



# Spring MVC 5.x



Rajeev Gupta MTech CS  
Java Trainer

# Spring 5 MVC

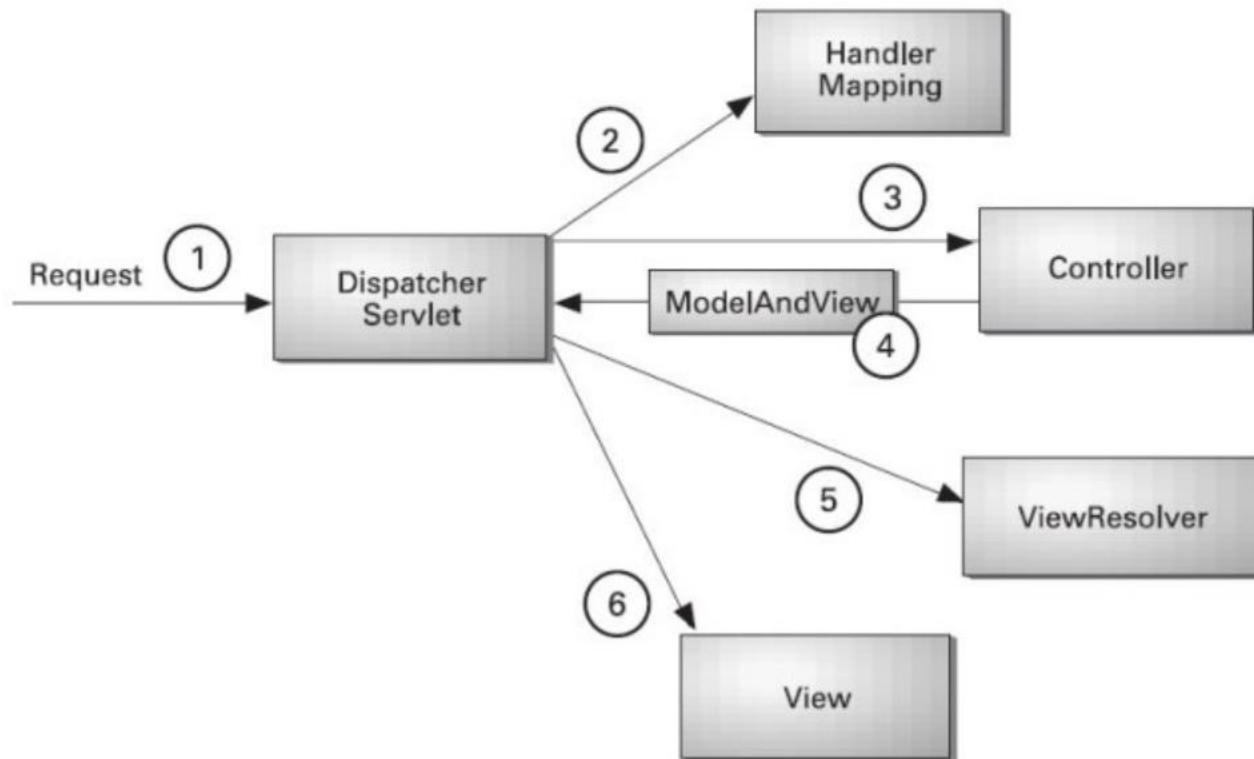
---

- Spring MVC Architecture
- Spring MVC Example
- Spring MVC annotations
- Form processing
- Form validation
- PRG pattern
- Flush attribute

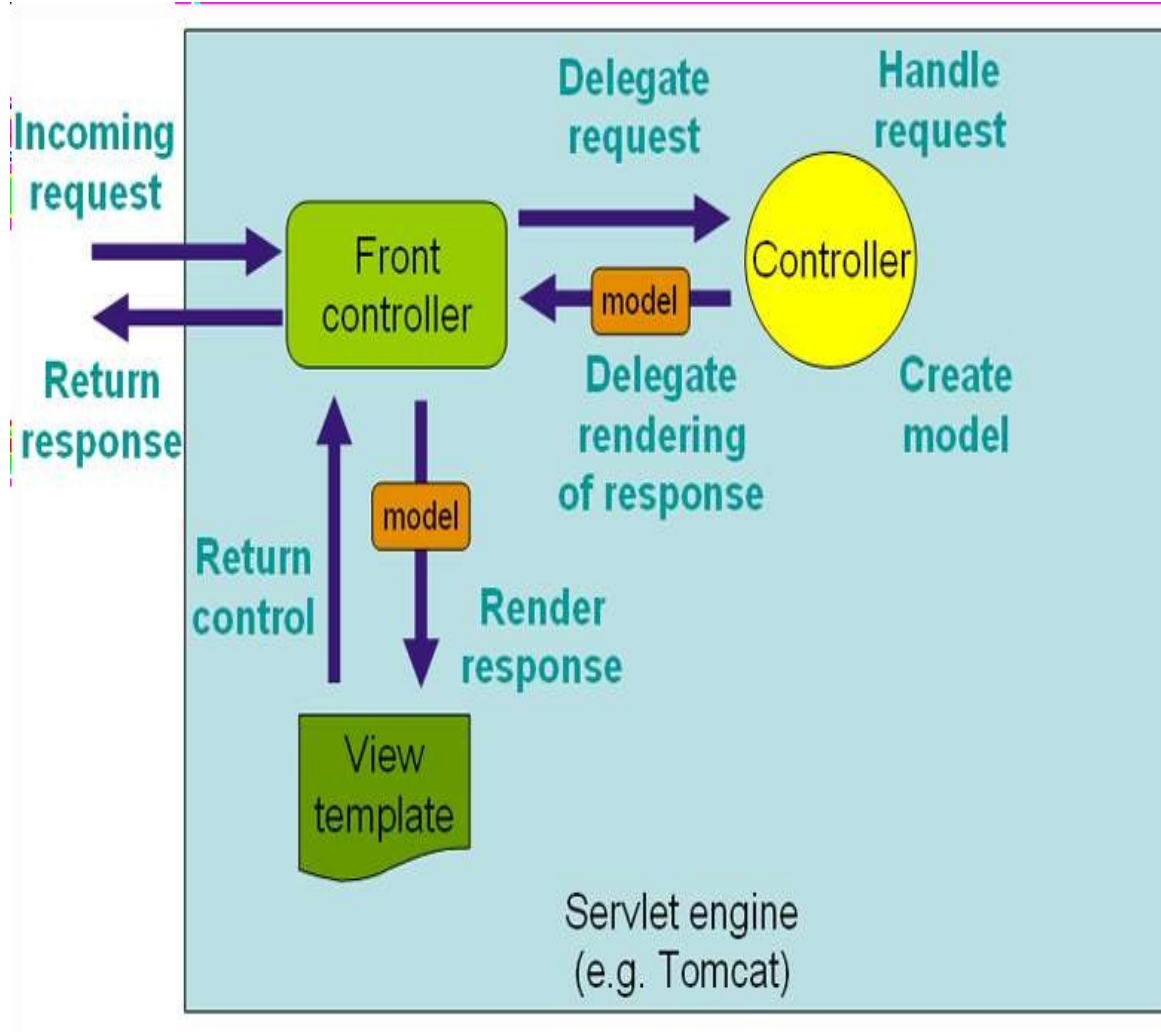
Model mapper

# Spring MVC Architecture

# Spring MVC basic Architecture



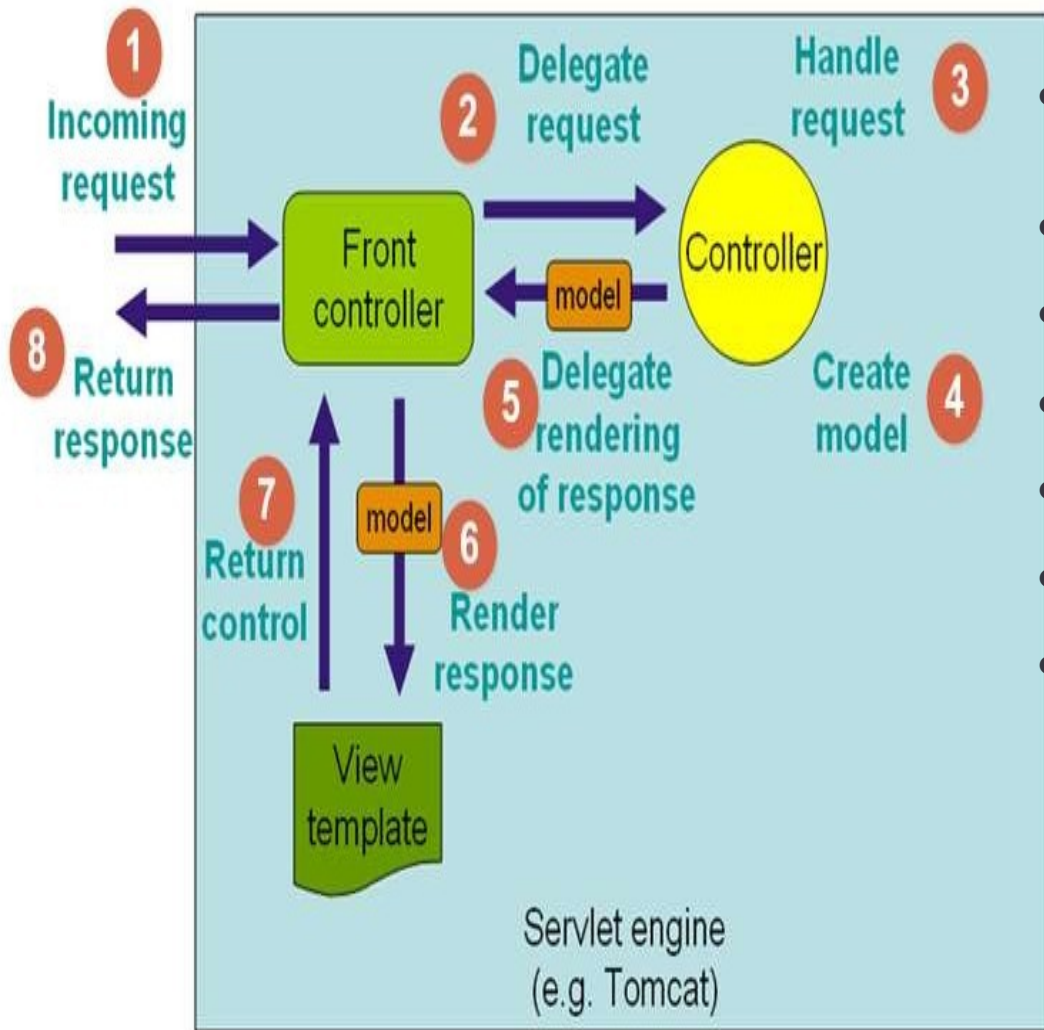
# DispatcherServlet



The

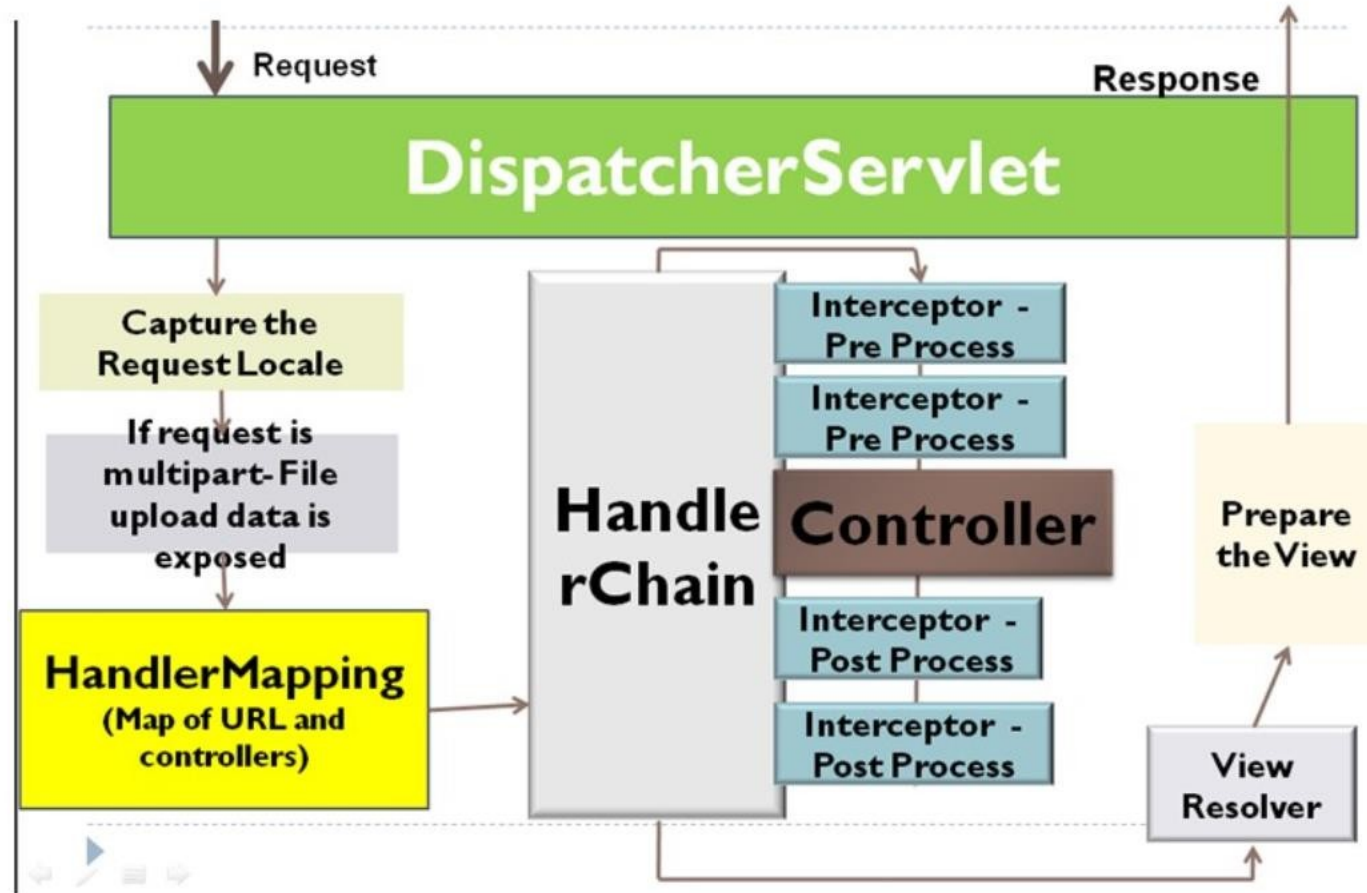
DispatcherServlet is an actual **Servlet** (it inherits from the

HttpServlet base class), and as such is declared in your web application



- Any incoming request that comes to the web application will be sent to Front Controller (Dispatcher Servlet)
- Front Controller decides to whom (Controller) it has to hand over the request, based on the request headers.
- Controller that took the request, processes the request, by sending it to suitable service class.
- After all processing is done, Controller receives the model from the Service or Data Access layer.
- Controller sends the model to the Front Controller (Dispatcher Servlet).
- Dispatcher servlet finds the view template, using view resolver and send the model to it.
- Using View template, model and view page is build and sent back to the Front Controller.
- Front controller sends the constructed view page to the browser to render it for the user requested.

# Spring MVC request flow



# Spring MVC request flow

---

- **DispatcherServlet** receives the request for a URL in the application.
- The **Locale Resolver** component will look for the the Locale information in the request header or session or Cookie as the configuration. The Locale is used to pick the resource files based on the language of the user. This Locale Resolver plays a key role in internationalization of the application.
- The **Theme Resolver** is bound to the request to make the views determine which theme/CSS needs to be applied.
- The **Multipart Resolver** component is invoked to check if the request is for a file upload and then wraps the request to facilitate the file upload functionality.
- The **Handler mapping** component is invoked on the request to get the respective controller which is responsible to handle this request.
- The **DispatcherServlet** then invokes the HandlerChain which will execute the following:
  - Checks if there are any interceptors mapped and invokes the Pre Processing logic.
  - The controllers handler method will be invoked where the request is processed and the result is returned.
  - The mapped interceptors post processing logic will be invoked
- The **DispatcherServlet** based on the result returned by the Controllers handlers method, the ResultToViewNameTranslator component is invoked to generate the view name.
- The **view resolver** will then decide on what view needs to be rendered (JSP/XML/PDF/VELOCITY etc.,) and then the result will be dispatched to the client.



# Spring MVC Configuration xml

# Spring MVC Configuration

- **Step 1:** Configure the web.xml with DispatcherServlet and details of the application context file location.

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

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
</context-param>

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

# Spring MVC Configuration

## □ **Step 2:** Configure the dispatcher-servlet.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"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.controller" />

    <bean id="viewResolver"
          class="org.springframework.web.servlet.view.InternalResourceViewResolver">

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

</beans>
```

# Spring MVC Configuration

## □ **Step 3:** Create controller and view

```
@Controller
public class HelloWorld {
    @RequestMapping("/helloworld")
    public ModelAndView helloWord() {
        String message = "Hello World, Spring 3.0!";
        return new ModelAndView("helloworld", "message", message);
    }
}

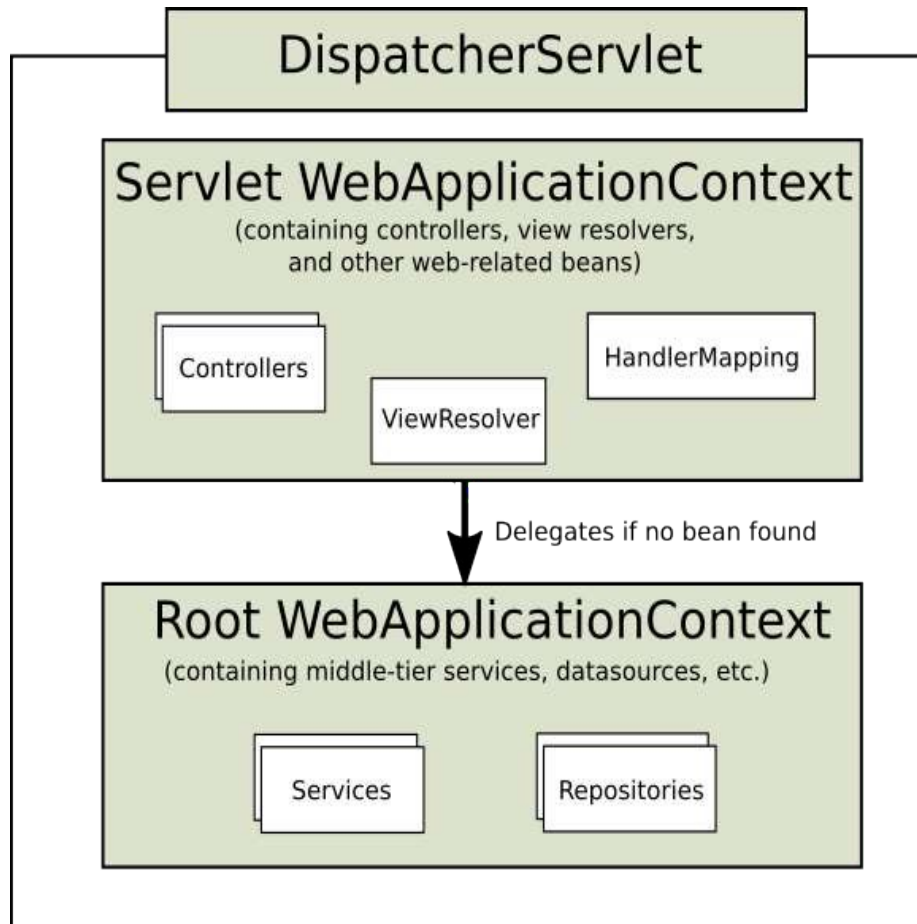
@Controller
@RequestMapping("/welcome")
public class HelloController {

    @RequestMapping(method = RequestMethod.GET)
    public String printWelcome(ModelMap model) {

        model.addAttribute("message", "Spring 3 MVC Hello World");
        return "hello";
    }
}
```

```
<meta http-equiv="Content-Type" content:
<title>Insert title here</title>
</head>
<body>
    ${message}
</body>
</html>
```

# WebApplicationContext vs RootApplicationinContext



Root Config Classes are actually used to Create Beans which are Application Specific and which needs to be available for Filters (As Filters are not part of Servlet).

Servlet Config Classes are actually used to Create Beans which are DispatcherServlet specific such as ViewResolvers, ArgumentResolvers, Interceptor, etc.

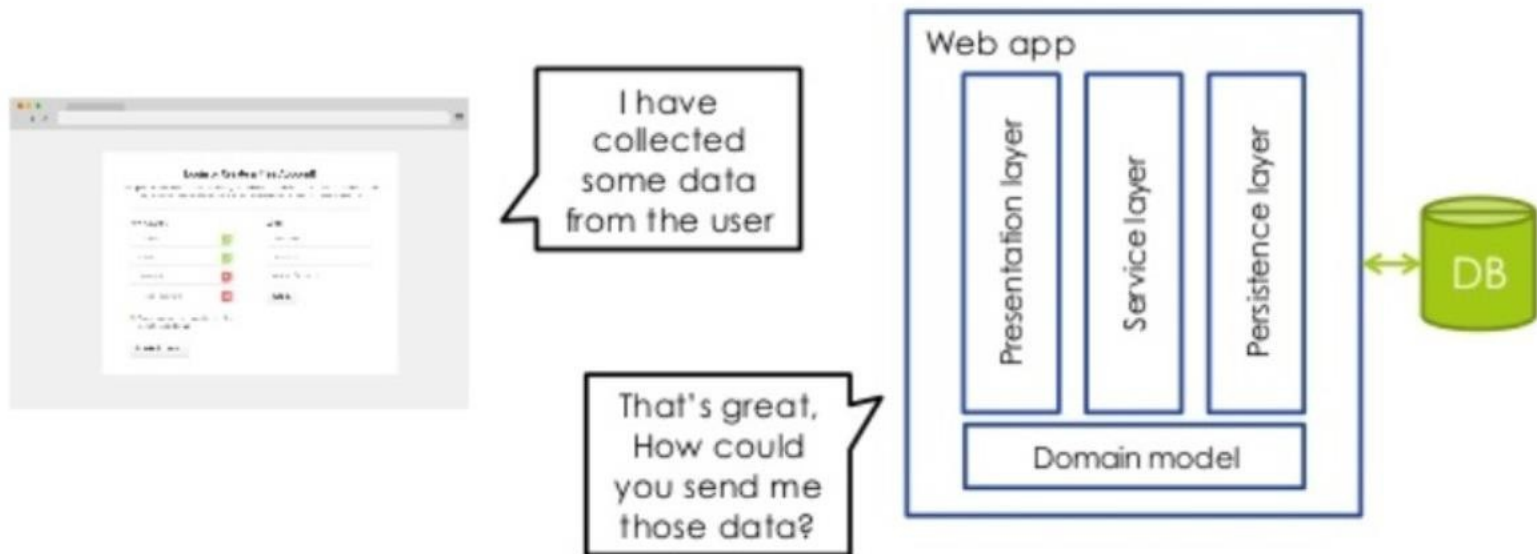
Root Config Classes will be loaded first and then Servlet Config Classes will be loaded.

Root Config Classes will be the Parent Context

MVC form processing

# Html form vs. web applications

- Html forms gives a place to enter data but do not provide space for web app to put such data



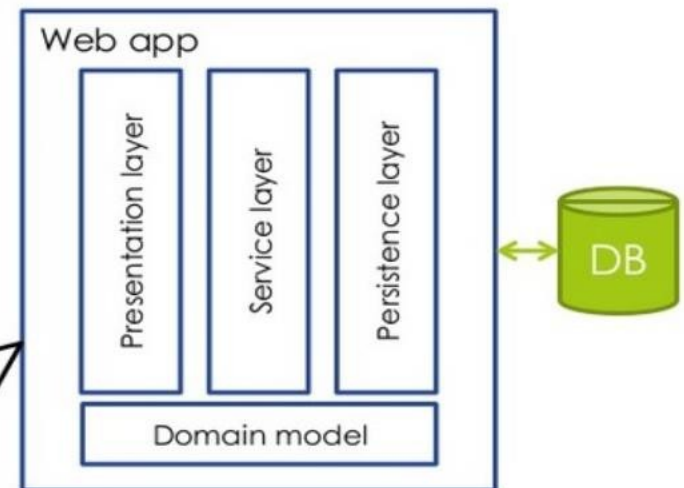
# Html form vs. web applications

- What happens when form submitted?
  - The browser send the data up to the server as a list of name value pair
  - Everything is going to be transferred to the web app as a String
- **HTTP/HTML does not provide a components that can buffer, validate and convert input coming from a form**
  - That is the way HTTP and HTML work, web applications can not control this



The best I can do is to send you everything as a list of Strings...

What? This means that I have to do all the data processing by myself





# Html form vs. web applications

---

- But, what if...
  - A field has to be interpreted as something different than a `String` (e.g., as a `Date`)? Conversion
  - The user forgets to provide a mandatory field? does he have to re-type everything from scratch? Buffering
  - We want to check that a field respects a given pattern? Validation
- When trying to solve these problems, HTML and HTTP are of **no use** to us

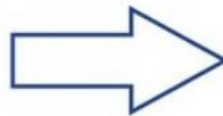
# Account Sign up example: Conversion

- What is the best way to move request parameter to the account object? (backing bean)

```
<spring:url value="/account" var="account"/>
<form action="${account}" method="post">
    Name:      <input type="text" name="name"/>    <br/>
    Surname:   <input type="text" name="surname"/> <br/>
    Email:     <input type="text" name="email"/>    <br/>
    Birthday:  <input type="text" name="birthday"/><br/>
               <input type="submit" value="Sign-up"/>
</form>
```

HTTP Request	
header	POST /account HTTP/1.1 Host: myserver.com User-Agent: ... Accept-Encoding: ...
body	name=John& surname=Smith& email=john@smith.com& birthday=10-1-1956

list of Strings



```
public class Account {

    private String name;
    private String surname;
    private String email;
    private Date birthday;

    // getters and setters

}
```


Account object

# Data binding

---

- Data binding is the process of "binding" the request parameters to a so called form bean/backing bean
- All we need to do is to declare an account object as a method parameter.

```
@RequestMapping("/account")  
public String addAccountFromForm(Account account) {  
    ...  
}
```



account will be automatically populated from the request parameters

- The following sequence of operations occurs:
  1. A new form bean is instantiated
  2. the form bean is added to the model
  3. the form bean is populated from the request parameters

# Account Sign up example: Data buffering (Pre populated values)

## □ Setting a default value for the form bean

- Assume that we want to ask the user for the permission of sending marketing e-mails
- To this end we add a marketingOk property in the Account form bean
- By default we want marketingOk to be checked
- We would like the registration page to use properties coming from a prepopulated Account bean

```
public class Account {  
  
    private String name;  
    private String surname;  
    private String email;  
    private Date birthday;  
    private boolean marketingOk = true;  
  
    // getters and setters  
}
```

We are prepopulating the Account bean

The diagram shows a registration form on the left and an 'Account bean' object on the right. A blue arrow points from the 'Account bean' to the form, indicating data prepopulation. The form contains the following elements:

- Name:
- Surname:
- Date:
- ☒ Please send me product updates via e-mail
-

# Revised registration form

- To deal with pre-populated form beans, Spring provides a set of data binding aware tags
- To use the tags from the form library, following directives need to be added to the top of JSP

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

modelAttribute binds the form to the account bean placed into the model

```
<form:form modelAttribute="account">
  Name:      <form:input path="name" />      <br/>
  Surname:   <form:input path="surname" />    <br/>
  Email:     <form:input path="email" />      <br/>
  Birthday:  <form:input path="birthday" />    <br/>

  <form:checkbox path="marketingOk" />
  Please send me product updates via e-mail <br/>

  <input type="submit" value="Sign-up" />
</form:form>
```

Each path attribute reference the property of the account bean

# MVC form validation

# Account Sign up example: Data validation

---

- To detect user's errors, we need to validate the form data that are encapsulated in the form bean
- Example: the email property should respect the pattern `foo@provider.com`

```
public class Account {  
  
    private String name;  
    private String surname;  
    private String email;  
    private Date birthday;  
    private boolean marketingOk = true;  
  
    // getters and setters  
}
```

- The **Bean Validation API** ([JSR-303](#)) is a specification that defines a metadata model and API for JavaBean validation
- Using this API, it is possible to **annotate** bean properties with **declarative validation constraints**
- **Examples:** `@NotNull`, `@Pattern`, `@Size`

# Adding constraints to the Account bean

```
public class Account {  
  
    @Pattern(regexp="^[A-Z]{1}[a-z]+$")  
    @Size(min=2, max=50)  
    private String name;  
  
    @Pattern(regexp="^[A-Z]{1}[a-z]+$")  
    @Size(min=2, max=50)  
    private String surname;  
  
    @NotBlank  
    @Email  
    private String email;  
  
    @NotNull  
    private Date birthday;  
    ...  
}
```

name and  
surname should  
start with a capital  
letter and have at  
least one  
additional  
lowercase letter

email must respect the  
username@provider.tld  
pattern

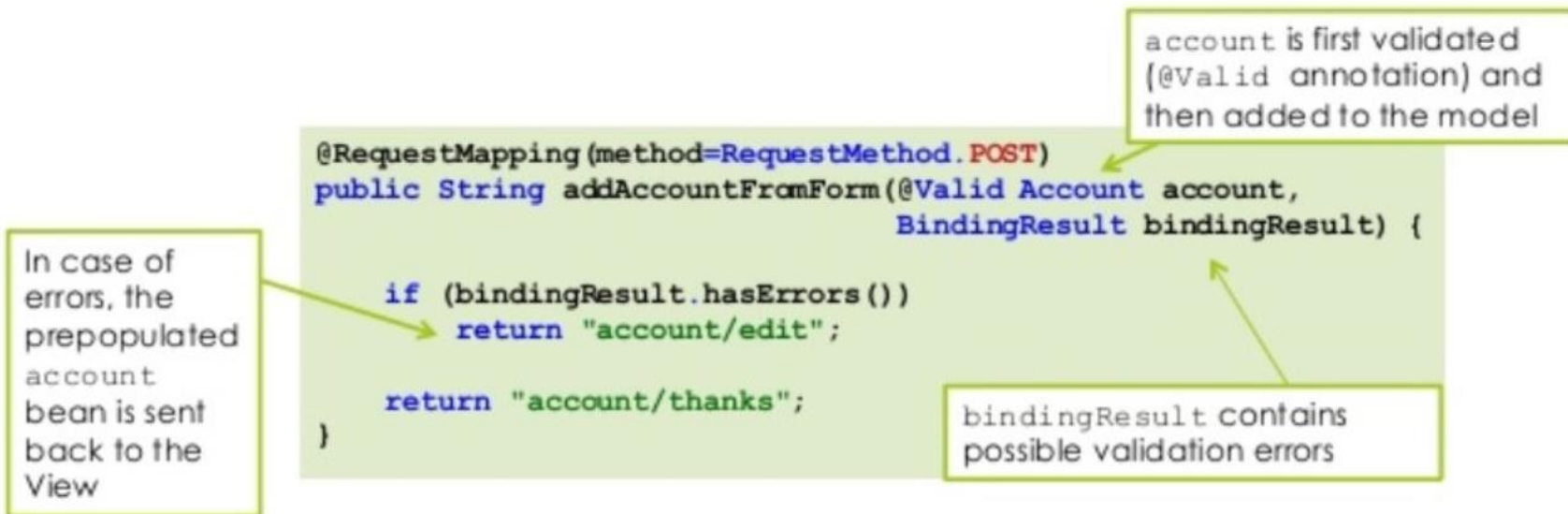
As with any other JEE API, the standard defines only the API specification

We are going to use **Hibernate Validator**, which is the reference implementation of the JSR-303 specification



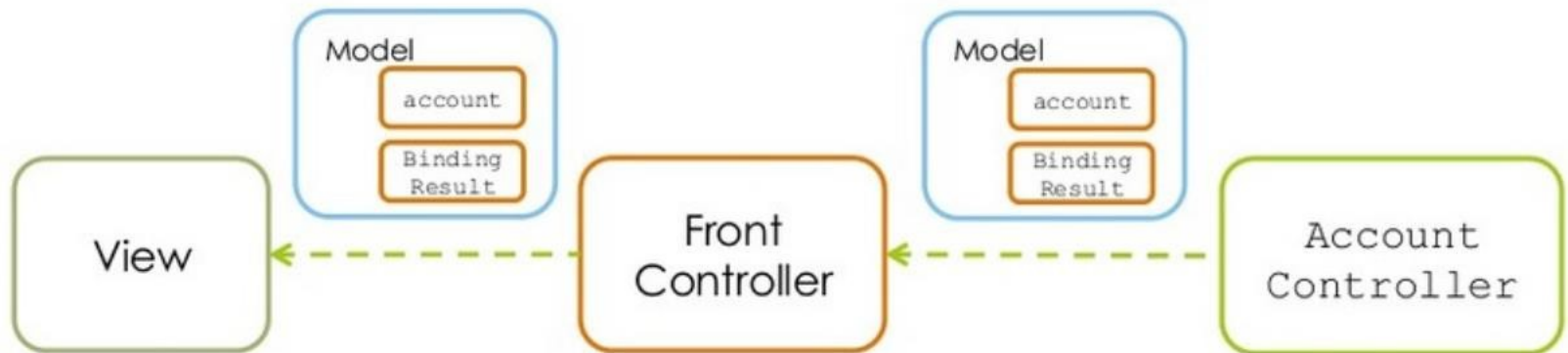
# Checking for validation errors

- together with the form bean, the handler method can now receive the result of the validation process



# BindingResult is the part of the model

- To this end the bindingResult object is automatically inserted into the model and send back to the view



# <form:errors>

---

## <form:errors>

Spring provides a <spring:errors> tag as part of the Spring's form tag library

The tag renders **error message** taken from the BindingResult object within a HTML <span> tag

```
<form:form modelAttribute="account">
Name:    <form:input path="name"/>    <form:errors path="name"/>    <br/>
Surname: <form:input path="surname"/> <form:errors path="surname"/> <br/>
Email:   <form:input path="email"/>   <form:errors path="email"/>   <br/>
...
</form:form>
```

```
public class Account {

    ...
    @NotNull(message = "the email address cannot be empty")
    @Email(message = "please provide a valid e-mail address")
    private String email;

    ...
}
```

# Resource Bundle

---

- A better alternative is to store the error messages in a separate file called the resource bundle
- By doing so, error messages can be updated independently from the source code ( loose coupling)

Resource bundle:

```
NotBlank.account.email=the email address cannot be empty  
Email.account.email=please provide a valid e-mail address  
NotNull.account.birthday=The date cannot be empty
```

Spring's convention dictates the following syntax for messages:

```
[ConstraintName].[ClassName].[FieldName]=[Message]
```

- We need to declare ReloadableResourceBundleMessageSource bean in order to load messages from a resource bundle

frontcontroller]-servlet.xml:

```
<bean id="messageSource"
      class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:validationMessages" />
</bean>
```

- The Spring DI Container will load the MessageSource

- **Remember:** the MessageSource bean **must** have the id equal to messageSource

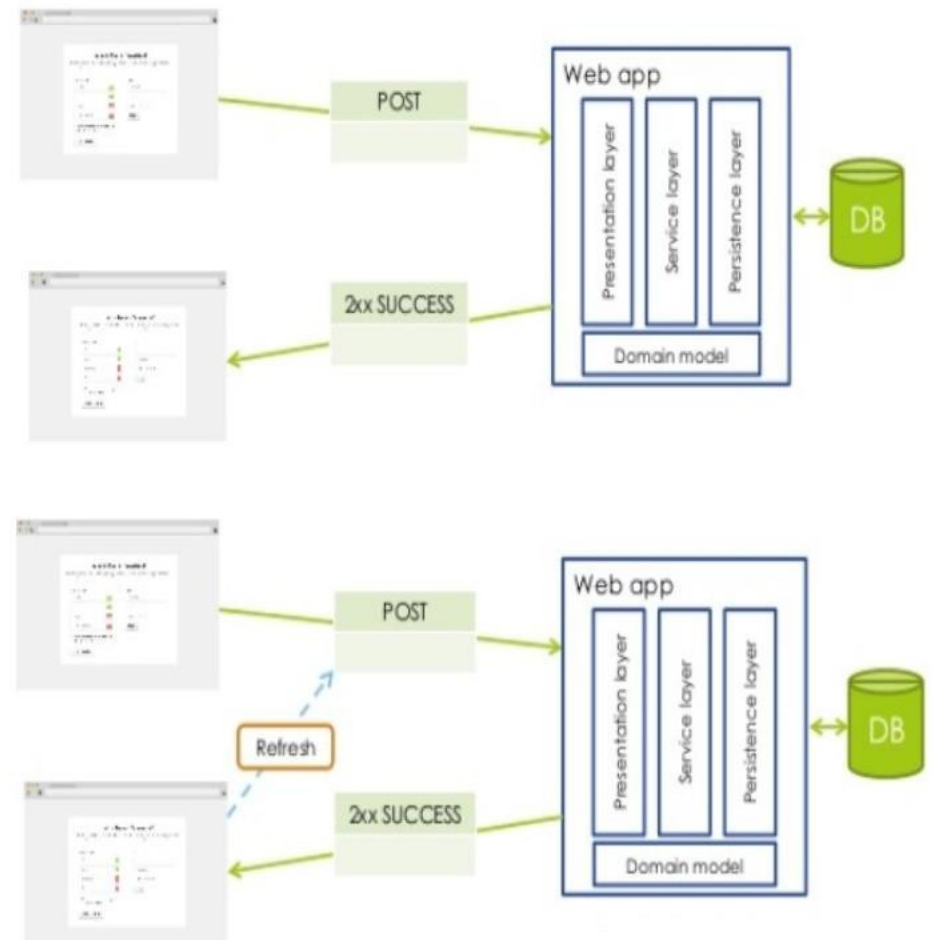
We can now specify the **message code**, rather than the message itself

```
public class Account {
    ...
    @NotBlank(message = "{NotBlank.account.email}")
    @Email(message = "{Email.account.email}")
    private String email;
    ...
}
```

PRG pattern

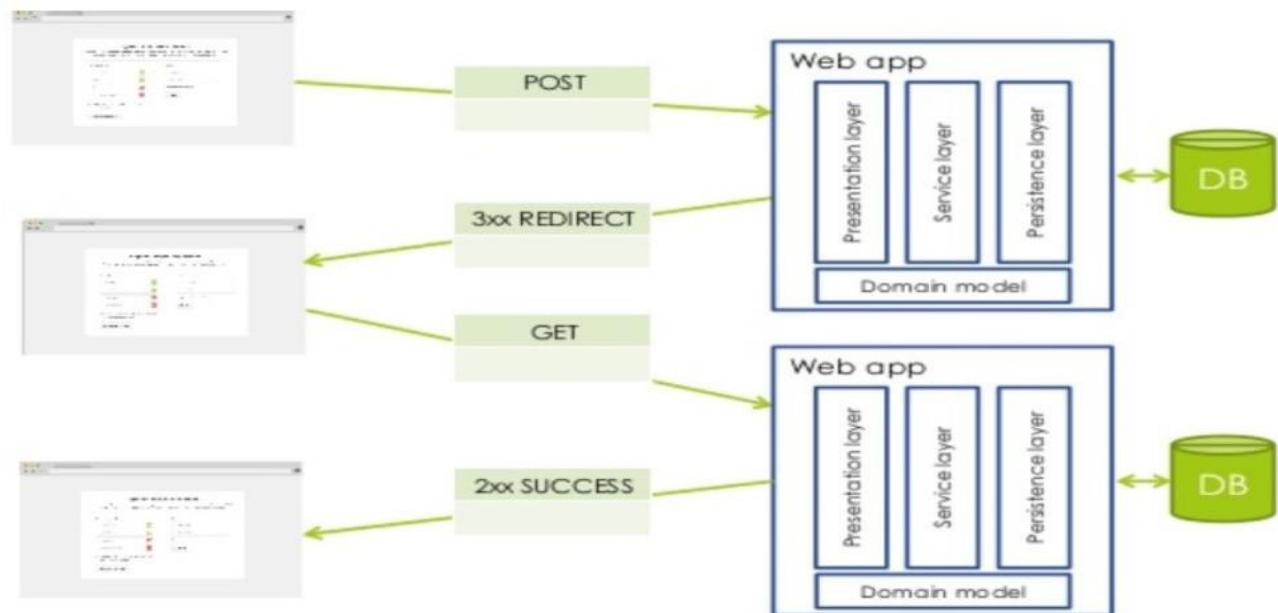
# Handling User input PRG pattern

- Web form are submitted to the server through http POST
- What if user press **refresh** on the browser?
- Double form submission



# The Post/Redirect/Get design pattern

- The PRG pattern solve the duplicate submission problem
  - According to the PRG pattern, the POST operation should not return web page directly, instead a redirect will be casted, causing an new GET operation to be executed
  - Upside: if the user refreshes the page, the GET request will be send, instead of original HTTP POST





# RedirectView **redirect:prefix**

- To force a redirect, a controller can return RedirectView instance
- RedirectView is a special view which redirects the user to a different URL, rather than rendering the view itself
- To return a RedirectView, it is sufficient to prefix the view name with label redirect

```
@RequestMapping(method=RequestMethod.POST)
public String addAccountFromForm(@Valid Account account,
                                BindingResult bindingResult) {

    if (bindingResult.hasErrors())
        return "account/edit";

    return "redirect:/account/thanks";
}
```

redirect: is a special indication that a redirect is needed

The rest of the view name is treated as the **redirect URL**

# RedirectView **redirect:prefix**

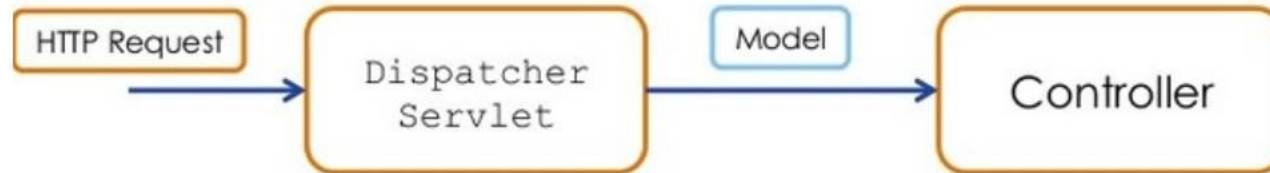
---

- **Remember : what follows the redirect:prefix is considered a url, not a view name**
- **that is /account/thanks will cause a GET request to http://myserver.com/webapp/account/thanks**
- hence we need a controller to handle the /account/thanks mapping
- The sole purpose of such a controller would be to return the /account/thanks view name
- That will cause the viewResolver to finally resolve the view name to /WEB-INF/account/thanks.jsp
- **Starting from Spring 3, it is possible to declarative set up controller whose unique purpose is to return a view name in FC-servlet.xml**

```
<mvc:view-controller path="/account/thanks"  
                    view-name="account/thanks"/>
```

# PRG pattern: Model attributes

- As the model contains the data to be rendered by the view, its lifetime is limited by the request/response lifecycle



- In other words, a new Model Object is created for each request that hits the DispatcherServlet

**Problem:** A redirect creates a new request, hence causing the model attributes to be discarded

What if we want to retain some model attributes?

# The flash Scope

---

- What if we want to retain some model attributes?
- **Solution?**
  - store attribute of interest in the flash scope

The **flash scope** works similarly to the session scope

The difference is that **flash attributes** are kept solely for the **subsequent request**

**Flash attributes** are stored before the redirect and made available as **model attributes** after the redirect

An handler method can declare an argument of type `RedirectAttributes`...

```
@RequestMapping(method=RequestMethod.POST)
public String addAccountFromForm(@Valid Account account,
                                BindingResult bindingResult,
                                RedirectAttributes redirectAttributes) {
```

...and use its `addFlashAttribute()` method to add attributes in the flash scope

```
@RequestMapping(method=RequestMethod.POST)
public String addAccountFromForm(@Valid Account account,
                                BindingResult bindingResult,
                                RedirectAttributes redirectAttributes) {

    if (bindingResult.hasErrors())
        return "account/edit";

    redirectAttributes.addFlashAttribute("name", account.getName());
    redirectAttributes.addFlashAttribute("surname", account.getSurname());
    return "redirect:/account/thanks";
}
```

name and surname will be automatically placed into the model object of the next request

they have been automatically inserted into the model

```
<html>
<head>
    <title>Thanks</title>
</head>
<body>
    Hi, ${name} ${surname}.
    You have been successfully registered. <br/>
</body>
</html>
```

# Spring amvc Java Configuration

## Step 1: cnfiguration for spring mvc bootstrap

```
1 package com.demo;
2
3 import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
4
5 public class WebInit extends AbstractAnnotationConfigDispatcherServletInitializer{
6
7     @Override
8     protected Class<?>[] getRootConfigClasses() {
9         return null;
10    }
11
12    @Override
13    protected Class<?>[] getServletConfigClasses() {
14        return new Class[] {MvcConfig.class};
15    }
16
17    @Override
18    protected String[] getServletMappings() {
19        return new String[] {"/"};
20    }
21
22 }
```



```
import org.springframework.web.servlet.view.InternalResourceViewResolver;
```

## Step 2: Configure viewresolver

```
@Configuration
```

```
@EnableWebMvc
```

```
@ComponentScan(basePackages = {"com.demo"})
```

```
public class MvcConfig extends WebMvcConfigurerAdapter{
```

```
    @Bean
```

```
    public InternalResourceViewResolver getInternalResourceViewResolver() {  
        InternalResourceViewResolver resolver = new InternalResourceViewResolver();  
        resolver.setPrefix("/WEB-INF/views/");  
        resolver.setSuffix(".jsp");  
        return resolver;  
    }
```

```
    @Override
```

```
    public void addResourceHandlers(ResourceHandlerRegistry registry) {  
        // Don't forget the ending "/" for location or you will hit 404.  
        registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");  
    }
```

```
}
```

## Step 3: Hello world controller

```
@Controller
```

```
public class HelloController {
```

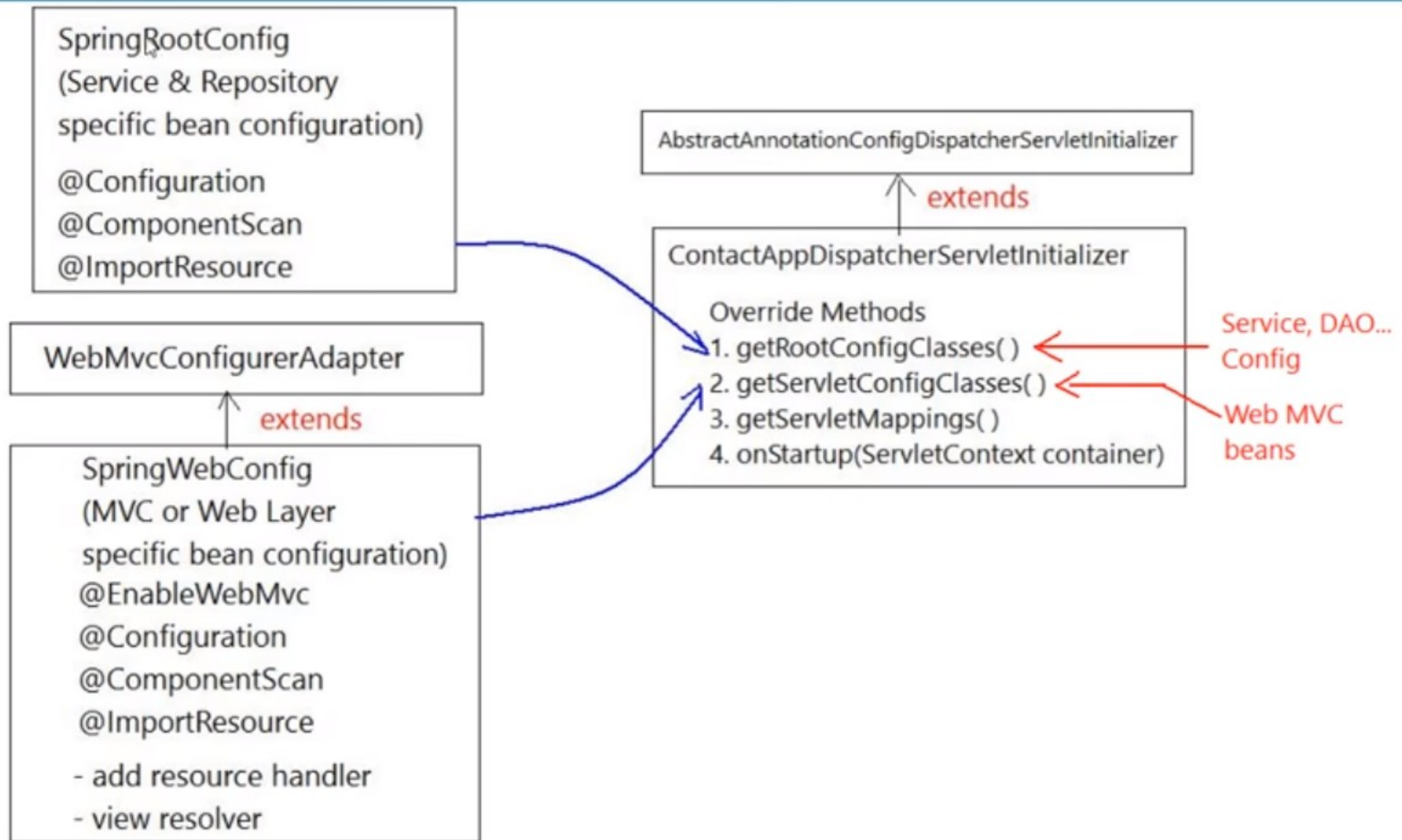
```
    @GetMapping(path = "hello")
```

```
    public ModelAndView hello(ModelAndView mv) {  
        mv.addObject("key", "hello to spring mvc");  
        mv.setViewName("demo");  
        return mv;  
    }
```

```
}
```



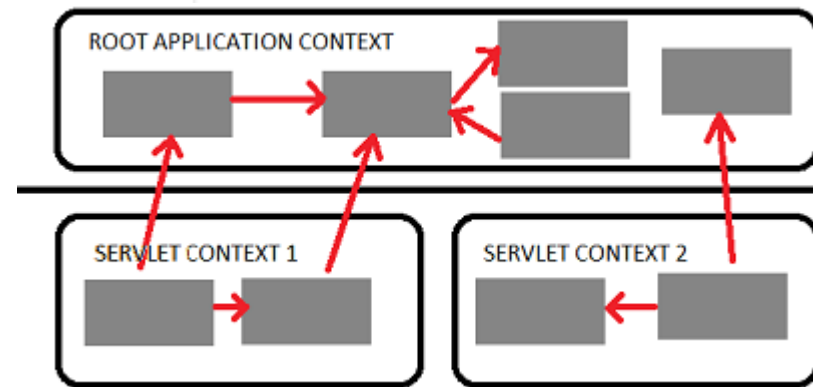
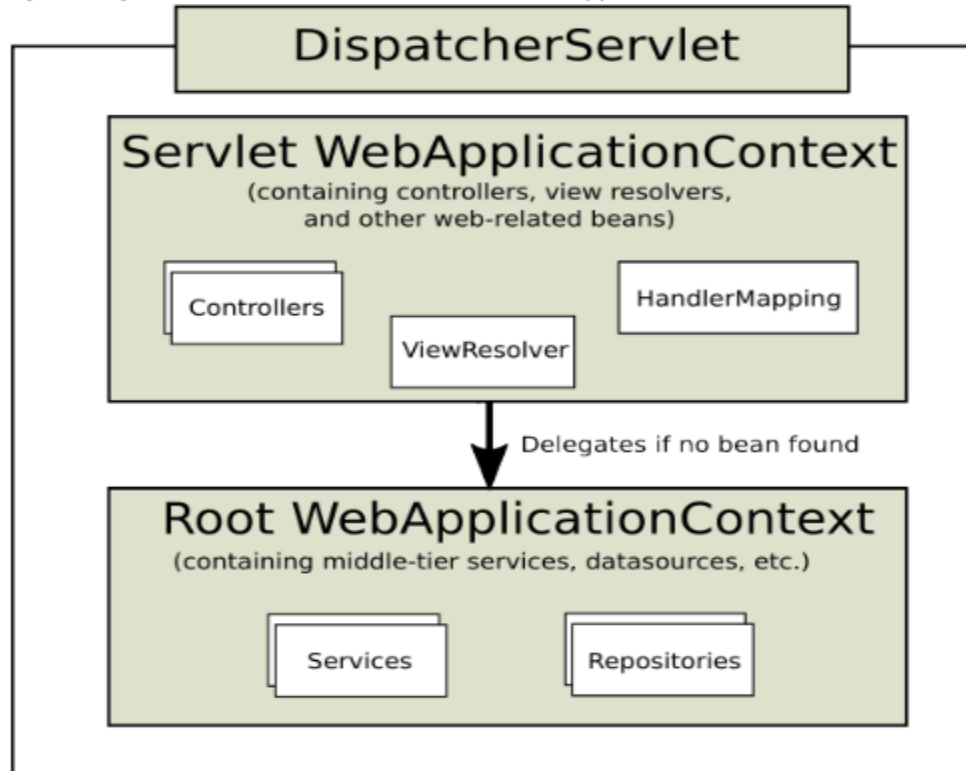
## Understanding Spring Web config



## WebApplicationContext vs applicationContext

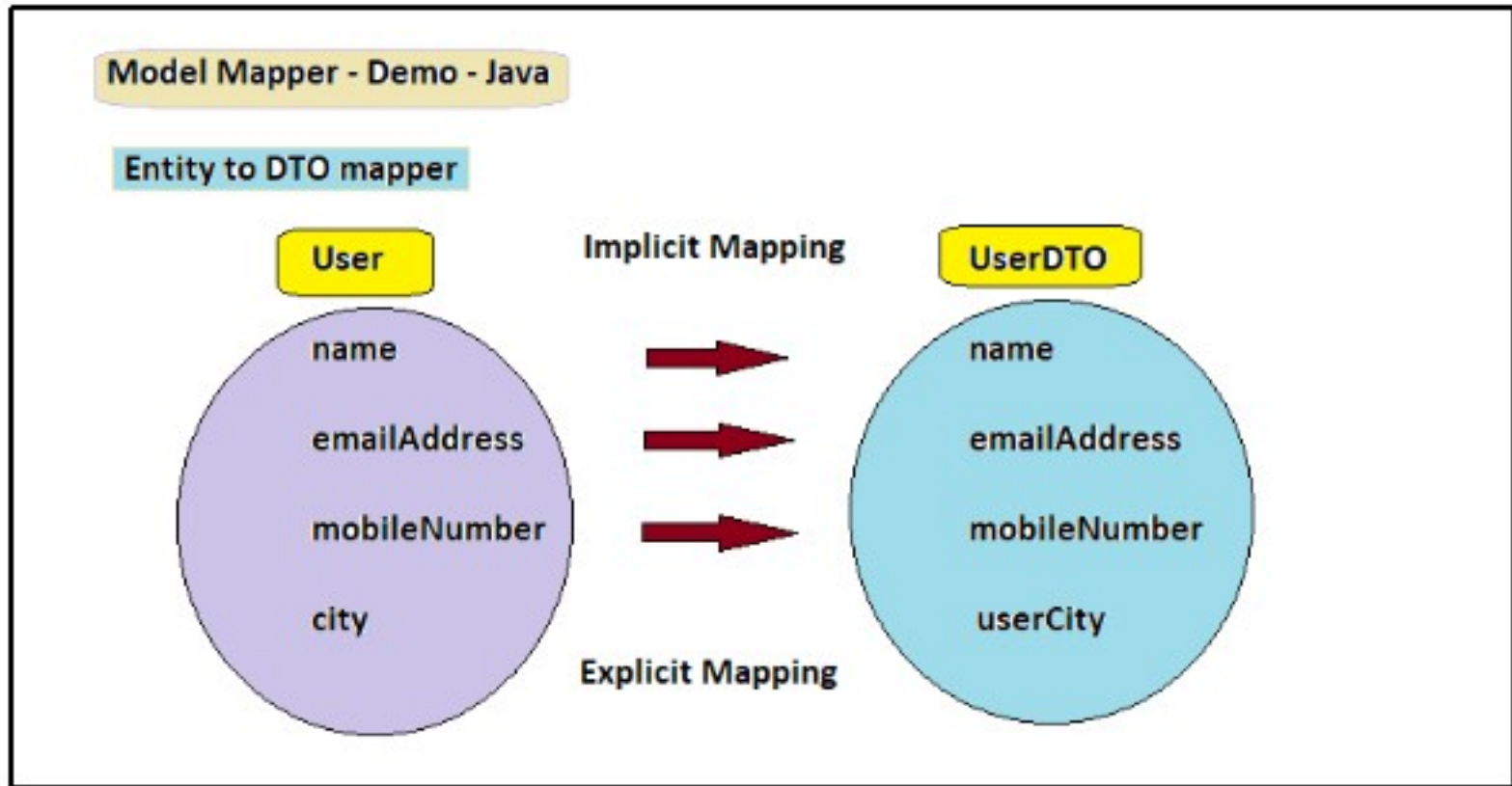
The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes (see Using themes), and that it knows which Servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it. Cited from [Spring web framework reference](#)

By the way servlet and root context are **both** webApplicationContext:



# Model mapper

# Model Mapping : Why required?



# Model Mapping :Example

We can use ModelMapper to implicitly map an user instance to a new UserDTO:

```
ModelMapper modelMapper = new ModelMapper();  
UserDTO userDTO = modelMapper.map(user, UserDTO.class);
```

## How It Works ?

When the map method is called, the source and destination types are analyzed to determine which properties implicitly match according to a matching strategy and other configuration.

ModelMapper will do its best to determine reasonable matches between properties.

If required we can also do the explicit mapping between properties.(inform the mapper about the properties explicitly)

```
modelMapper.addMappings(new PropertyMap<User, UserDTO>() {  
    protected void configure() {  
        map().setUserCity(source.getCity());  
    }  
});
```

```
<dependency>  
  <groupId>org.modelmapper</groupId>  
  <artifactId>modelmapper</artifactId>  
  <version>1.1.0</version>  
</dependency>
```

# Model Mapping : Why required?

