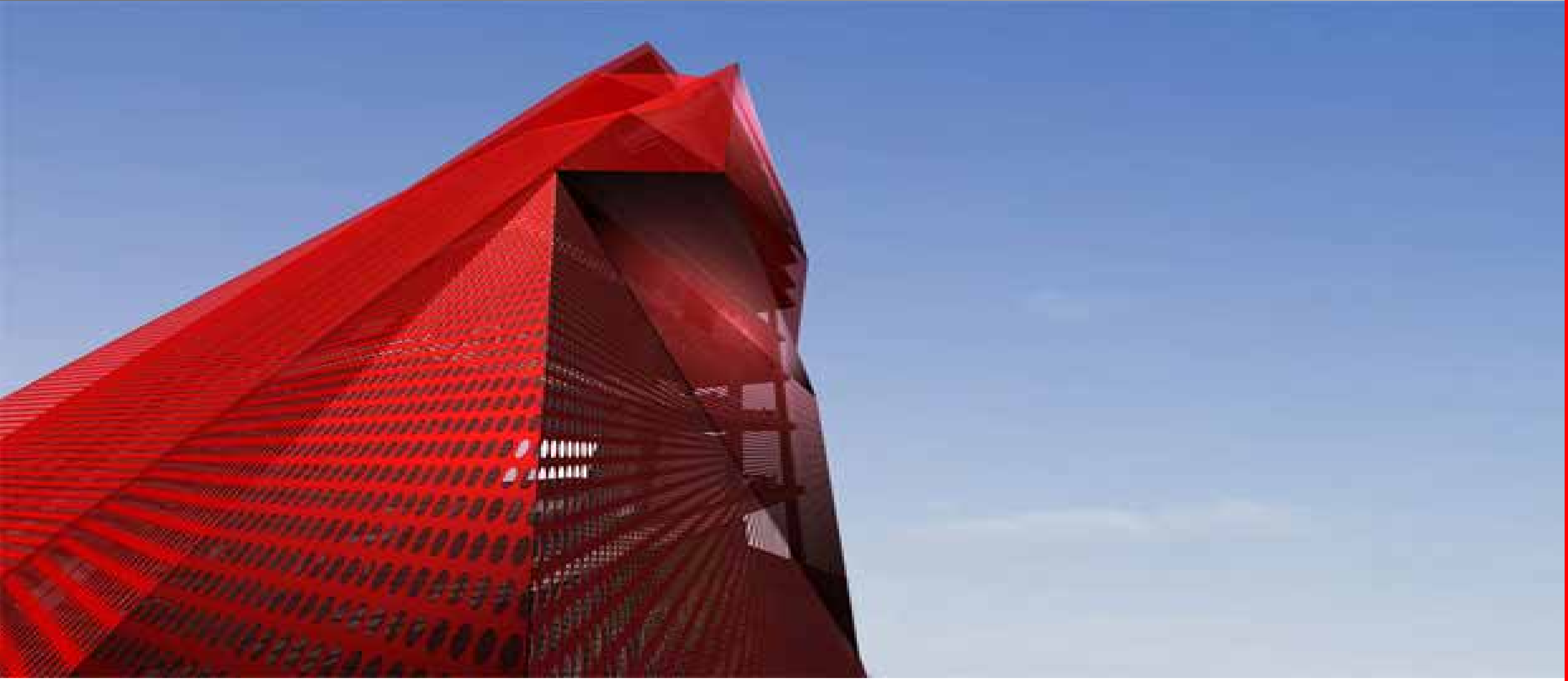


Training – Technical development

Spring – MVC Concepts and Programming

Duration – 4 hours

Agenda

- Introduction
- Using Spring in Web Applications
- Overview of Spring Web
- Dispatcher Servlet configuration
- Extracting Request parameter
- View Resolvers

Spring IoC/DI

- IoC is in short ‘the object sharing and life cycle management ’ service provided by the Spring container.
- You can compare this to the Java Beans object creation and scope management offered by the java web server (container) for Java Server pages in web applications.
- The IoC container provides all the dependencies required for creating a POJO object i.e. initializing the POJO properties which may be simple types, collections or other java objects which may be POJO again, i.e. the IoC injects the dependencies into the objects and hence the term Dependency Injection.

Who supports IoC ?

- The BeanFactory inteface standardizes the Spring bean container behavior.
- It provides the basic client view of a bean container which is responsible for creating and managing the life cycle of POJO bean objects.
- ApplicationContext in web applications
- The couple of implementations such as, XmlWebApplicationContext , FileXmlApplicationContext are responsible for managing the POJO objects in different contexts.

How Dependency Injection works ?

- The IoC container offers these services to POJO objects which are not tied to any specific framework.
- The POJO object configuration can be programmatic or declarative.
- These POJO objects managed by the container are called as beans.

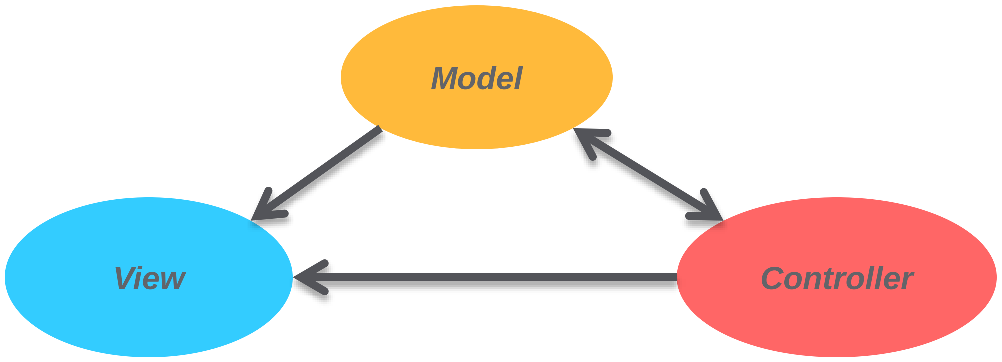
Injecting Dependencies

- The property values are injected into the bean objects in three ways
- Setter injection :Through setter methods in POJO classes.
- Constructor injection: Through parameterized constructors:
- Method injection: dynamic method implementations by the container in the same way as CMP EJB.

MVC Introduction

- MVC is software design pattern for developing Web applications. It is about the following three components:
  - Model
  - View
  - Controller
- Model : This is responsible for maintaining data
- View: This is responsible for presenting the data to the end User
- Controller : This is responsible for manipulating the data and providing interaction between model and view.

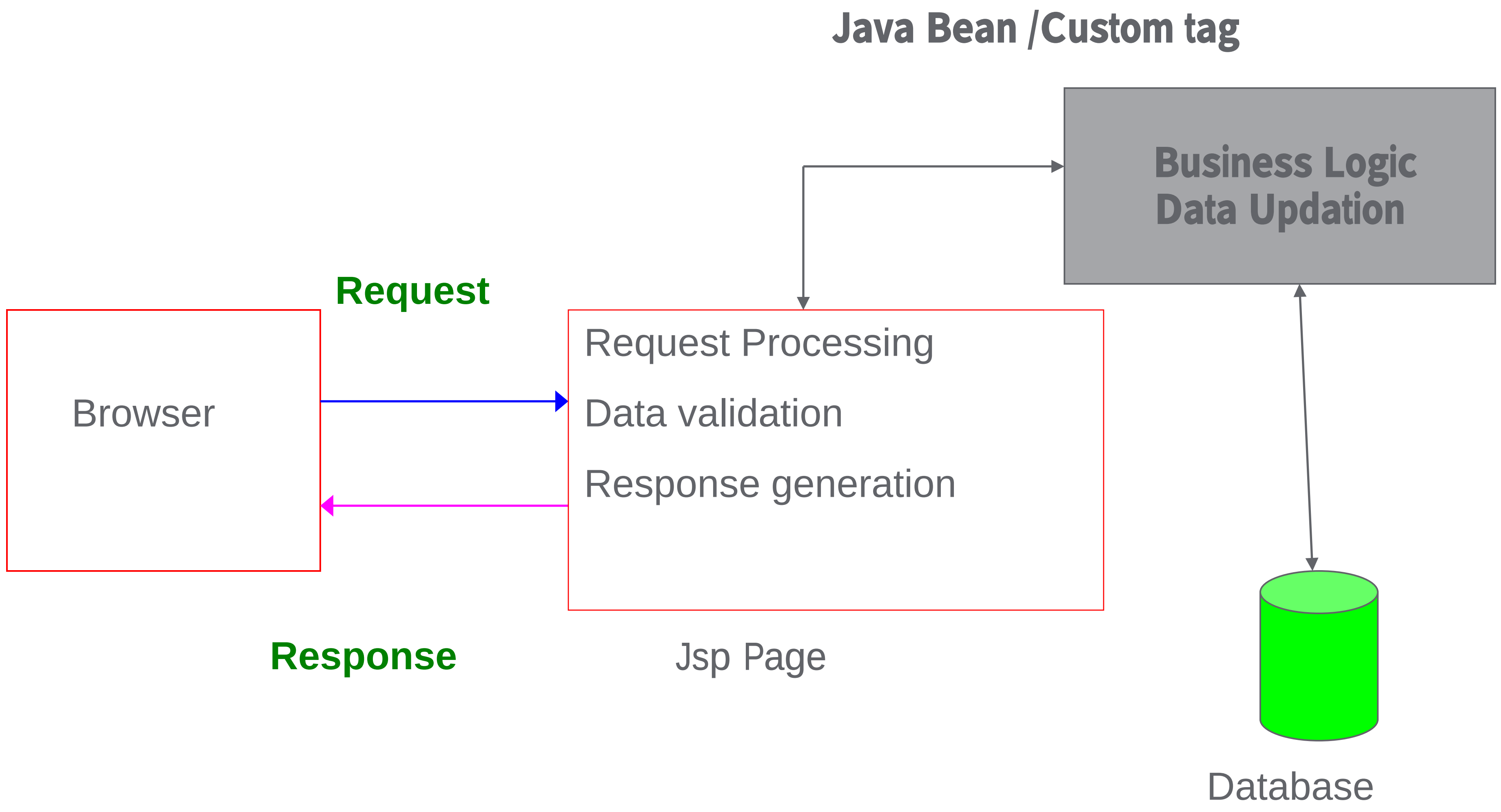
- MVC is a popular design pattern as it isolates the presentation and business logic and supports separation of concerns.



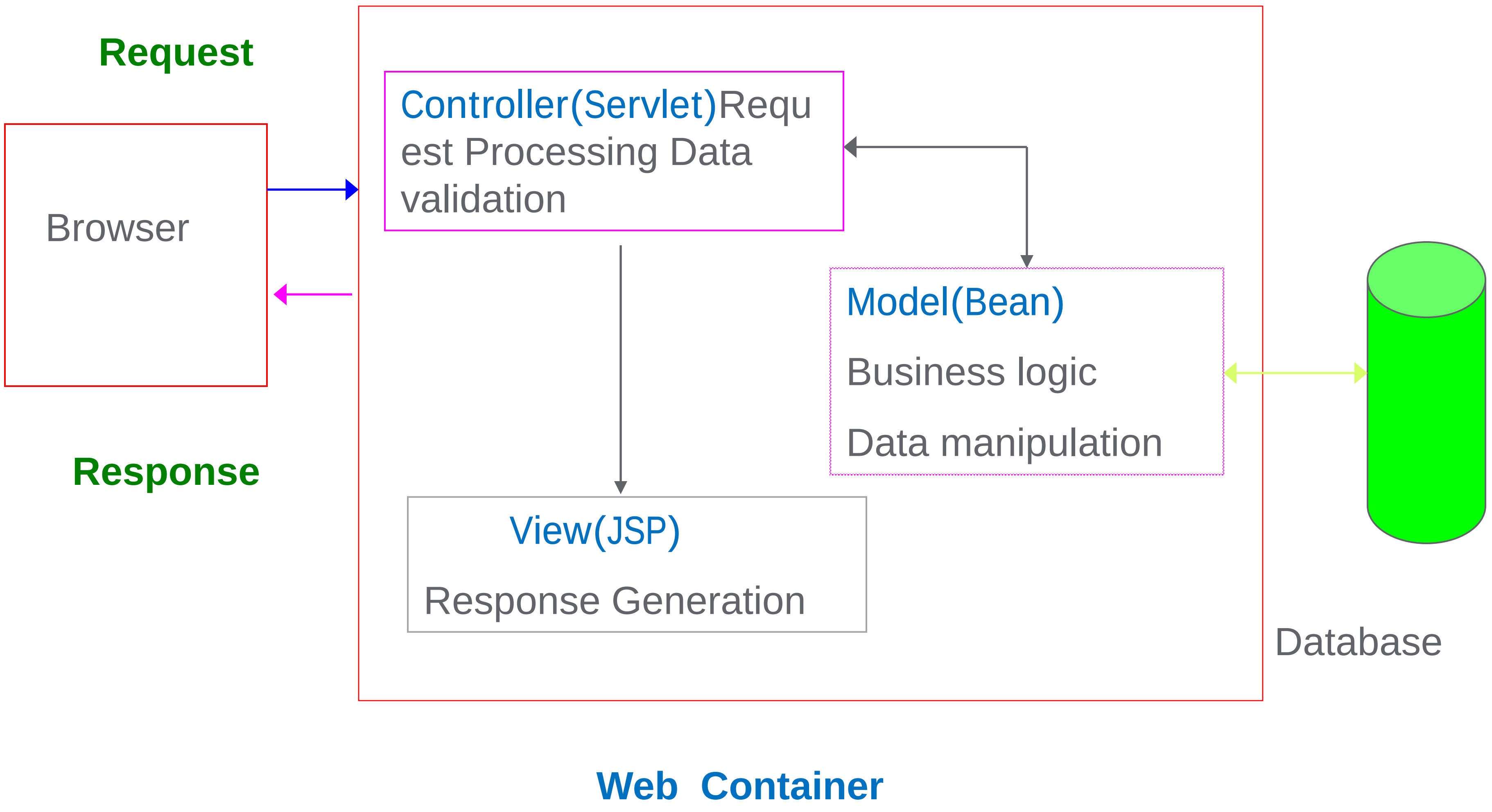
Model 1 architecture

- Simple to develop
- Not suitable for large scale application because lot of code is duplicated
- Difficult to maintain and trouble shoot
- No code reusability

Code Separation in JSP



Model 2-MVC architecture



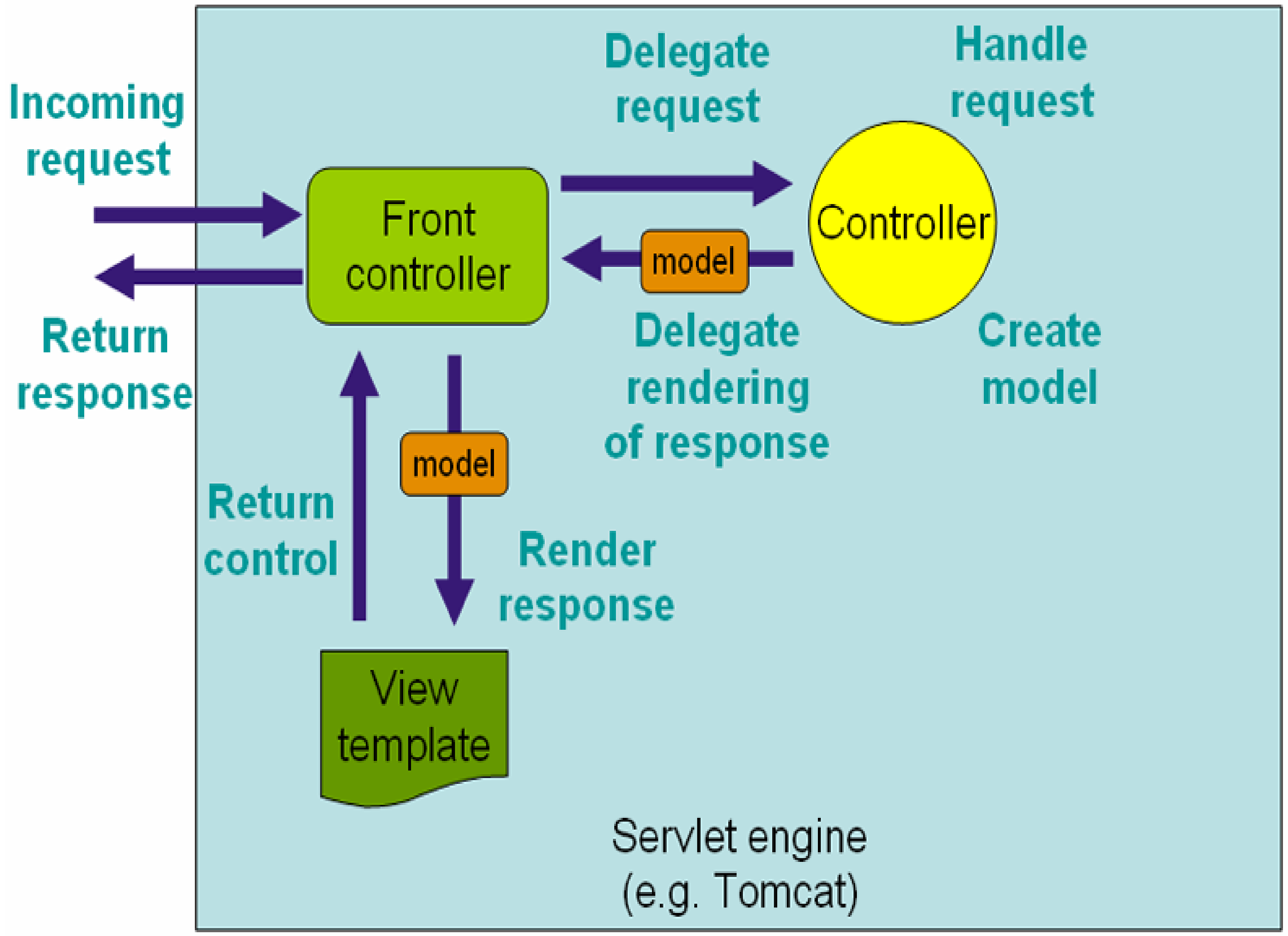
MVC Advantage

- Clear separation of business logic,data access and validation and view(presentation)code
- Excellent reuse of code,same model or controller can be applied to another type of view,application
- Model,View and controller can be implemented or changed independent of each other

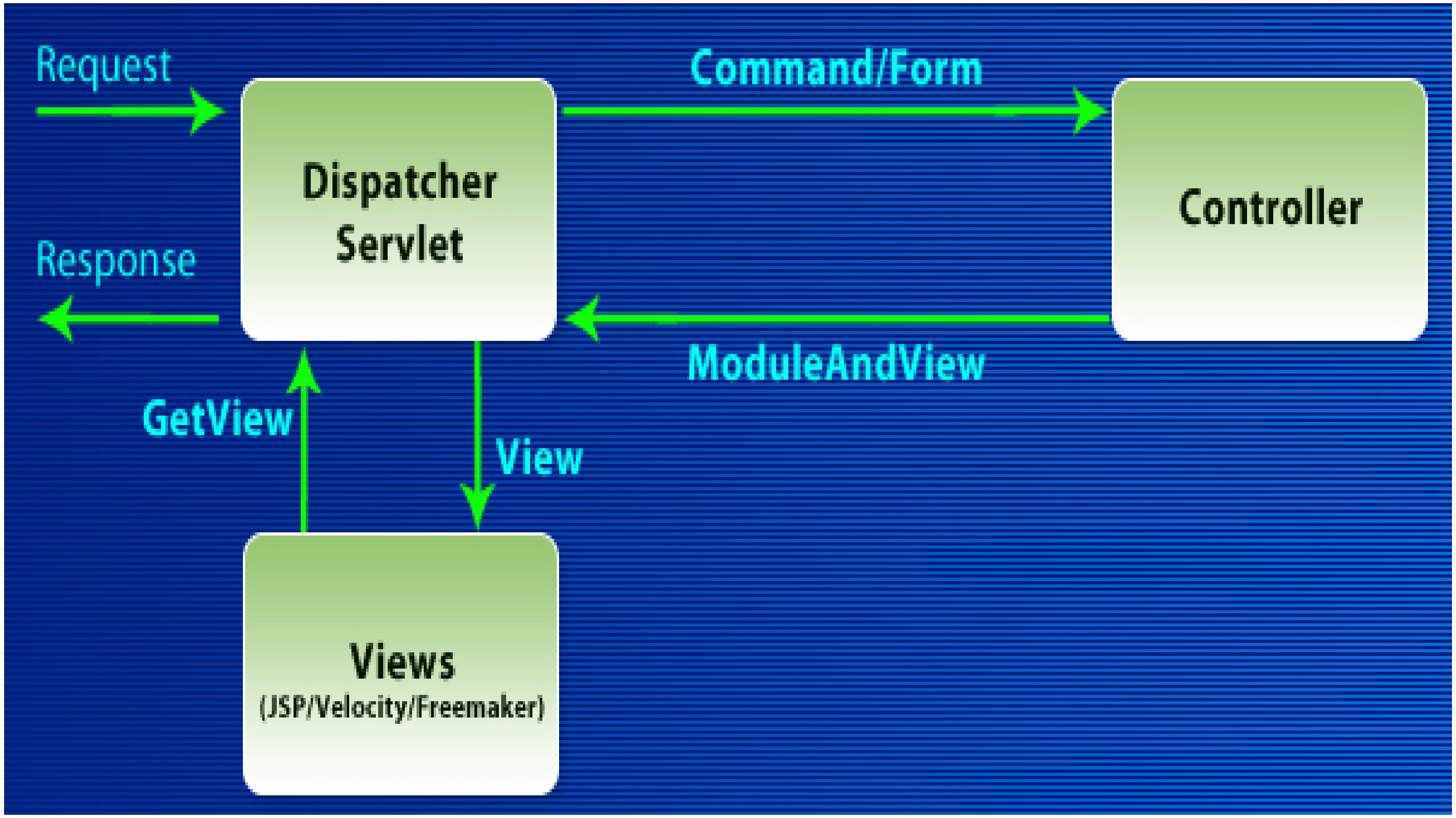
Spring MVC Features

- Clear separation of roles.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans.
- Adaptability, non-intrusiveness, and flexibility.
- Reusable business code, no need for duplication.
- Customizable binding and validation.
- Customizable handler mapping and view resolution.
- Flexible model transfer.
- Customizable locale and theme resolution,
- A simple yet powerful JSP tag library

Spring MVC framework



MVC data flow



Spring MVC Components

- Spring Separates the Dispatcher and Controller behavior in two different components.
- DispatcherServlet responsible for intercepting the request and dispatching for specific urls. Analogous to ActionServlet of Struts framework.
- Controller interface implementations specifies user defined model, view coordination, equivalent to Action in Struts.
- ModelAndView class objects encapsulates view and model linking.
- Any Java Object can be used as Command object (FormBean in Struts).

The DispatcherServlet

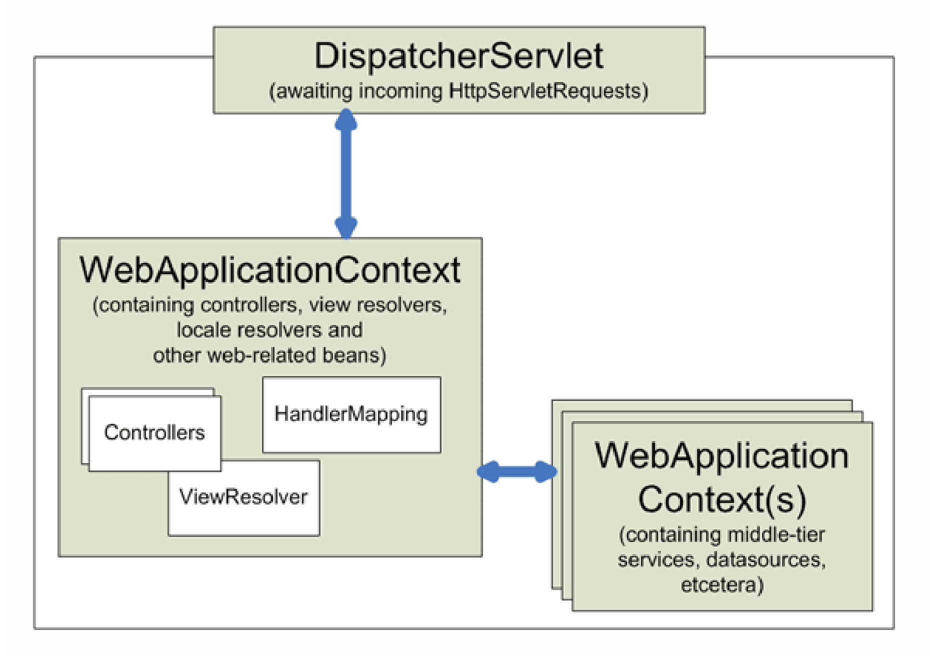
- Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

- The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), and as such is declared in the web.xml of your web application. You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file.

```
<web-app>
    <servlet>
            <servlet-name>example</servlet-name>
            <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
            <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
            <servlet-name>example</servlet-name>
            <url-pattern>*.form</url-pattern>
    </servlet-mapping>
</web-app>
```

Context hierarchy in Spring Web MVC

- In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext. These inherited beans can be overridden in the servlet-specific scope, and you can define new scope-specific beans local to a given servlet instance


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

- Upon initialization of a DispatcherServlet, the framework looks for a file named [servlet-name]-servlet.xml in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

Example

```
<web-app>
    <servlet>
                  <servlet-name>demo</servlet-name>
                  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                  <load-on-startup>1</load-on-startup>
   </servlet>
   <servlet-mapping>
                  <servlet-name>demo</servlet-name>
                  <url-pattern>/demo/*</url-pattern>
   </servlet-mapping>
</web-app>
```

- With the above servlet configuration in place, you will need to have a file called /WEB-INF/demo-servlet.xml in your application; this file will contain all of your Spring Web MVC-specific components (beans).

Special beans in the WebApplicationContext

- The Spring DispatcherServlet uses special beans to process requests and render the appropriate views. These beans are part of Spring Framework. You can configure them in the WebApplicationContext, just as you configure any other bean. However, for most beans, sensible defaults are provided so you initially do not need to configure them. These beans are described in the following table.

Bean type	Explanation
controllers	Form the **C** part of the MVC.
handler mappings	Handle the execution of a list of pre-processors and post-processors and controllers that will be executed if they match certain criteria (for example, a matching URL specified with the controller).
view resolvers	Resolves view names to views.
locale resolver	A locale resolver is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views
Theme resolver	A theme resolver is capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart file resolver	Contains functionality to process file uploads from HTML forms.
handler exception resolvers	Contains functionality to map exceptions to views or implement other more complex exception handling code.

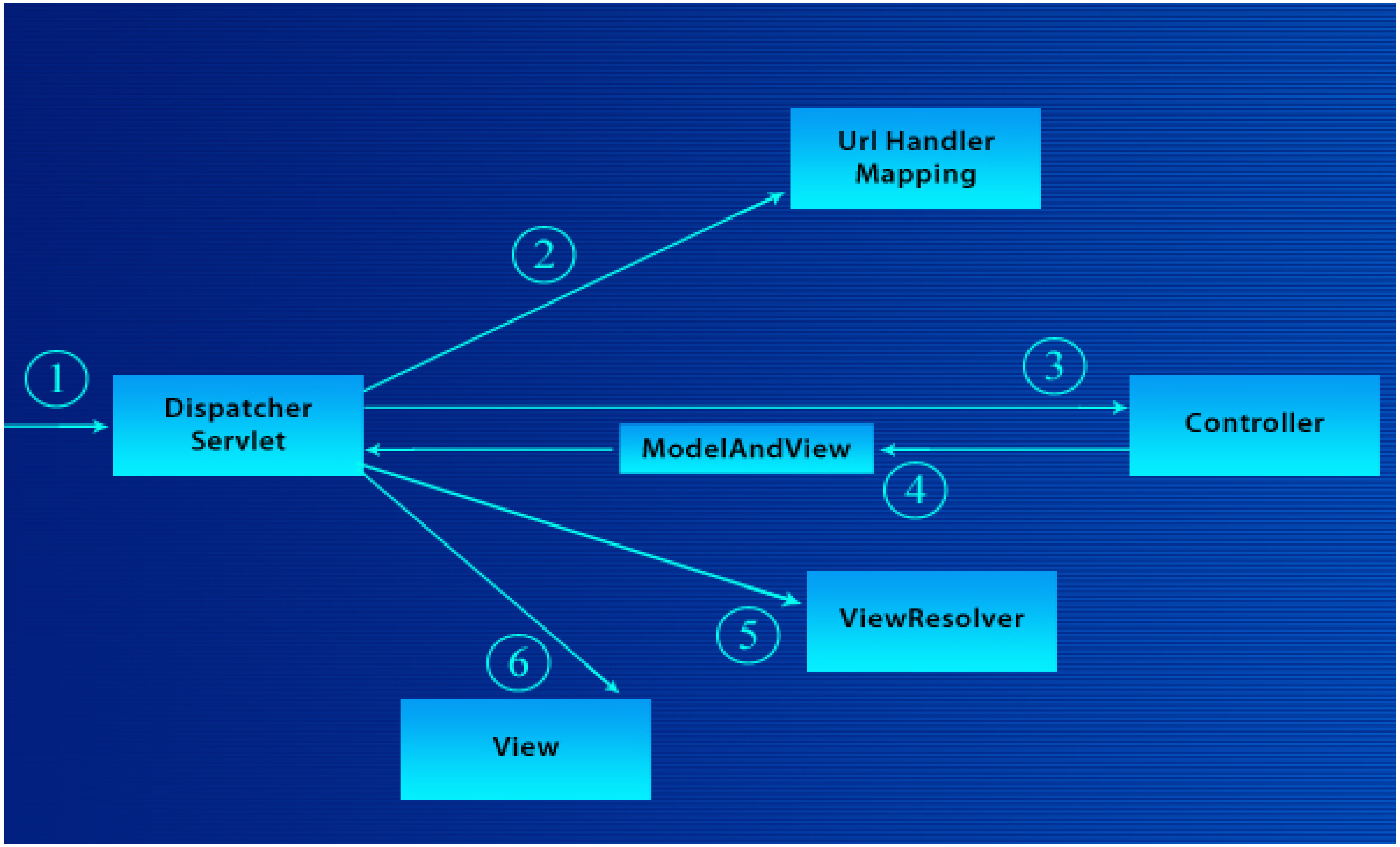
demo-servlet.xml

```
<beans>
   <!-- Controller class mappings  à
   <bean id="DemoControl" class="MyController"/>
         <bean id="HelloControl" class="HelloController">
      <property name="productManager">
          <ref bean="prodMan"/>
      </property>
  </bean>
<bean id="urlMapping" class= "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
     <property name="mappings">
                      <props>
                        <prop key="/work.htm">DemoControl</prop>
                        <prop key="/hello">HelloControl</prop>
                      </props>
                     </property>
 </bean> <beans>
```

beans for the Dispatcher

- Controllers: classes that implement Controller.
- Handler mappings: handle the execution of a list of pre- and post-processors and controllers that will be executed if they match certain criteria.
- View Resolvers: resolves view names to views. (JSPs)
- Locale Resolver : resolves the locale a client is using and loading the specific property files (through ResourceBundle) for internationalized views.
- Theme resolver : Class capable of resolving themes your web application can use, for example, to offer personalized layouts.
- Handler exception resolver: offer functionality to map exceptions to views orimplement other more complex exception handling code.
- multipart file resolver : offers the functionality to process file uploads from HTML forms.

Request flow in MVC



Steps for writing the first MVC program

1. Create a Dynamic Web project in eclipse based IDE
2. Add all the Spring MVC related jar files in the project
3. Create a requester (Commonly an index file)
4. Create the Dispatcher Servlet entry in Web.xml file
5. Complete the servlet mapping for specific URL
6. Create the spring configuration file based on the name in Web.xml
7. Create a Controller class with appropriate request mapping
8. Write handler mapping in controller class
9. Crate views mostly in jsp
10. Configure view resolver in spring xml file
11. Execute the application.

The ‘C’ in MVC

- Controllers provide access to the application behavior which is typically defined by a service interface.
- Controllers interpret user input and transform this input into a specific model which will be represented to the user by the view.
- Spring provides a base interface Controller and wide variety of its implementations.

Controller interface

```
package org.springframework.web.servlet.mvc;
public interface Controller
{
 public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws Exception;

/* process the request and return a ModelAndView object which the
DispatcherServlet will render.*/

}
```

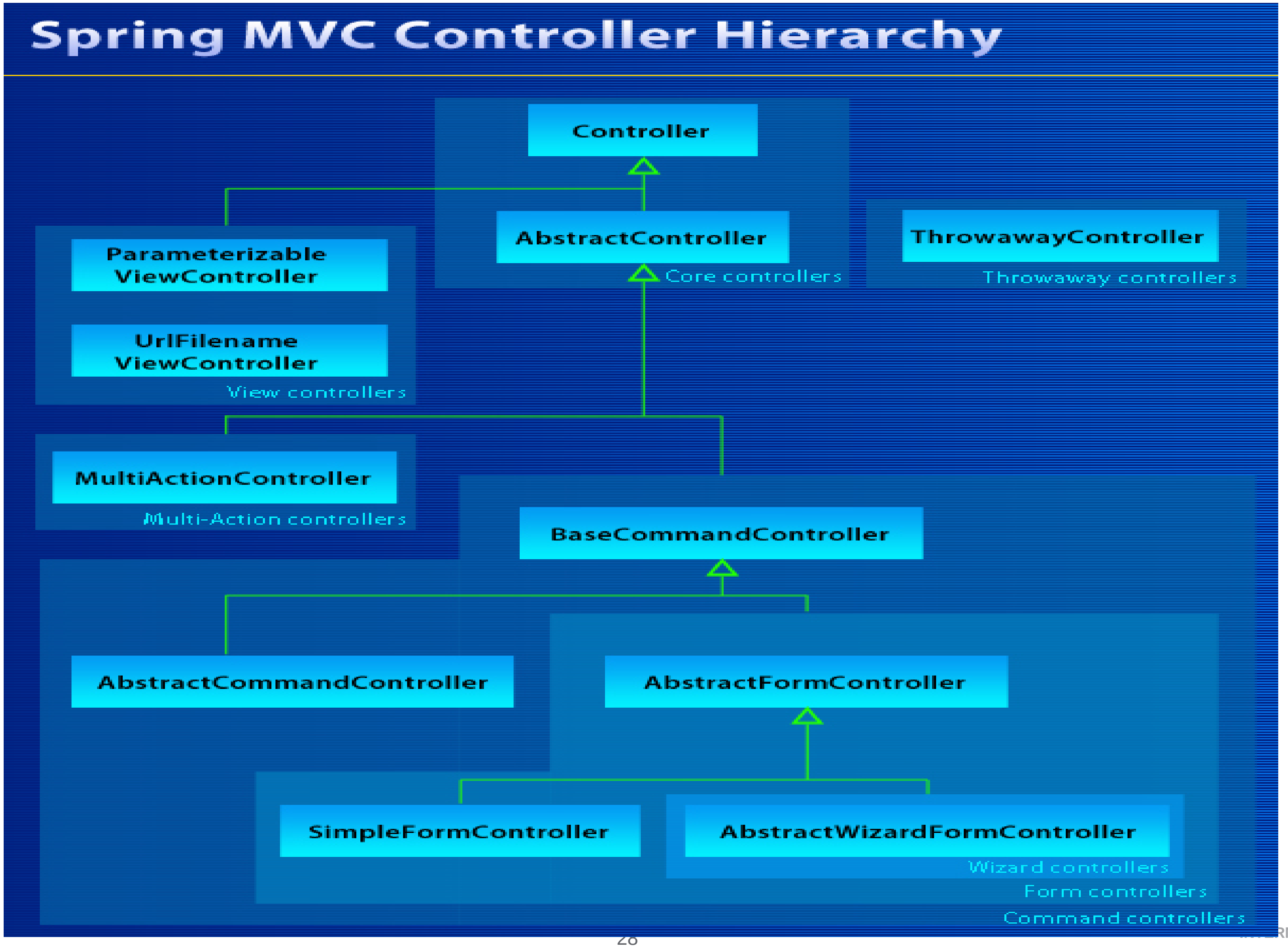
Spring’s Controllers

- AbstractController : abstract base class.
- ParameterizableViewController: returns view name that is defined in the web application context (No har coding of view name).
- UrlFilenameViewController: inspects the URL and retrieves the filename of the file request and uses tha as a view name. e.g. the filename of http://server.com/index.html request is index.
- MultiActionController: groups multiple actions into one controller.

Command controllers

- Command controllers: provide a way to interact with data objects and dynamically bind parameters from the HttpServletRequest to the data object specified.
- These data objects are similar to Struts ActionForm, but in Spring, they don't have to extend from a framework-specific class.

Spring Controllers



More Controllers

- *AbstractCommandController*: used to create your own command controller. Capable of binding request parameters to a data object specified.
- It offers validation features and lets you specify in the controller what to do with the command object that has been populated with request parameter values.

Form Controllers

- Spring provides couple of Form Controllers to deal with client side data validation and submission to action url.
- AbstractFormController : base class to define your own form controller and add validation features.
- SimpleFormController: supports creating a form with a corresponding command object used as FormBean.

Wizard view in web ?

- The AbstractWizardFormController supports wizard like view in web page.
- Extend this class and override couple of methods to provide custom wizard view.

Handler mappings

- Using a handler mapping incoming web requests are mapped to appropriate handlers. These mappings can contain filter-interceptors or url mapping handlers.
- A HandlerExecutionChain will be constructed by the DispatcherServlet which contains the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request.
- The DispatcherServlet will execute the handler and interceptors in the chain (executed before or after the actual handler was executed, or both).

Mapping Requests With @RequestMapping

- You use the @RequestMapping annotation to map URLs such as /savings onto an entire class or a particular handler method.
- Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.
- Example

```
@Controller
@RequestMapping("/check")
public class HelloWorldController {
……
…….
}
```

In the example, the @RequestMapping is used on the type (class) level, which indicates that all handling methods on this controller are relative to the /check path.

@RequestMapping for Methods

- In the following example **`getHelloWorld()`**method has a further @RequestMapping refinement: it only accepts GET requests, meaning that an HTTP GET for /appointments invokes this method.
- The **`postHelloWorld()`** has a similar refinement, which accepts only POST requests

- Example

```
@RequestMapping(method = RequestMethod.GET)
public ModelAndView getHelloWorld() {
System.out.println("GET Controller called");
String message = "Hello World, Spring MVC 3.0! by Irfan";
return new ModelAndView("hello", "message", message);
}
@RequestMapping(method = RequestMethod.POST)
public ModelAndView postHelloWorld() {
System.out.println("POST Controller called");
String message = "Hello World, Spring MVC 3.0! by Irfan";
return new ModelAndView("hello", "message", message);
}
```

@RequestMapping for Methods

- A @RequestMapping on the class level is not required.
- Without it, all paths are simply absolute, and not relative. The following example shows a multi-action controller using @RequestMapping:

```
@Controller
public class MultiController {
@RequestMapping("/hello")
public ModelAndView helloWorld() {
}

@RequestMapping("/check")
public ModelAndView checkWorld() {
}

@RequestMapping("/")
public ModelAndView allWorld() {
}
}
```

Binding request parameters to method parameters

- Controller can access the request parameter
- To access the request parameter use @RequestParam annotation
- Parameter name should be provided
- A variable is required to hold the parameter value

- Example

```
@RequestMapping("/check")
public ModelAndView findName(@RequestParam("name") String name) {
System.out.println("Controller with parameter called");
String message = "Hello World, Spring MVC 3.0! by "+name;
return new ModelAndView("hello", "message", message);
}
```

- Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting @RequestParam's required attribute to false (e.g., @RequestParam(value="id", required=false)).
- Type conversion is applied automatically if the target method parameter type is not String

Model : the ‘M’ in MVC

- The model is generally defined as a Map that can contain objects that are to be displayed in the View.
- The ModelAndView object encapsulates the relationship between view and model and is returned by corresponding Controller methods.
- The ModelAndView class uses a ModelMap class that is a custom Map implementation where valuse are added in key-value fashion.

Actions on Model

- The Controller generates the model object and forwards it to view page.
- Before forwarding validation and command object population will be done by Controller.
- The model map values and command object properties can be accessed in view pages and presented to the user.

View : The presentation

- The view page can be explicitly returned as part of ModelAndView object by the controller
- In case of mapping logical name of view can be resolved to particular view page in case the ModelAndView doesn't contain the view reference.(i.e. null value)
- The view name can be independent of view technology (i.e. without using naming ‘.jsp’ in controller) and resolved to specific technology by using View Resolver and rendered by View.

View Page mapping

- The controller can return a logical view name, which a view resolver resolves to a particular view name and technology and uses view class to render it.
- So the application doesn't tie you to a specific view technology. (i.e. jsp,velocity etc.- no hard coded references to view pages instead it will just return the file name of the view minus its file extension).
- This makes faster portability to other view technologies.

View Resolvers in Spring

- All MVC frameworks for web applications provide a way to address views.
- Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology.
- Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT views
- A custom view resolver can be defined by implementing ViewResolver and View interfaces.

View - URLBasedViewResolver

- The view class is responsible for rendering the view in the browser.
- Spring provides view classes for JSPs, Velocity templates and XSLT views.

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

View - InternalResourceViewResolver

```
<bean id="viewResolver2“ class=
"org..web..view.InternalResourceViewResolver">
<property name="viewClass" value=
"org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
</bean>
```

View - ResourceBundleViewResolver

```
<bean id="viewResolver"

class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

- # And a properties file is uses (views. properties in WEB-INF/classes):
- #These properties file will be specific to client locale.

- The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url.
- .

Chaining View Resolvers

- Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances.
- Chaining view resolvers is straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order.
- The higher the order property, the later the view resolver will be positioned in the chain.
- If a specific view resolver does not result in a view, Spring examines the context for other view resolvers.
- If additional view resolvers exist, Spring continues to inspect them until a view is resolved.
- If no view resolver returns a view, Spring throws a ServletException

Example - Chaining View Resolvers

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

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

Internationalization in Spring

- DispatcherServlet offers to automatically resolve messages using the client's locale and is done with LocaleResolver objects.
- The LocaleResolver can also change the current request Locale.
- Another way : ResourceBundleMessageSource bean can be configured with messageSource id and having the property file base name configured.
- The Locale resolvers are configured in application context of DispatcherServlet .
- An interceptor in the handler mapping can be attached to the request to change the locale under specific circumstances, based on request parameter.

Default View name resolving

- The URL http://localhost/registration.html will result in logical view name of 'registration' being generated by the DefaultRequestToViewNameTranslator.
- This logical view name will then be resolved into the '/WEB-INF/jsp/registration.jsp' view by the InternalResourceViewResolver bean.
- Advantage is hide the actual view names from the web user.

MVC Annotations

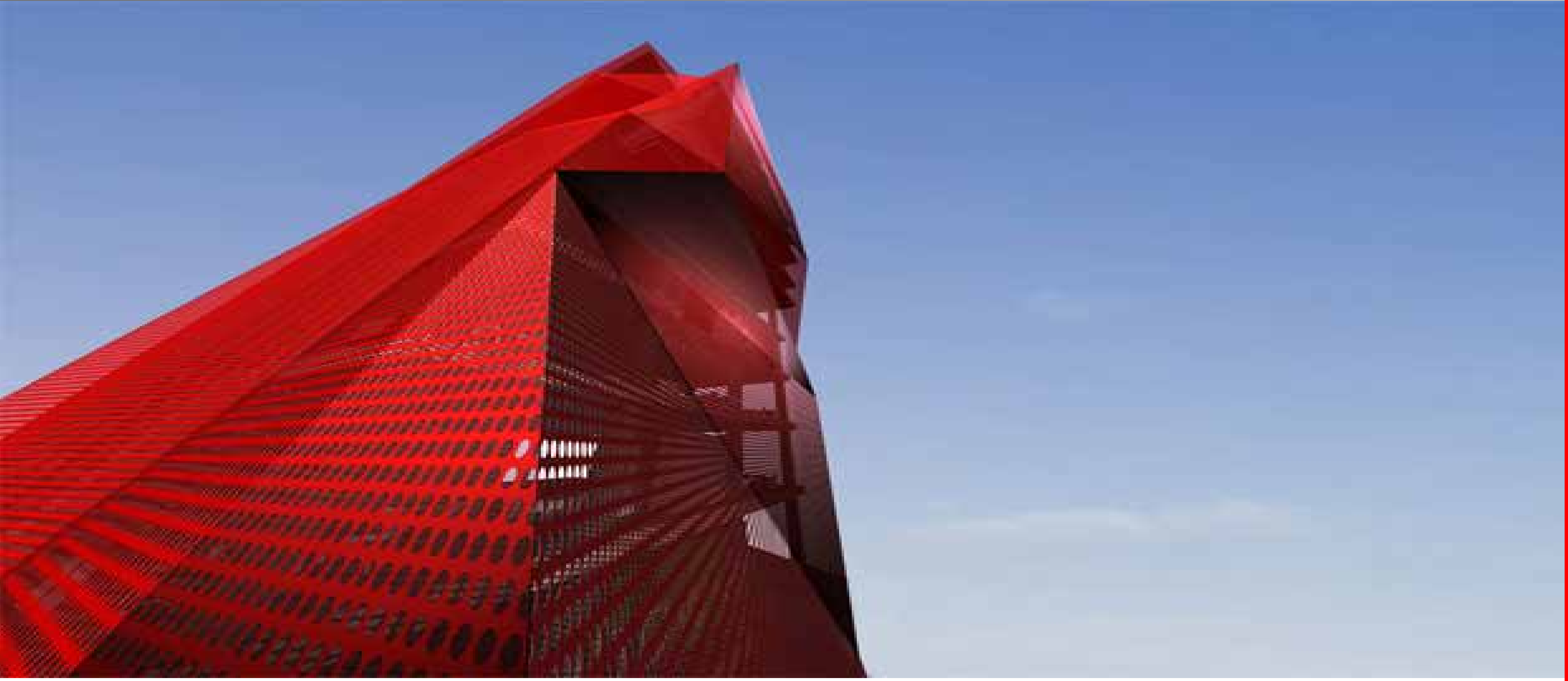
- @Controller
- @RequestMapping
- @SessionAttributes
- @RequestParam
- @ModelAttribute
- @ResponseBody
- @CookieValue
- @ExceptionHandler
- @InitBinder
- @Valid

Spring's form tag library

- Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC.
- Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and Generate the html code.
- This form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with.
- These form tags make JSPs easier to develop, read and maintain.

Handling exceptions

- Spring provides HandlerExceptionResolvers to ease the pain of unexpected exceptions occurring while the request is being handled by a controller which matched the request.
- These provide information about what handler was executing when the exception was thrown.
- Furthermore, a programmatic way of handling exception gives many more options for how to respond appropriately before the request is forwarded to another URL.
- The HandlerExceptionResolver interface specifies the behavior for this handler.



Prepared by M Irfan
Engineering Services– Technical Development

+ 91 20 6705 2532 irfanismailmohammad@hsbc.co.in

Prepared February 2013

INTERNAL