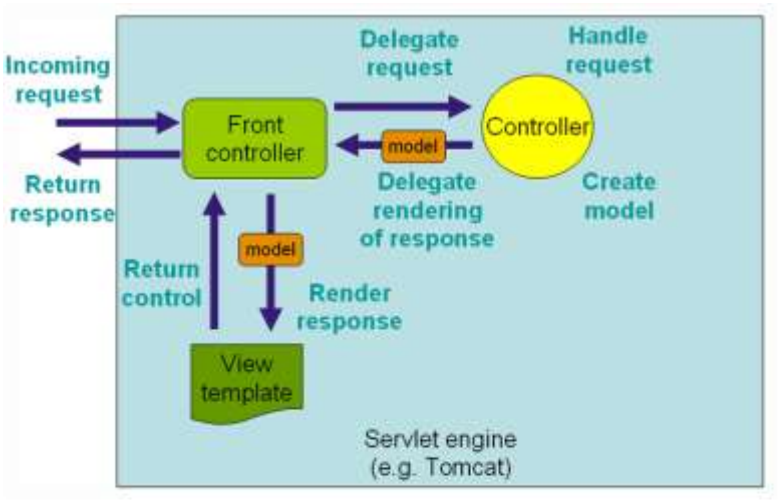
Introduction to Spring Web MVC framework

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale, time zone and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods

The DispatcherServlet

The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram.



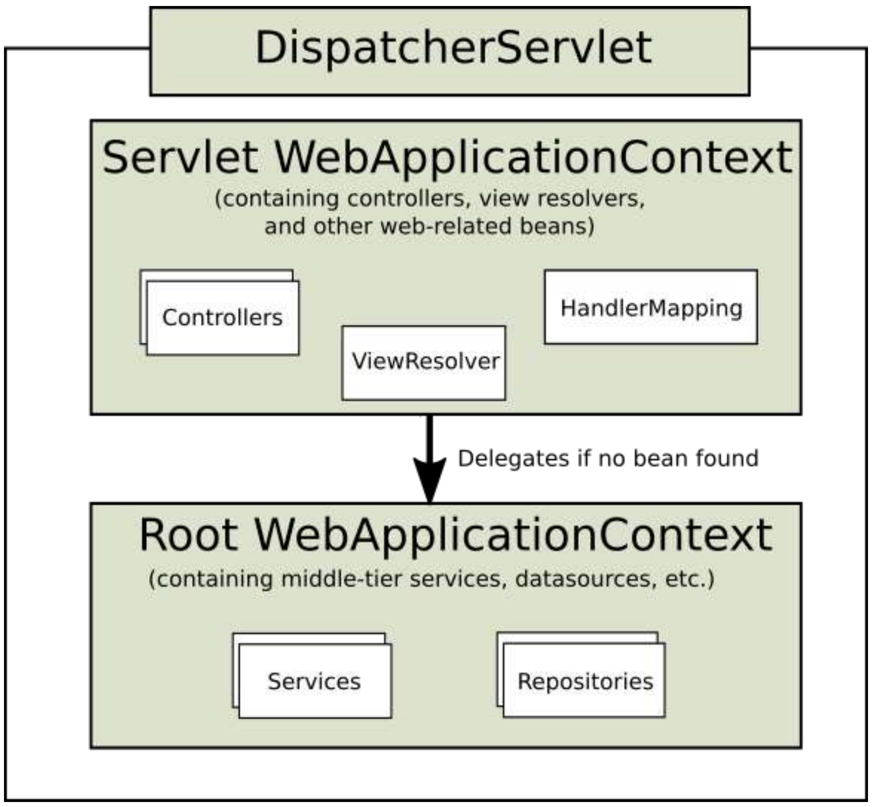
The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), and as such is declared in the web.xml of your web application. You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file.

```
<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>/example/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Typical context hierarchy in Spring Web MVC


DispatcherServlet
Servlet WebApplicationContext
(containing controllers, view resolvers, and other web-related beans)
Controllers
ViewResolver
HandlerMapping
Delegates if no bean found
Root WebApplicationContext
(containing middle-tier services, datasources, etc.)
Services
Repositories

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

Implementing Controllers

Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public String helloWorld(Model model) {
        model.addAttribute("message", "Hello World!");
        return "helloWorld";
    }
}
```

The `@Controller` annotation indicates that a particular class serves the role of a *controller*.

```
@Controller
@RequestMapping("/requestMethods")
public class RequestMethodController {

             @RequestMapping(method = RequestMethod.GET)
           public String get() {
               return "This is WebMethod Accessed Directly";
           }

           @RequestMapping(path = "/{day}", method = RequestMethod.GET)
           public String getForDay(@PathVariable String someParam, Model
model) {
               return "This Method accessed thru path and taken a additional
Parm :: " + someParam;
           }

}
```

Composed @RequestMapping Variants

Spring Framework 4.3 introduces the following method-level *composed* variants of the `@RequestMapping` annotation that help to simplify mappings for common HTTP methods and better express the semantics of the annotated handler method. For example, a `@GetMapping` can be read as a `GET` `@RequestMapping`.

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

```
@Controller
@RequestMapping("/composedController")
public class ComposedController {

   @GetMapping
    public String get() {
        return "This is WebMethod Accessed Directly";
    }

   @GetMapping(path = "/{day}")
    public String getForDay(@PathVariable String someParam, Model model) {
        return "This Method accessed thru path and taken a additional Parm
:: " + someParam;
    }

}
```

URI Template Patterns

URI templates can be used for convenient access to selected parts of a URL in a `@RequestMapping` method.

In Spring MVC you can use the `@PathVariable` annotation on a method argument to bind it to the value of a URI template variable:

```
@GetMapping("/owners/{ownerId}")
public String findOwner(@PathVariable String ownerId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    model.addAttribute("owner", owner);
    return "displayOwner";
}
```

A method can have any number of `@PathVariable` annotations:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public String findPet(@PathVariable String ownerId, @PathVariable String
petId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "displayPet";
}
```

URI Template Patterns with Regular Expressions

Sometimes you need more precision in defining URI template variables. Consider the URL `"/spring-web/spring-web-3.0.5.jar"`. How do you break it down into multiple parts?

```
@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-
{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")
public void handle(@PathVariable String version, @PathVariable String
extension) {
    // ...
}
```

Path Patterns

In addition to URI templates, the `@RequestMapping` annotation and all *composed* `@RequestMapping` variants also support Ant-style path patterns (for example, `/myPath/*.do`). A combination of URI template variables and Ant-style globs is also supported (e.g. `/owners/*/pets/{petId}`).

When a URL matches multiple patterns, a sort is used to find the most specific match.

Binding request parameters to method parameters with @RequestParam

Use the `@RequestParam` annotation to bind request parameters to a method parameter in your controller.

The following code snippet shows the usage:

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap
model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }
}
```

Resolving views

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.

View resolvers

ViewResolver	Description

AbstractCachingViewResolver	Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
XmlViewResolver	Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml.
ResourceBundleViewResolver	Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.
UrlBasedViewResolver	Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
InternalResourceViewResolver	Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolver javadocs for details.
VelocityViewResolver / FreeMarkerViewResolver	Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView ,respectively, and custom subclasses of them.
ContentNegotiatingViewResolver	Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

As an example, with JSP as a view technology, you can use UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over to the RequestDispatcher to render the view.

```
<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>
```

```
<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
```