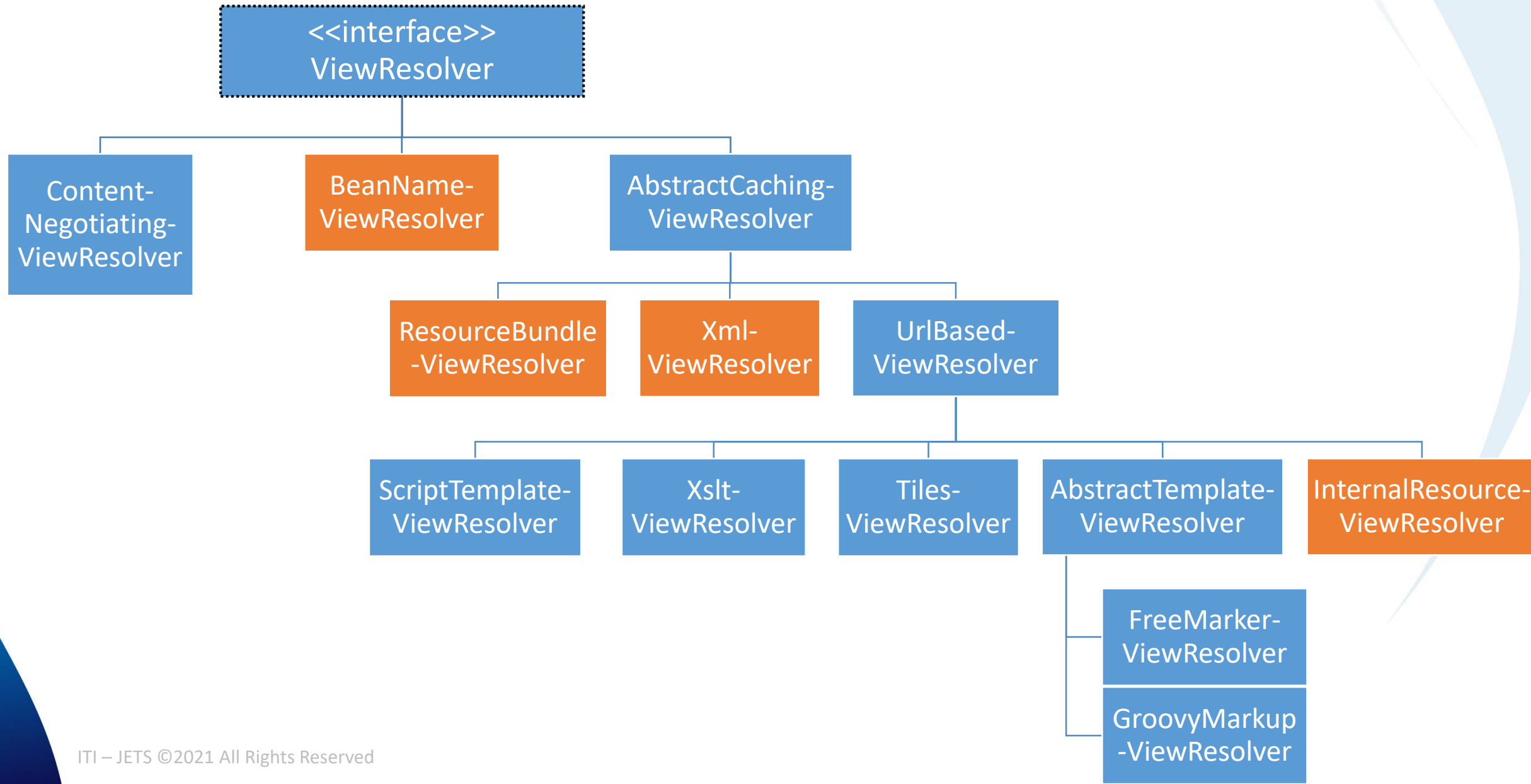


# Interface ViewResolver

- Interface to be implemented by objects that can resolve views by name.
- View state doesn't change during the running of the application, so implementations are free to cache views.
- Implementations are encouraged to support internationalization, i.e. localized view resolution.



# Types of View Resolvers





# Types of View Resolvers

name	description
InternalResourceViewResolver	<ul style="list-style-type: none"><li>• Taking the logical view name returned in a ModelAndView object and surrounding it with a prefix and a suffix to arrive at the path of a View within the web application.</li></ul>
BeanNameViewResolver	<ul style="list-style-type: none"><li>• Looks up implementations of the View interface as beans in the Spring context, assuming that the bean name is the logical view name.</li></ul>
ResourceBundleViewResolver	<ul style="list-style-type: none"><li>• Uses a resource bundle (e.g., a properties file) that maps logical view names to implementations of the View interface</li></ul>
XmlViewResolver	<ul style="list-style-type: none"><li>• Resolves View beans from an XML file that is defined separately from the application context definition files</li></ul>



# InternalResourceViewResolver

- Convenient subclass of `UrlBasedViewResolver` that supports `InternalResourceView`
  - i.e. Servlets, JSPs and subclasses such as `JstlView`.
- The view class for all views generated by this resolver can be specified via `UrlBasedViewResolver.setViewClass(java.lang.Class<?>)`.
- The default is `InternalResourceView`, or `JstlView` if the JSTL API is present.
- It's good practice to put JSP files that just serve as views under WEB-INF, to hide them from direct access (e.g. via a manually entered URL).
- Only controllers will be able to access them then.





# InternalResourceViewResolver (Ex.)

- InternalResourceViewResolver will produce resource name based on:
- Prefix + Logical View Name + Suffix
- The default values for Prefix is "" and for suffix ""

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



# BeanNameViewResolver

- A simple implementation of ViewResolver that interprets a view name as a bean name in the current application context.
- This resolver can be handy for small applications, keeping all definitions ranging from controllers to views in the same place.
- For **larger applications**, **XmlViewResolver** will be the better choice, as it separates the XML view bean definitions into a dedicated views file.
- Note:
- **Neither this ViewResolver nor XmlViewResolver supports internationalization.** Consider **ResourceBundleViewResolver** if you need to apply different view resources per locale.



# BeanNameViewResolver (Ex.)

- Note:
- This **ViewResolver** implements the Ordered interface in order to allow for flexible participation in **ViewResolver** chaining.
- First you define the **BeanNameViewResolver** in your application context.

```
<bean id="viewResolver"  
      class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
```

- Then define your custom view in xml

```
<bean id="customView"  
      class="com.jediver.spring.view.MyCustomView"/>
```

- The result is the logical view name which will be referred from controller is "**customView**".



# XmlViewResolver

- A **ViewResolver** implementation that uses bean definitions in a dedicated XML file for view definitions, specified by resource location.
- The file will typically be located in the WEB-INF directory; the default is `"/WEB-INF/views.xml"`.
- This **ViewResolver** does not support internationalization at the level of its definition resources.
- Consider **ResourceBundleViewResolver** if you need to apply different view resources per locale.



# XmlViewResolver (Ex.)

- Inside views.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="customView"
        class="com.jediver.spring.view.MyCustomView"/>
</beans>
```

- And define your XmlViewResolver in your spring context

```
<bean id="viewResolver2"
      class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="location">
    <value>/WEB-INF/spring/views.xml</value>
  </property>
</bean>
```



# ResourceBundleViewResolver

- A **ViewResolver** implementation that uses bean definitions in a **ResourceBundle**, specified by the bundle basename.
- The bundle is typically defined in a properties file, located in the classpath. The default bundle basename is "views".
- This **ViewResolver** supports localized view definitions, using the default support of **PropertyResourceBundle**.
  - For example, the basename "views" will be resolved as class path resources "views\_de\_AT.properties", "views\_de.properties", "views.properties" - for a given Locale "de\_AT".



# ResourceBundleViewResolver (Ex.)

- Inside viewsProperties.properties

```
customView.class=com.jediver.spring.view.MyCustomView
```

- And define your XmlViewResolver in your spring context

```
<bean id="viewResolver"  
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">  
  <property name="basename" value="classpath:viewsProperties" />  
</bean>
```



# Multiple View Resolvers

- Fortunately, you aren't limited for choosing only one view resolver for your application.
- To use multiple view resolvers, simply declare all the view resolvers you will need in the spring context configuration file.
- Spring determine which resolver has priority over the others by the value of the order property in each view resolver.
- **InternalResourceViewResolver** always needs to be last, as it will attempt to resolve any view name, no matter whether the underlying resource actually.





# Multiple View Resolvers (Ex.)

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.BeanNameViewResolver">
    <property name="order" value="1"/>
</bean>
<bean id="viewResolver1"
      class="org.springframework.web.servlet.view.XmlViewResolver">
    <property name="location">
      <value>/WEB-INF/spring/views.xml</value>
    </property>
    <property name="order" value="2"/>
</bean>
<bean id="viewResolver2"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
    <property name="prefix">
      <value>/WEB-INF/pages/</value>
    </property>
    <property name="suffix">
      <value>.jsp</value>
    </property>
    <property name="order" value="3"/>
</bean>
```



# Lesson 7

## Views





# What is a View?

- The View class is responsible for rendering the response from a Controller.
- Spring MVC supports many different view rendering technologies, including:
  - JSP and JSTL
  - Velocity (<http://velocity.apache.org/>)
  - FreeMarker (<https://freemarker.apache.org/>)
  - PDF
  - Excel
  - JasperReports (an open-source reporting tool,  
<https://community.jaspersoft.com/project/jasperreports-library>)



# Using JSP as a View

- Binding form data
- Displaying errors
- Externalizing text
- Internationalization



# Using JSP as a View

- Binding form data
- Displaying errors
- Externalizing text
- Internationalization



# Binding form data

- Binding form data is to tell Spring which properties of the command object to populate with the form data when the form is submitted.
- Before Spring 2.0 we used the `<spring:bind>` Tag for binding form data;
- After Spring 2.0 there is another Tag library specialized in handling the form data binding, it is the Form Tag library.

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```



# Binding form data (Ex.)

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
  <body>
    <form:form method="POST" action="register.htm" commandName="user">
      <b> User Name:</b>
      <form:input path="name" />
      <br/>
      <b> Salary:</b>
      <form:input path="salary" />
      <br/>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```



# Using JSP as a View

- Binding form data
- **Displaying errors**
- Externalizing text
- Internationalization





# Displaying binding errors

- When a field's value is rejected during validation, an error message code is associated with the field in the Errors object.
- The `<form:errors>` tag looks for any error message codes associated with the field (which is specified with the path attribute) and then tries to resolve those messages from an **external properties file**.
- We could put the error messages in the same messages file as the other externalized messages or put them in a separate properties file.
- We could also catch all errors by `<form:errors path="*" />`



# Displaying binding errors (Ex.)

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<html>
  <body>
    <form:form method="POST" action="register.htm" commandName="user">
      <b> User Name:</b>
      <form:input path="name" />
      <form:errors path="name"/>
      <br/>
      <b> Salary:</b>
      <form:input path="salary" />
      <form:errors path="salary"/>
      <br/>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```



# Using JSP as a View

- Binding form data
- Displaying errors
- Externalizing text
- Internationalization



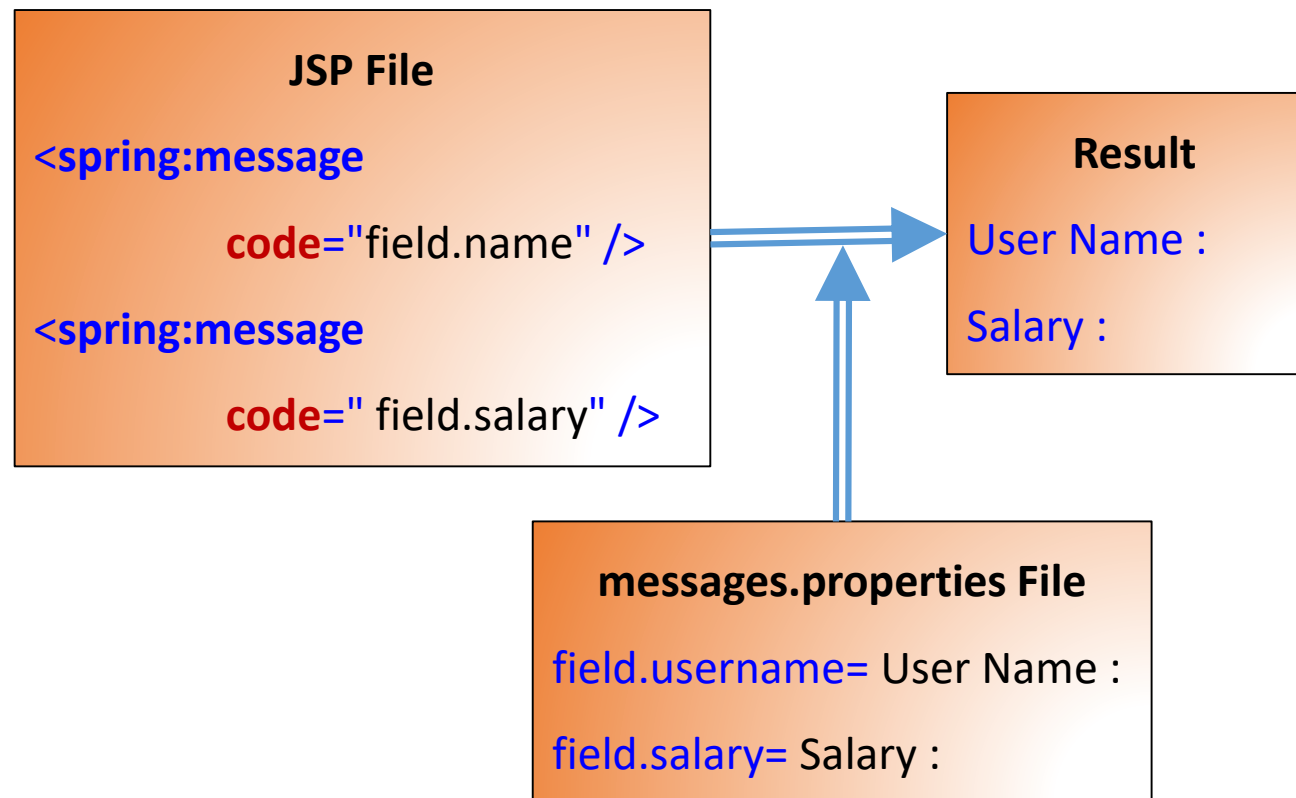
# Externalizing text

- Keep Static text consistence among different pages.
- Easy to change.
- Internationalization support.
- You can use the externalized messages by passing the message code to `<spring:message>`'s code attribute.
- To use the `<spring:message>` tag, you must import the spring tag library as follows:

```
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>
```



# Externalizing text (Ex.)





# Externalizing text (Ex.)

- Spring's **ResourceBundleMessageSource** resolves message codes to actual message values.
- It has to be registered in the spring context for example as follows:

```
<bean id="messageSource"  
      class="org.springframework.context.support.ResourceBundleMessageSource">  
    <property name="basename" value="messages" />  
</bean>
```

- It's important to name the bean with "**messageSource**" because that's the name Spring will use to look for a message source.
- The basename property determine the name of the properties file the messages is externalized in.
- In this case **/WEB-INF/classes/messages.properties**



# Externalizing text (Ex.)

- If we want to put the error messages in separate properties file we have to change some settings in the messageSource declaration to make it accept multiple files

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value> messages </value>
      <value> errors </value>
    </list>
  </property>
</bean>
```



# Externalizing text (Ex.)

```
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
  <body>
    <form:form method="POST" action="register.htm" commandName="user">
      <b><spring:message code="field.username"/></b>
      <form:input path="name" />
      <form:errors path="name"/>
      <br/>
      <b><spring:message code="field.salary"/></b>
      <form:input path="salary" />
      <form:errors path="salary"/>
      <br/>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```





# Using JSP as a View

- Binding form data
- Displaying errors
- Externalizing text
- **Internationalization**



# Internationalization

- One of the benefits of message externalization is that it facilitates the internationalization.
- From the previous example if you want to add support for Arabic language all you need is to write another properties file with the name "**messages\_ar\_EG.properties**" and save it in the path "/WEB-INF/classes/" you don't even want to add more setting

## Messages\_ar\_EG.properties File

field.username= اسم المستخدم

field.salary= المرتب



# Define Custom view using View Interface

- MVC View for a web interaction.
- Implementations are responsible for rendering content, and exposing the model.
- A single view exposes multiple model attributes.
- This class and the MVC approach associated with it is discussed in details in Chapter 12 of Expert One-On-One J2EE Design and Development by Rod Johnson (Wrox, 2002).
- Views should be beans. They are likely to be instantiated as beans by a ViewResolver. As this interface is stateless, view implementations **should be thread-safe**.



# Define Custom view using View Interface (Ex.)

- First we define our class that:
  - Implement View interface
  - Override `public String getContentType();`
  - To provide the response content type you want to produce.
  - Override `public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception;`
  - Model Data, request and response sent to your custom view to perform your business code.



# Define Custom view using View Interface (Ex.)

```
public class MyCustomView implements View {  
  
    @Override  
    public String getContentType() {  
        return "text/html";  
    }  
  
    @Override  
    public void render(Map<String, ?> model, HttpServletRequest request,  
        HttpServletResponse response) throws Exception {  
        response.setContentType(getContentType());  
        PrintWriter writer = response.getWriter();  
        writer.println("This is my custom dummy view.<br/>");  
        writer.println("<h3>Model attributes</h3>");  
        for (Map.Entry<String, ?> entry : model.entrySet()) {  
            writer.printf("%s = %s<br/>", entry.getKey(), entry.getValue());  
        }  
    }  
}
```



# Define Custom view using AbstractPdfView

- Spring Also provide some implementations for view interface by default in case of you want to use:
- For Example spring provide **AbstractPdfView** as custom view to produce pdf by integrate with third party API called "itext".
- So first import the dependency of this library:

```
<dependency>  
    <groupId>com.lowagie</groupId>  
    <artifactId>itext</artifactId>  
    <version>2.1.7</version>  
</dependency>
```



# Define Custom view using **AbstractPdfView** (Ex.)

- First we define our class that:
  - Implement **org.springframework.web.servlet.view.document.AbstractPdfView**
  - Override **protected void buildPdfDocument(Map<String, Object> model, Document document, PdfWriter writer, HttpServletRequest hsr, HttpServletResponse hsr1) throws Exception;**
  - Model Data, reference to the output pdf document file, pdf writer instance to use (optional), request and response sent to your custom view to perform your business code.



# Define Custom view using AbstractPdfView (Ex.)

```
public class PdfCustomView extends AbstractPdfView {

    @Override
    protected void buildPdfDocument(Map<String, Object> model, Document document,
        PdfWriter writer, HttpServletRequest hsr,
        HttpServletResponse hsr1) throws Exception {
        List<User> users = (List<User>) model.get("users");
        Table table = new Table(4);
        table.addCell("Counter #");
        table.addCell("User Id");
        table.addCell("User Name");
        table.addCell("user Salary");
        for (int i = 0; i < users.size(); i++) {
            User user = users.get(i);
            table.addCell("" + i + 1);
            table.addCell("" + user.getId());
            table.addCell(user.getName());
            table.addCell("" + user.getSalary());
        }
        document.add(table);
    }
}
```





## Lesson 8

# Using Annotations





# Enabling URL Mapping by Annotation

- The target of using annotation is to simplify the lifecycle configuration for request.
- Spring Configuration **under version 5**:
- The default URLHandlerMapping that can read @RequestMapping and other Annotation is `org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping`
- So to enable spring context to read the annotation configuration of MVC e.g. `@RequestMapping`, you must declare the `DefaultAnnotationHandlerMapping` to enable you to use spring mvc annotations which is deprecated and replaced by `RequestMappingHandlerMapping`.
- You can do so explicitly as follows:

```
<bean  
    class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping" />
```



# Enabling URL Mapping by Annotation (Ex.)

- Or Spring MVC provide namespace for MVC so you can import it by **springmvc namespace**:

```
xmlns:mvc="http://www.springframework.org/schema/mvc"  
xsi:schemaLocation="http://www.springframework.org/schema/beans  
http://www.springframework.org/schema/beans/spring-beans.xsd  
http://www.springframework.org/schema/mvc  
http://www.springframework.org/schema/mvc/spring-mvc.xsd"
```

- **<annotation-driven>** Tag by using this tag it implicitly create bean of **DefaultAnnotationHandlerMapping**.

```
<mvc:annotation-driven />
```



# Enabling URL Mapping by Annotation (Ex.)

- Spring Configuration **starting from version 5**:
- The **DefaultAnnotationHandlerMapping** has been removed.
- The default URLHandlerMapping that can read @RequestMapping and other Annotation is **org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping**
- So to enable spring context to read the annotation configuration of MVC e.g. **@RequestMapping**, you must declare the **RequestMappingHandlerMapping** to enable you to use spring mvc annotations.
- You can do so explicitly as follows:

```
<bean  
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping" />
```



# Enabling URL Mapping by Annotation (Ex.)

- If you use the `<annotation-driven>` Tag by using this tag it doesn't create bean of `RequestMappingHandlerMapping`.

```
<mvc:annotation-driven />
```

- **But still** we need to **declare it** because this tag also define the `HandlerAdapter` that used to handle the request.



# Enabling URL Mapping by Annotation (Ex.)

- Or if you make your configuration annotation based using @Configuration so you use annotation called "**@EnableWebMvc**" which is enable spring MVC annotations and also define bean of **RequestMappingHandlerMapping**.
- We use it as the following example:

```
@Configuration
@EnableWebMvc
public class ApplicationConfiguration {
    ...
}
```



# Define Controllers

- To define a controller for static resource as you defined in **ParameterizableViewController**, We can use **<mvc:view-controller>** tag to register controller that selects a view for rendering.

```
<mvc:view-controller path="/contactUs.htm" view-name="contactUs" />
```

- In this case, when **"/contactUs.htm"** is requested, the controller will respond with the logical name of view **"contactUs"**.
- The actual view is defined as normal by **ViewResolver** as we discussed in lessons before.
- E.g. if we use **InternalResourceViewResolver**, so the physical name is **prefix + logicalName + suffix**.



# Define Controllers (Ex.)

- To define a controller for your dynamic resource by define normal class and annotate it with Spring MVC annotations.

1. Either by xml bean definition:

- The Controller Class:

```
@RequestMapping("/register.htm")  
public class UserController {  
    ...  
}
```

- The XML Bean Definition:

```
<bean  
    class="com.jediver.spring.controller.UserController" />
```





# Define Controllers (Ex.)

2. Either by annotation bean definition:

- The Controller Class:

```
@Controller
@RequestMapping("/register.htm")
public class UserController {
    ...
}
```

- The XML Bean Definition:

```
<context:component-scan base-package="com.jediver.spring"/>
```



# Mapping Requests With @RequestMapping

- The @RequestMapping annotation is used to map URLs Such as `"/displayall.htm"` on:
  - Either on EntireClass

```
@Controller
@RequestMapping("/displayall.htm")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(method = RequestMethod.GET)
    public String displayAllUsers(Model data) {
        List<User> users = userService.getAllUsers();
        data.addAttribute("users", users);
        return "displayAll";
    }
}
```



# Mapping Requests With @RequestMapping

- The @RequestMapping annotation is used to map URLs Such as `"/displayall.htm"` on:
  - Or on a particular handler method.

```
@Controller
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/displayall.htm",
                    method = RequestMethod.GET)
    public String displayAllUsers(Model data) {
        List<User> users = userService.getAllUsers();
        data.addAttribute("users", users);
        return "displayAll";
    }
}
```



# @RequestMapping

- Also you can mix between using **@RequestMapping on Both levels**.
- The **@RequestMapping** on the class level in this type represent a **relative path** which indicates that all handling methods on this controller are relative to this path.



# @RequestMapping

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/getUsers", method = RequestMethod.GET)
    public String displayAllUsers(Model data) {
        List<User> users = userService.getAllUsers();
        data.addAttribute("users", users);
        return "displayAll";
    }

    @RequestMapping(value = "/getUser", method = RequestMethod.GET)
    public String displayUser(@RequestParam("userId") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }
}
```

Serve requests by URL:  
/admin/users/getUsers  
.htm

Serve requests by URL:  
/admin/users/getUser.  
htm



# @RequestMapping

- The previous @RequestMapping example methods serve requests with GET Method.
- If you didn't specify method attribute for annotation so they supports all your requests regardless method type.
- You can force your controller to serve only requests with specific method type (i.e GET or POST)



# @RequestMapping

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/getUsers")
    public String displayAllUsers(Model data) {
        List<User> users = userService.getAllUsers();
        data.addAttribute("users", users);
        return "displayAll";
    }

    @RequestMapping(value = "/getUser")
    public String displayUser(@RequestParam("userId") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }
}
```

Serve requests with  
any method by URL:  
/admin/users/getUsers  
.htm

Serve requests with  
any method by URL:  
/admin/users/getUser.  
htm



# Advanced @RequestMapping Options (Params)

- You can narrow request matching through **request parameter conditions**.
- For Example:
  - `params= "myParam"` ====> Check for the presence of specific parameter.
  - `params= "!myParam"` ====> Check for the absence of specific parameter.
  - `params= "myParam=myValue"` ====> Check for specific parameter value.





# Advanced @RequestMapping Options (Params) (Example)

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/getUser", method = RequestMethod.GET, params = "userId")
    public String displayUser(@RequestParam("userId") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }

    @RequestMapping(value = "/getUser", method = RequestMethod.GET, params = "!userId")
    public String displayUser2(@RequestParam(name = "userId", defaultValue = "1") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }
}
```

Serve requests by URL:  
/admin/users/getUser.  
htm?userId=2

Serve requests by  
URL:  
/admin/users/getUser  
.htm



# Advanced @RequestMapping Options (Headers)

- You can narrow request matching through **request header conditions**.
- For Example:
  - `headers= "myHeader"` ====> Check for the presence of specific header.
  - `headers= "!myHeader"` ====> Check for the absence of specific header.
  - `headers= "headerName=myValue"` ====> Check for specific header value.



# Advanced @RequestMapping Options (Headers) (Example)

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/getUser", method = RequestMethod.GET, headers = "userId")
    public String displayUser(@RequestHeader("userId") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }

    @RequestMapping(value = "/getUser", method = RequestMethod.GET, headers = "!userId")
    public String displayUser2(@RequestHeader(name = "userId", defaultValue = "1") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }
}
```

Serve requests with  
header called userId by  
URL:  
/admin/users/getUser.  
htm

Serve requests  
without header called  
userId by URL:  
/admin/users/getUser  
.htm



# URI Template Patterns

- A URI Template is a URI-like string, containing one or more variable names.
- When you substitute values for these variables, the template becomes a URI
- For example,
  - The URI Template
    - <http://www.example.com/admin/users/{userId}> contains the variable `userId`.
  - Assigning the value `10` to the variable yields
    - To be accessed with <http://www.example.com/admin/users/10>.
- In Spring MVC you can use the `@PathVariable` annotation on a method argument to bind it to the value of a URI template variable.
- A method can have any number of `@PathVariable` annotations




# URI Template Patterns (Example)

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value =("/{userId}", method = RequestMethod.GET)
    public String displayUser(@PathVariable("userId") Integer id, Model data) {
        User user = userService.getUser(id);
        data.addAttribute("users", user);
        return "displayAll";
    }
}
```



Serve requests by URL:  
/admin/users/2.htm



# Defining @RequestMapping handler methods

- An **@RequestMapping** handler method can have a **very flexible signatures**.
- There are supported:
  - Method arguments.
  - Return values.
- You can define as much as you want in same controller class.



# Supported Handler Method Argument Types

- `HttpServletRequest` / `HttpServletResponse`
  - Request or response objects (Servlet API)
- `HttpSession`
  - Session object (Servlet API): of type `HttpSession`
- `java.util.Locale`
  - The current request locale
- `java.io.InputStream` / `java.io.Reader`
  - Input stream instance for access to the request's content
- `java.io.OutputStream` / `java.io.Writer`
  - Output stream instance for generating the response's content



# Supported Handler Method Argument Types

- `java.security.Principal`
  - Containing the currently authenticated user
- `java.util.Map` / `org.springframework.ui.Model`
  - Map or model instance represent the model object the will be passed to view after controllers finished.
- `Command` or `form objects`
  - To bind request parameters to bean properties
- `org.springframework.validation.Errors`
  - An instance of errors that contains all errors of binding and validation errors.
- `org.springframework.validation.BindingResult`
  - An instance of errors that contains all errors of binding errors only.
- `org.springframework.web.bind.support.SessionStatus`





# Supported Handler Method Argument Types

- **@RequestParam** annotated parameters
  - For access to specific Servlet request parameters
- **@PathVariable** annotated parameters
  - For access to URI template variables
- **@RequestHeader** annotated parameters
  - For access to specific Servlet request HTTP headers
- **@RequestBody** annotated parameters
  - For access to the HTTP request body
- **@RequestPart** annotated parameters
  - For access to the content of a "multipart/form-data" request part.



# Supported Handler Method Argument Types

- A **ModelAndView** object
- A **Model** object
- A **Map** object for exposing a model
- A **View** object
- A **String value** that is interpreted as the logical view name.
- **void**
- **Any other return type** is considered to be a single model attribute to be exposed to the view
- **Note:**
  - If the method is annotated with **@ResponseBody**, the return type is written to the response HTTP body



# Binding request parameters With @RequestParam

- The **@RequestParam** annotation is used to bind request parameters to a method parameter in a controller.

```
@RequestMapping(method = RequestMethod.GET, params = "userId")
public String displayUser1(@RequestParam("userId") Integer id, Model data) {
    User user = userService.getUser(id);
    data.addAttribute("users", user);
    return "displayAll";
}
```



# Binding request parameters With @PathVariable

- The **@PathVariable** annotation is used to bind request parameters based on URI to a method parameter in a controller.

```
@RequestMapping(value =("/{userId}", method = RequestMethod.GET)
public String displayUser2(@PathVariable("userId") Integer id, Model data) {
    User user = userService.getUser(id);
    data.addAttribute("users", user);
    return "displayAll";
}
```



# Binding request parameters With @RequestBody

- The **@RequestBody** annotation is used to indicate that a method parameter should be bound to the value of the HTTP request body.

```
@RequestMapping(value = "/writeData", method = RequestMethod.PUT)
public void handle(@RequestBody String body, Writer writer) throws IOException {
    writer.write(body);
}
```



# Mapping the response body with the @ResponseBody

- The **@ResponseBody** annotation is similar to **@RequestBody**
- This annotation can be put on a method and indicates that the return type should be written straight to the HTTP response body (and not placed in a Model, or interpreted as a **view name**).

```
@ResponseBody
@RequestMapping(value = "/getUserJson", method = RequestMethod.GET, params = "userId")
public String displayUser3(@RequestParam("userId") Integer id, Model data) {
    User user = userService.getUser(id);
    return user.toJson();
}
```



# Accessing Model Data With @ModelAttribute

- When @ModelAttribute is placed on a method parameter, it maps a model attribute to the specific, annotated method parameter.
- This is how the controller gets a reference to the object holding the data entered in the form.

```
@RequestMapping(value = "register", method = RequestMethod.POST)
public String processSubmit(@ModelAttribute("user") User user) {
    userService.addUser(user);
    return "userAdded";
}
```



# Mapping cookie values with the @CookieValue

- The **@CookieValue** annotation allows a method parameter to be bound to the value of an HTTP cookie.

```
@RequestMapping(value = "/displayCookie", method = RequestMethod.GET)
public void displayCookieInfo(@CookieValue("JSESSIONID") String cookie) {
    //...
}
```





# Mapping request header attributes with the `@RequestHeader`

- The `@RequestHeader` annotation allows a method parameter to be bound to a request header.

Host	localhost:8080
Accept	text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Language	fr,en-gb;q=0.7,en;q=0.3
Accept-Encoding	gzip,deflate
Accept-Charset	ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive	300

```
@RequestMapping(value = "/displayHeaders", method = RequestMethod.GET)
public void displayHeaderInfo(@RequestHeader("Accept-Encoding") String encoding,
    @RequestHeader("Keep-Alive") long keepAlive) {
    //...
}
```



# Form Processing

- Working with forms in a web application involves two operations:
  - Displaying the form
  - Processing the form submission
- We need to handle those two operations using controllers.
- The two operations can be handled in one controller with two handler methods.



# Form Processing (Ex.)

- **Displaying Form:**
- When the form is displayed, it'll need a model object to bind to the form fields and hold data.
- **createNewUser()** method simply creates a new instance of a Customer and adds it to the model. It then wraps up by returning signup as the logical name of the view that will render the form.

```
@RequestMapping(value = "/register", method = RequestMethod.GET)
public String createNewUser(Model model) {
    model.addAttribute(new User());
    return "formView";
}
```



# Form Processing (Ex.)

- **Creating View:**
- The view will be a jsp page which uses Spring's form binding library
- Also **commandName** is removed in **version 5** and replaced with **modelAttribute**

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<%@taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
  <body>
    <c:set var="contextPath" value="${pageContext.request.contextPath}"/>
    <form:form method="POST" modelAttribute="user"
      action="${contextPath}/admin/users/register.htm" >
      <b><spring:message code="field.username"/></b>
      <form:input path="name" />
      <form:errors path="name"/>
      <br/>
      <b><spring:message code="field.salary"/></b>
      <form:input path="salary" />
      <form:errors path="salary"/>
      <br/>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```



# Form Processing (Ex.)

- **Creating View:**
- The **<form:form>** tag binds the User object (identified by the modelAttribute instead of commandName attribute)
- The **<form:input>** tag has a path attribute that references the property of the User object that the form is bound to.
- When the form is submitted, whatever values these fields contain will be placed into a User object and submitted to the server for processing.
- With **no URL specified** as **form action**, it'll be submitted back to **the same URL path** that **displayed the form**.



# Form Processing (Ex.)

- **Processing Form Input:**
- After the form is submitted, we'll need a handler method that takes a User object (populated with data from the form) and saves it.

```
@RequestMapping(value = "/register", method = RequestMethod.POST)  
public String addUser(@ModelAttribute("user") User user) {  
    userService.addUser(user);  
    return "successView";  
}
```



# Form Processing Full Example

```
@Controller
@RequestMapping("/admin/users")
public class UserController {

    @Autowired
    private UserService userService;

    @RequestMapping(value = "/register", method = RequestMethod.GET)
    public String createNewUser(Model model) {
        model.addAttribute(new User());
        return "formView";
    }

    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("user") User user) {
        userService.addUser(user);
        return "successView";
    }
}
```



# Input Validation

- Beginning with Spring 3.
- Spring MVC has the ability to automatically validate Controller inputs.
- Spring 3 includes support for JavaBean validation specification (JSR-303 ).
- To trigger validation of a Controller input you have to do two steps for input validation:
  - Annotate input argument with @Valid

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String addUser(@Valid @ModelAttribute("user") User user) {
    userService.addUser(user);
    return "successView";
}
```

- Annotate bean properties to specify validation rules
- The following table includes the built-in bean validation constraints.





# Input Validation (Ex.)

Constraint	description	example
@DecimalMax	The value of the field or property must be a decimal value lower than or equal to the number in the value element.	@DecimalMax("30.00") BigDecimal discount;
@DecimalMin	The value of the field or property must be a decimal value greater than or equal to the number in the value element.	@DecimalMin("5.00") BigDecimal discount;
@Digits	The value of the field or property must be a number within a specified range. The integer element specifies the maximum integral digits for the number, and the fraction element specifies the maximum fractional digits for the number.	@Digits(integer=6, fraction=2) BigDecimal price;
@Future	The value of the field or property must be a date in the future.	@Future Date eventDate;



# Input Validation (Ex.)

Constraint	description	example
@Max	The value of the field or property must be an integer value lower than or equal to the number in the value element.	@Max(10) int quantity;
@Min	The value of the field or property must be an integer value greater than or equal to the number in the value element.	@Min(5) int quantity;
@NotNull	The value of the field or property must not be null.	@NotNull String username;
@Null	The value of the field or property must be null.	@Null String unusedString;
@Past	The value of the field or property must be a date in the past.	@Past Date birthday;



# Input Validation (Ex.)

Constraint	description	example
@Pattern	The value of the field or property must match the regular expression defined in theregexp element.	@Pattern (regexp= "\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;
@Size	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a String, the size of the string is evaluated. If the field or property is a collection the size of the Collection is evaluated. If the field or property is a Map or array , the size of the it is evaluated. Use one of the optional max or min elements to specify the boundaries.	@Size (min=2, max=240) String briefMessage;



# Handling Validation Errors

- In all of the validation annotations, message attribute can be used with the message to be displayed in the form when validation fails so that the user knows what needs to be corrected.

```
public class User {  
  
    private int id;  
    @Size(min = 3, max = 50,  
          message = "Your full name must be between 3 and 50 ")  
    private String name;  
    @Min(value = 1200,message = "Invalid Salary")  
    private float salary;  
}
```



# Handling Validation Errors

- In order to check whether the validation succeeded or failed, we need to pass **BindingResult** Object to the handler method.
- **BindingResult** Object knew whether the form had any validation errors.
- Checking if there were any errors or not is by calling its **hasErrors()** method.

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String addUser(@Valid @ModelAttribute("user") User user, BindingResult result) {
    if (result.hasErrors()) {
        return "formView";
    } else {
        userService.addUser(user);
        return "successView";
    }
}
```



# Displaying Validation Errors

- Spring's form binding JSP tag library is used to display the errors.
- The `<form:errors>` tag can render field validation errors.
- The `<form:errors>` tag's path attribute specifies the form field for which errors should be displayed.
- If there are **multiple errors** for **a single field**. `<form:errors path="name"/>`
  - They will be all displayed, separated by an HTML `<br/>` tag.
- If you'd rather have them **separated some other way**.
  - Then you can use the delimiter attribute. `<form:errors path="name" delimiter=", "/>`
- To display all of the errors in one place (perhaps at the top of the form)
  - Use `<form:errors>` tag, with its path attribute set to `*`. `<form:errors path="*/>`



# Redirecting to Controller

- The special "**redirect:**" Prefix allows you to redirect to another resource.
- If a view name is returned that has this Prefix, the view resolver will recognize this as a special indication that a redirect is needed.
- The rest of the view name will be treated as the redirect URL.

```
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String addUser(@Valid @ModelAttribute("user") User user, BindingResult result) {
    if (result.hasErrors()) {
        return "formView";
    } else {
        userService.addUser(user);
        return "redirect:/another.htm";
    }
}
```



# Forward to Controller

- The special "**forward:**" Prefix allows you to forward to another controller.
- If a view name is returned that has this Prefix, the view resolver will recognize this as a special indication that a forward is needed.
- The rest of the view name will be treated as the forward URL.

```
@RequestMapping(value = "/register", method = RequestMethod.POST)  
public String addUser(@Valid @ModelAttribute("user") User user, BindingResult result) {  
    if (result.hasErrors()) {  
        return "formView";  
    } else {  
        userService.addUser(user);  
        return "forward:/another.htm";  
    }  
}
```





# Uploading Files

1. Using commons-fileupload API
2. Using Servlets 3.0 API



# Uploading Files

- Remember:
  - The form method must be **POST**
  - The **enctype** attribute must have the value **multipart/form-data**
  - The controller method must have an argument of type **MultiPartFile**



# Uploading Files

```
<form method="POST" action="uploadFile" enctype="multipart/form-data">
```

File to upload: `<input type="file" name="file">`

Name: `<input type="text" name="name">`

`<input type="submit" value="Upload">` Press here to upload the file!

```
</form>
```



# Uploading Files

```
@RequestMapping(value = "/uploadFile", method = RequestMethod.POST)
@ResponseBody
public String uploadFileHandler(@RequestParam("name") String name,
    @RequestParam("file") MultipartFile file) {
    if (!file.isEmpty()) {
        .....
        return "You successfully uploaded file=" + file.getOriginalFilename();
    } else {
        return "You failed to upload " + file.getOriginalFilename()
            + " because the file was empty.";
    }
}
```



# Uploading Files- commons-fileupload

1. Include dependency for the Apache **commons-fileupload** library
2. Define **multipartResolver** in the **Dispatcher-Servlet.xml** of type: `“org.springframework.web.multipart.commons.CommonsMultipartResolver”`




# Uploading Files- Servlets 3.0 API

1. In **web.xml** add **<multipart-config>** tag inside the **<servlet>** tag of the Dispatcher Servlet

```
<multipart-config>  
    <max-file-size>5242880</max-file-size>5MB  
    <max-request-size>20971520</max-request-size>20MB  
    <file-size-threshold>0</file-size-threshold>  
</multipart-config>
```

2. Define **multipartResolver** in the **Dispatcher-Servlet.xml** of type:  
“org.springframework.web.multipart.support.StandardServletMultipartResolver”



# References & Recommended Reading





# References & Recommended Reading

- [Spring Framework Documentation Version 5.3.6.RELEASE](#)
- Spring in Action 5th Edition
- Cloud Native Java
- Learning Spring Boot 2.0
- Spring 5 Recipes: A Problem-Solution Approach



