**SPRING MVC**

**MVC is an abbreviation for a design pattern. What does it stand for and what is the idea behind it?**

Model–View–Controller is a software architectural pattern which is designed to decouple three components, each of them can be easily swapped with a different implementation, and together they provide a fully functional user interface.

The Spring Web MVC module of Spring Framework provides an MVC (Model-View-Controller) framework that you can use for developing servlet-based web applications. Spring Web MVC is a non-intrusive framework that provides a clear separation of concerns between application objects that form the web layer.

1) Model: holds the current data and business logic.

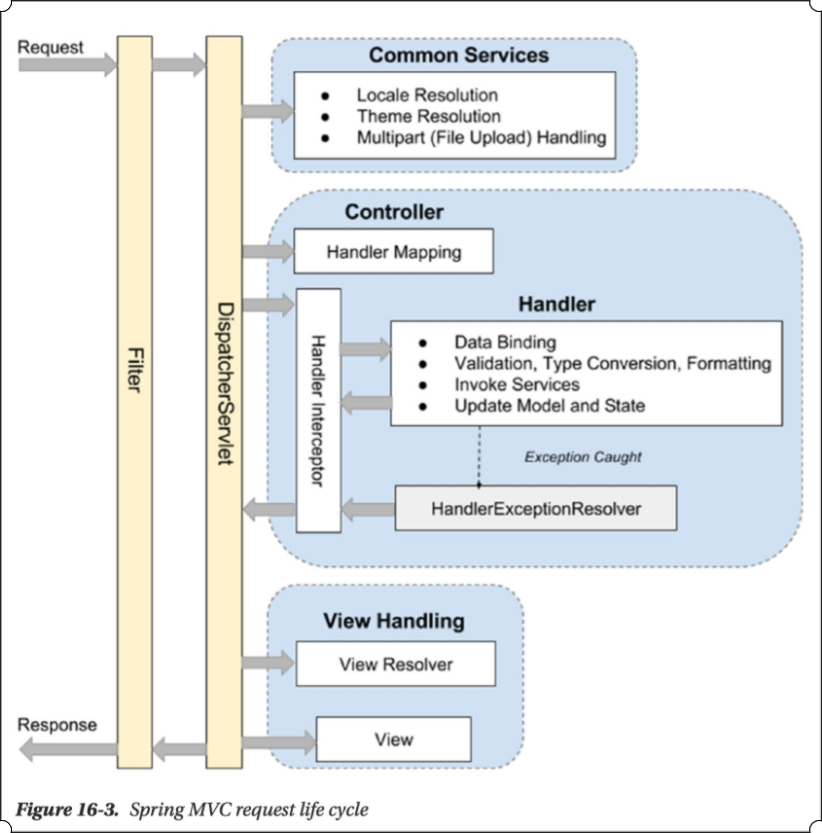
2) View: presenting data. User interacts with view.

3) Controller: accept requests from view, issues commands to model; manipulate data from model, interacts with view.

***Advantages:***

* Separation of concerns
* Decoupling among MVC
* Reuse of model and controllers with different views

***Spring MVC Request Life Cycle-***



1) Filter: The filter applies to every request. There are several commonly used filters.

2) Dispatcher servlet: The servlet analyzes the requests and dispatches them to the appropriate controller for processing. This is where DispatcherServlet implement Front Controller design pattern.

3) Common services: The common services will apply to every request to provide supports including i18n, theme, and file upload. Their configuration is defined in the DispatcherServlet’s WebApplicationContext.

4) Handler mapping: This maps incoming requests to handlers (a method within a Spring MVC controller class). Spring MVC will automatically register a HandlerMapping implementation maps handlers based on HTTP paths expressed through the @RequestMapping annotation at the type or method level within controller classes.

5) Handler interceptor: In Spring MVC, you can register interceptors for the handlers for implementing common checking or logic. For example, a handler interceptor can check to ensure that only the handlers can be invoked during office hours.

6) Handler exception resolver: to deal with unexpected exceptions thrown during request processing by handlers.

7) View Resolver: Spring MVC’s ViewResolver interface supports view resolution based on a logical name returned by the controller.

**Do you need spring-webmvc.jar in your classpath or is it part of spring-core?**

Yes, you do need this jar. It’s not part of Spring Core.

A typical Spring Web MVC project depends on spring-beans, spring-context, spring-core, spring-expression, spring-web and spring-webmvc JAR files. These JAR files are required for building a basic Spring Web MVC application

**What is difference between applicationContext.xml and spring-servlet.xml in XML configuration of Spring appllication?**

Spring lets you define multiple contexts in a parent-child hierarchy.

The applicationContext.xml defines the beans for the "root webapp context", i.e. the context associated with the webapp.

The spring-servlet.xml (or whatever else you call it) defines the beans for one servlet's app context. There can be many of these in a webapp, one per Spring servlet (e.g. spring1-servlet.xml for servlet spring1, spring2-servlet.xml for servlet spring2).

Beans in spring-servlet.xml can reference beans in applicationContext.xml, but not vice versa.

All Spring MVC controllers must go in the spring-servlet.xml context.

In most simple cases, the applicationContext.xml context is unnecessary. It is generally used to contain beans that are shared between all servlets in a webapp.

***<!-- Scans for annotated @Controllers in the classpath -->***

***<context:component-scan base-package="org.test.web">***

***<context:include-filter type="annotation" expression="org.springframework.stereotype.Controller"/>***

***</context:component-scan>***

In applicationcontext.xml we add filter for remaining packages like domain entities, DAOs and services excluding controller.

***<context:component-scan base-package="org.test">***

***<context:exclude-filter type="annotation" expression="org.springframework.stereotype.Controller"/>***

***</context:component-scan>***

The RootContextConfig class in Java config is equivalent to applicationContext.xml file (located in src/main/resources/META-INF/spring/)

WebContextConfig class in Java config is equivalent to spring-servlet.xml file that configures beans belonging to the web layer of the application like ViewResolver, HandlerMapping, Controller.

**What is the DispatcherServlet and what is it used for?**

Front controller pattern stands for a single servlet delegates responsibility for a request to other components of an application, to perform actual processing.

Following front controller pattern, Spring MVC provides DispatcherServlet receiving all the requests and delegates the processing to request handlers (controllers). Once the processing is done, ViewResolver will render a view based on the view name.

A Spring web application may define multiple dispatcher servlets, each of which has its own namespace, its own Spring application context and its own set of mappings and handlers.

***Used for --***

* Receives requests and delegates them to registered handlers
* Resolve views by mapping view-names to view instances
* Resolves exceptions

The below figure shows that the following sequence of activities are performed by Spring Web MVC during request processing:

1) Request is first intercepted by the DispatcherServlet servlet

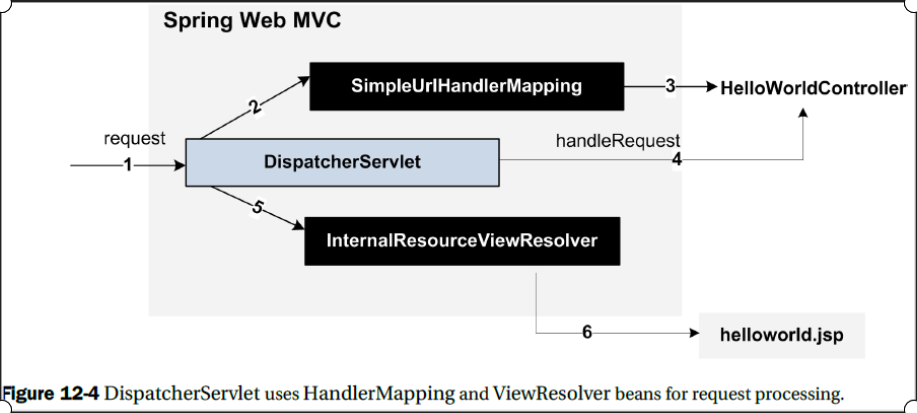
2)DispatcherServlet uses the HandlerMapping bean (which is SimpleUrlHandlerMapping bean in the case of ‘Hello World’ web application) to find an appropriate controller for handling the request

3) DispatcherServlet calls the request handling method of the controller (which is HelloWorldController’s handleRequest method in the case of ‘Hello World’ web application)

4) DispatcherServlet sends the view name returned by the controller to the ViewResolver bean (which is

InternalResourceViewResolver bean in the case of ‘Hello World’ web application) to find the actual view (JSP or servlet) to be rendered

5) DispatcherServlet dispatches the request to the actual view (JSP or servlet). The model data returned by the controller are made available to the view as request attributes.



***To enable Spring MVC within a web application -***

1) Configuring the root WebApplicationContext

2) Configuring the servlet filters required by Spring MVC

3) Configuring the dispatcher servlets within the application

4) Use Java to configure DispatcherServlet in the servlet container

***public class DemoWebAppInitializer***

***extends AbstractAnnotationConfigDispatcherServletInitializer {***

***@Override***

***protected String[] getServletMappings() {***

***return new String[] { "/" }; // Map DispatcherServlet to /***

***}***

***@Override***

***protected Class<?>[] getRootConfigClasses() {***

***return new Class<?>[] { SecurityConfig.class, DataServiceConfig.class };***

***}***

***@Override***

***protected Class<?>[] getServletConfigClasses() { // Specify configuration class***

***return new Class<?>[] { WebConfig.class };***

***}***

***@Override***

***protected Filter getServletFilters() {***

***CharacterEncodingFilter cef = new CharacterEncodingFilter();***

***cef.setEncoding("UTF-8");***

***cef.setForceEncoding(true);***

***return new Filter{ new HiddenHttpMethodFilter(), cef};***

***}***

***}***

To make things more practical, Spring class AbstractAnnotationConfigDispatcherServletInitializer, an implementation of WebApplicationInitializer, was extended because it contains concrete implementations of methods needed for the configuration of Spring web applications that use Java-based Spring configuration.

***getRootConfigClasses():*** A root application context of type AnnotationConfigWebApplicationContext will be created.

***getServletConfigClasses():*** A web application context of type AnnotationConfigWebApplicationContext will be created

***getServletMappings():*** The DispatcherServelt’s mappings (context) are specified by the array of strings returned by this method.

***getServletFilters():*** As the name of the methods says, this one will return an array of implementations of javax.servlet.Filter that will be applied to every request By providing an empty class that extends AbstractSecurityWebApplicationInitializer, you are basically telling Spring that you want DelegatingFilterProxy enabled, so springSecurityFilterChain will be used before any other registered javax.servlet.Filter.

Any class that extends AbstractAnnotationConfigDispatcherServletInitializer will automatically be used to configure DispatcherServlet and the Spring application context in the application’s servlet context.

This initializer create a DispatcherServlet and a ContextLoaderListener.

getServletMappings() identifies one or more paths that DispatcherServlet will be mapped to. It will handle all requests coming into the application.

getRootConfigClasses() is called internally, and the configuration classes are used to create the root application context, which will become the parent ApplicationContext that contains bean definitions shared by all child (DispatcherServlet) contexts.

***Spring Boot -***

In Spring Boot The spring-boot-starter-web starter by default configures DispatcherServlet to the URL pattern “/” and adds Tomcat as the embedded servlet container, which runs on port 8080. Spring Boot by default serves the static resources (HTML, CSS, JS, images, etc.) from the following CLASSPATH locations:

* /static
* /public
* /resources
* /META-INF/resources

**Is the DispatcherServlet instantiated via an application context?**

The DispatcherServlet is not instantiated via an application context. It is instantiated before any application context is created. parent ApplicationContext is created by ContextLoaderListener, child ApplicationContext is created by Spring MVC DispatcherServlet.

***Spring MVC WebApplicationContext Hierarchy-***

***Parent ApplicationContext*** - It is also called RootApplicationContext.

In a web application, parent ApplicationContext is usually created using org.springframework.web.context.ContextLoaderListener. it includes the application-level configurations such as the back-end data source, security, and service and persistence layer configuration. Say, it contains all non-web beans. It’s available to all servlet-level WebApplicationContexts.

***Child ApplicationContext -***

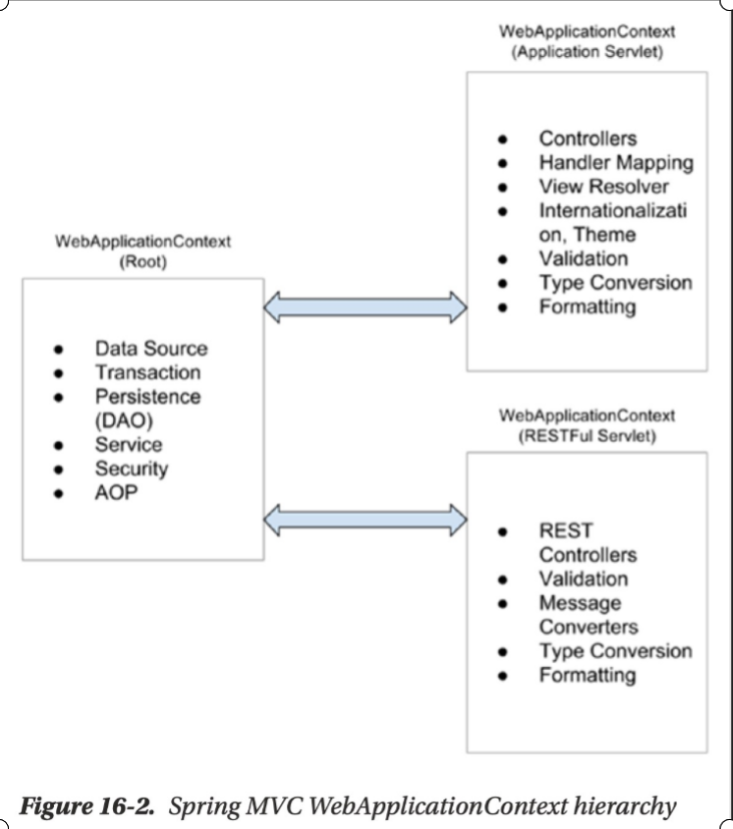
1) It is also called the web context or the DispatcherServletContext.

2) It is created by Spring MVC DispatcherServlet.

3) Beans in the web context can access the beans in the parent context, but not conversely.

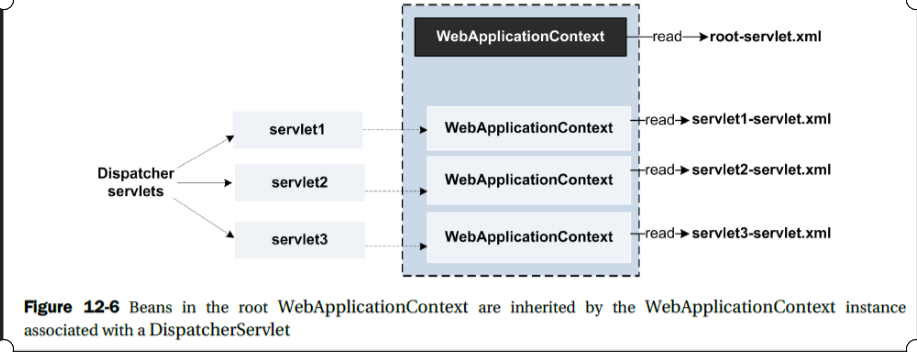
4) We can have two DispatcherServlet instances in an application.

One servlet supports the user interface (called the application servlet), and the other provides services in the form of RESTful-WS to other applications (called the RESTful servlet).



If your web application consists of multiple modules, you may define a DispatcherServlet for each of the modules in the web.xml file. In such a scenario, each DispatcherServlet has its own web application context XML file that contains beans (like controllers, view resolvers, and so on) specific to that module. You should note that these beans are not shared between DispatcherServlet instances. The beans that are shared between DispatcherServlet instances are defined in the root web application context XML file. As mentioned earlier, the root web application context XML file defines data sources, services and DAOs, and so on, that are typically shared by different modules of a web application.

The following figure shows relationship between beans defined by the web application context XML file associated with a DispatcherServlet and the beans defined by the root web application context XML file.



In the above figure, servlet1, servlet2 and servlet3 are the names of DispatcherServlet instances configured in the web.xml file. And, servlet1-servlet.xml, servlet2-servlet.xml and servlet3-servlet.xml are web application context XML files that are loaded by servlet1, servlet2 and servlet3, respectively. When DispatcherServlet instances are initialized, an instance of WebApplicationContext is created corresponding to each servlet1-servlet.xml, servlet2-servlet.xml and servlet3-servlet.xml files and associated with the DispatcherServlet instance. A WebApplicationContext instance is also created corresponding to the root web application context XML file, root-servlet.xml. The beans contained in the root WebApplicationContext instance are available to all the WebApplicationContext instances associated with DispatcherServlets.

**What is a web application context? What extra scopes does it offer?**

At the time of initialization, a DispatcherServlet loads the corresponding web application context XML file and creates an instance of Spring’s WebApplicationContext object. WebApplicationContext is a sub-interface of ApplicationContext interface that provides features that are specific to web applications. In addition to the standard Spring bean scopes singleton and prototype, there are four additional scopes available in a web application context - request, session, application and websocket.

1) ***Request -*** Spring container creates a new bean instance for every HTTP request. The bean instance is destroyed by the Spring container when the HTTP request completes.

2) ***Session -*** Spring container creates a new bean instance when an HTTP Session is created. The bean instance is destroyed by the Spring container when the HTTP Session is destroyed.

3) ***Application*** - Spring container creates a new bean instance when the ServletContext is created, and destroys it when the ServletContext is destroyed.

4) ***Websocket -*** Spring container creates a new bean instance when the WebSocket session is created, and is destroyed when the WebSocket session is destroyed.

Above scopes are valid only for ApplicationContext implementations that are applicable in web application scenarios. For instance, if you are using XmlWebApplicationContext or AnnotationConfigWebApplicationContext.

You can think of WebApplicationContext object as an object that represents a Spring container instance in Spring Web MVC applications. Basically, WebApplicationContext is a Spring application context for web applications.

The beans that are registered within the WebApplicationContext can also access the ServletContext or ServletConfig by implementing the ServletContextAware and ServletConfigAware interfaces respectively.

**What is the @Controller annotation used for?**

Spring Web MVC provides classes, like MultiActionController, UrlFilenameViewController, AbstractController, and so on, that you can extend to create your controllers. If you extend a Spring-specific class or implement a Spring-specific interface to create a controller, the controller class becomes tightly coupled with Spring. Spring defines annotations like @Controller, @RequestMapping, @ModelAttribute, and so on, that allow you to create controllers with flexible method signatures.

In XML configuration, <mvc:annotation-driven> element of Spring’s mvc schema enables use of Spring Web MVC annotations in implementing controllers. However, In Java-based configuration approach, the @EnableWebMvc annotation serves the same purpose as the <annotation-driven> element of Spring’s mvc schema.

**How is an incoming request mapped to a controller and mapped to a method?**

The Dispatcher Servlet scans classes annotated with @Controller annotation to map the web requests to the methods annotated with @RequestMapping, which are mapped to a certain request URL.

The @RequestMapping annotation is used at the type and method-level to map requests to controllers and its methods.

The following example listing shows @RequestMapping annotation usage in SomeController (a Spring Web MVC controller) class:

***@Controller***

***@RequestMapping("/type\_Level\_Url")***

***public class SomeController {***

***@RequestMapping("/methodA\_Url")***

***public ModelAndView methodA() { ..... }***

***@RequestMapping("/methodB\_Url")***

***public ModelAndView methodB() { ..... }***

***}***

The <annotation-driven> element of Spring’s mvc schema creates an instance of RequestMappingHandlerMapping (a HandlerMapping implementation) that is responsible for mapping a web request to an appropriate @RequestMapping annotated method.

RequestMappingHandlerMapping considers controller methods as endpoints, and is responsible for uniquely mapping a request to a controller method based on the @RequestMapping annotations at type- and method-level. In the case of SomeController, if the request path is /type\_Level\_Url/methodA\_Url, methodA is invoked, and if the request path is /type\_Level\_Url/methodB\_Url, methodB is invoked. You should note that if a request cannot be mapped uniquely to a controller method, then a HTTP 404 (which means, resource not found) status code is returned.

The attributes of @RequestMapping annotation are used to narrow down the mapping of a request to a particular controller or a controller method. You can specify these attributes at both type- and method-level @RequestMapping annotations.

1) Mapping requests based on request path

***@Controller***

***@RequestMapping("/myUrl/\*\*")***

***public class SomeController { ..... }***

2) Mapping requests based on HTTP methods

***@RequestMapping(path = "/list", method = RequestMethod.GET)***

***public ModelAndView listFixedDeposits() { ..... }***

3) Mapping requests based on request parameters

***@RequestMapping(params = "fdAction=createFDForm", method = RequestMethod.POST)***

***public ModelAndView showOpenFixedDepositForm() { ..... }***

4) Mapping requests based on the MIME type of the request

***@RequestMapping(consumes = "application/json")***

***public void perform() { ..... }***

***@RequestMapping(produces = "application/json")***

***public void perform() { ..... }***

5) Mapping requests based on a request header value

***@RequestMapping(headers = "Content-Type=text/plain")***

***public void perform() { ..... }***

**What is the difference between @RequestMapping and @GetMapping?**

@GetMapping equals to `@RequestMapping(method = RequestMethod.GET)`

Instead of using the generic @RequestMapping annotation, you can use HTTP method-specific annotations, like @GetMapping, @PostMapping, @PutMapping, and so on. These HTTP method-specific annotations are created by using @RequestMapping as a meta-annotation.

Custom annotations that are created by using existing annotations as meta-annotations are referred to as composed annotations. The HTTP method-specific annotations, like @GetMapping, @PostMapping, and so on, are examples of composed annotations.

The following example listing shows how the @GetMapping annotation is defined:

***@Target(ElementType.METHOD)***

***@Retention(RetentionPolicy.RUNTIME)***

***@Documented***

***@RequestMapping(method = RequestMethod.GET)***

***public @interface GetMapping {***

***.....***

***}***

Notice that the @GetMapping annotation is meta-annotated with @RequestMapping annotation and the method attribute is explicitly set to RequestMethod.GET.

**What is @RequestParam used for?**

We can pass HttpServletRequest object to a controller method and use it to retrieve request parameters. Instead of passing HttpServletRequest object to a controller method, you can annotate a method argument with @RequestParam annotation to assign value of a request parameter to the method argument.

You should note that the @RequestParam annotation can only be used if the method is annotated with @RequestMapping or @ModelAttribute annotations.

The following example listing shows FixedDepositController’s closeFixedDeposit method that is invoked when a user clicks the Close button to close a fixed deposit:

***public class FixedDepositController {***

***@RequestMapping(params = "fdAction=close", method = RequestMethod.GET)***

***public String closeFixedDeposit(@RequestParam(value = "fixedDepositId") int fdId) {***

***fixedDepositService.closeFixedDeposit(fdId);***

***return "redirect:/fixedDeposit/list";***

***}***

***}***

@RequestParam’s name (alias for value attribute) attribute specifies the name of the request parameter whose value is assigned to the method argument. In the above example listing, @RequestParam annotation is used to assign the value of fixedDepositId request parameter to fdId method argument. As the type of the fdId argument is int, Spring is responsible for converting the fixedDepositId request parameter to int type.

By default, method parameters that use this annotation are required, but you can specify that a method parameter is optional by -

1) Setting the @RequestParam annotation’s required flag to false or

2) By declaring the argument with an java.util.Optional wrapper.

Type conversion is automatically applied if the target method parameter type is not String or custom Java types (like Address).

***NB.*** use of @RequestParam is optional. By default, any argument that is a simple value type (as determined by BeanUtils#isSimpleProperty) and is not resolved by any other argument resolver, is treated as if it were annotated with @RequestParam.

To pass all the request parameters to a controller method, define an argument of type Map<String, String> or MultiValueMap<String, String> (an object provided by Spring that implements java.util.Map interface) and annotate it with @RequestParam annotation.

**What are the differences between @RequestParam and @PathVariable?**

@PathVariable instructs Spring MVC to bind the path variable within the URL - for example, ***http:// localhost:8080/singer/1*** into the id argument of the findSingerById() method. - Note that for the id argument, the type is Long, while Spring’s type conversion system will automatically handle the conversion from String to Long for us.

***@RequestMapping(value = "/{userId}", method = RequestMethod.GET)***

***public String findSingerById(@PathVariable("userId") Long id, Model model) {***

***// ...***

***}***

In URL below, firstName and lastName are request parameters

http://localhost:8080/greeting?firstName=dammy&lastName=good

In URL below, firstName and lastName are path parameters (path variables)

http://localhost:8080/firstname/dammy/lastname/good

**What are some of the parameter types for a controller method (@RequestMapping annotated method)?**

@RequestMapping annotated methods can have flexible method signatures. The argument types that can be passed to @RequestMapping annotated methods include --

* javax.servlet.ServletRequest,
* javax.servlet.ServletResponse,
* javax.servlet.http.HttpSession,
* java.security.Principal,
* HttpMethod,
* org.springframework.validation.BindingResult,
* org.springframework.web.bind.support.SessionStatus,
* org.springframework.ui.Model,
* org.springframework.ui.ModelMap,
* java.util.Map,
* java.util.Locale
* Errors,
* java.io.InputStream, java.io.Reader,
* java.io.OutputStream, java.io.Writer,
* UriComponentsbuilder

and so on. To view a complete list of arguments that can be passed to @RequestMapping annotated method, please refer to @RequestMapping Javadoc.

JDK 8’s java.util.Optional is supported as a method argument in combination with annotations that have a required attribute (for example, @RequestParam, @RequestHeader, and others) and is equivalent to required=false.

Reactive types are not supported for any arguments.

**What are some of the return types for a controller method (@RequestMapping annotated method)?**

The supported return types for @RequestMapping annotated methods include --

* HttpEntity,
* ResponseEntity,
* HttpHeaders,
* ModelAndView,
* org.springframework.web.servlet.View,
* String,
* java.util.Map,
* java.util.concurrent.Callable,
* void,
* ListenableFuture,
* CompletableFuture | CompletionStage: Asynchronous

and so on. To view a complete list of return types supported for @RequestMapping annotated methods, refer to @RequestMapping Javadoc.

Reactive types are supported for all return values.

**What annotations might you use on a controller method parameter?**

* @PathVariable
* @MatrixVariable
* @RequestParam
* @CookieValue
* @RequestBody
* @RequestHeader
* @ModelAttribute
* @SessionAttribute
* @SessionAttributes
* @RequestAttribute

**How @ModelAttribute and @SessionAttributes annotations are useful when dealing with model attributes?**

@ModelAttribute annotation is used for adding and retrieving model attributes to and from Spring’s Model object. The model attributes returned by a @RequestMapping method are stored in Spring’s Model object. A model attribute may represent a form backing object or a reference data.

@ModelAttribute annotation is used on methods and method arguments to store and retrieve model attributes from Spring’s Model object, respectively.

1) Adding model attributes using method-level @ModelAttribute annotation

***@Controller***

***@RequestMapping(path = "/fixedDeposit")***

***.....***

***public class FixedDepositController {***

***private static Logger logger = LogManager.getLogger(FixedDepositController.class);***

***.....***

***@ModelAttribute(name = "newFixedDepositDetails")***

***public FixedDepositDetails getNewFixedDepositDetails() {***

***FixedDepositDetails fixedDepositDetails = new FixedDepositDetails();***

***fixedDepositDetails.setEmail("You must enter a valid email");***

***logger.info("getNewFixedDepositDetails() method: Returning a new instance of***

***FixedDepositDetails");***

***return fixedDepositDetails;***

***}***

***.....***

***}***

The getNewFixedDepositDetails method creates and returns a new instance of FixedDepositDetails object. As the getNewFixedDepositDetails method is annotated with @ModelAttribute annotation, the returned FixedDepositDetails instance is added to the Model object. @ModelAttribute’s name attribute (alias for value attribute) specifies that the returned FixedDepositDetails object is stored with name newFixedDepositDetails in the Model object.

Note that the scope of model attributes is request. This means that the model attributes are lost when a request completes, or if a request is redirected.

As @RequestMapping and @ModelAttribute annotated methods can accept Model objects as argument, you can directly add model attributes to the Model object in a @ModelAttribute or @RequestMapping annotated method.

***public class SampleWebController {***

***@ModelAttribute***

***public void doSomething(Model model) {***

***model.addAttribute("myobject", new MyObject());***

***model.addAttribute("otherobject", new OtherObject());***

***}***

***}***

In the above example listing, the Model object is passed as an argument to doSomething method that directly adds model attributes to the Model object. As the doSomething method adds model attributes directly to the Model object, the doSomething method’s return type is specified as void, and the @ModelAttribute’s name attribute is not specified.

When a method is annotated with both @RequestMapping and @ModelAttribute annotations, the value returned by the method is considered as a model attribute, and not as a view name. In such a scenario, view name is determined by Spring’s RequestToViewNameTranslator class that determines the view to render based on the request URI of the incoming request.

It is important to note that you can define multiple methods annotated with @ModelAttribute annotation in a controller. When a request is dispatched to a @RequestMapping annotated method of a controller, all the @ModelAttribute annotated methods of that controller are invoked before the @RequestMapping annotated method is invoked.

2) Retrieving model attributes using @ModelAttribute annotation on arguments of a @RequestMapping annotated method. You can use @ModelAttribute annotation on arguments of a @RequestMapping annotated method to retrieve model attributes from the Model object.

The following example listing shows FixedDepositController’s openFixedDeposit method that uses @ModelAttribute annotation to retrieve newFixedDepositDetails object from the Model object:

***@Controller***

***@RequestMapping(path = "/fixedDeposit")***

***.....***

***public class FixedDepositController {***

***.....***

***@ModelAttribute(name = "newFixedDepositDetails")***

***public FixedDepositDetails getNewFixedDepositDetails() {***

***.....***

***logger.info("getNewFixedDepositDetails() method: Returning a new instance of***

***FixedDepositDetails");***

***.....***

***}***

***.....***

***@RequestMapping(params = "fdAction=create", method = RequestMethod.POST)***

***public String openFixedDeposit(***

***@ModelAttribute(name = "newFixedDepositDetails")***

***FixedDepositDetails fixedDepositDetails,.....) {***

***.....***

***fixedDepositService.saveFixedDeposit(fixedDepositDetails);***

***logger.info("openFixedDeposit() method: Fixed deposit details successfully saved.***

***Redirecting to show the list of fixed deposits.");***

***.....***

***}***

***}***

In the above example listing, @ModelAttribute annotated getNewFixedDepositDetails method is invoked before @RequestMapping annotated openFixedDeposit method. When the getNewFixedDepositDetails method is invoked, the returned FixedDepositDetails instance is stored in the Model object with name newFixedDepositDetails. Now, the openFixedDeposit method’s fixedDepositDetails argument is annotated with @ModelAttribute(name="newFixedDepositDetails"); therefore, the newFixedDepositDetails object is obtained from the Model object and assigned to the fixedDepositDetails argument.

If the model attribute specified by the @ModelAttribute annotation is not found in the Model, Spring automatically creates a new instance of the method argument type, assigns it to the method argument and also puts it into the Model object. To allow Spring to create an instance of the method argument type, the Java class of the method argument type must provide a no-argument constructor.

**What is use of RequestToViewNameTranslator?**

RequestToViewNameTranslator determines the view to be rendered when a @RequestMapping annotated method doesn’t explicitly specify the view to be rendered. When a @RequestMapping method is also annotated with @ModelAttribute annotation, the value returned by the method is considered as a model attribute. In such a situation, the RequestToViewNameTranslator object is responsible for determining the view to be rendered based on the incoming web request. Similarly, if a @RequestMapping annotated method returns void, org.springframework.ui.Model or java.util.Map, the RequestToViewNameTranslator object determines the view to be rendered.

DefaultRequestToViewNameTranslator is an implementation of RequestToViewNameTranslator that is used by default by DispatcherServlet to determine the view to be rendered when no view is explicitly returned by a @RequestMapping method. DefaultRequestToViewNameTranslator uses the request URI to determine the name of the logical view to render. DefaultRequestToViewNameTranslator removes the leading and trailing slashes and the file extension from the URI to determine the view name. For instance, if the URL is [***http://localhost:8080/doSomething.htm***](http://localhost:8080/doSomething.htm), the view name becomes 'doSomething'.

**What is @SessionAttributes annotation?**

We know that all the @ModelAttribute annotated methods of a controller are always invoked before the @RequestMapping annotated method. This behavior may not be acceptable in situations in which @ModelAttribute methods obtain data from the database or from an external web service to populate the model attribute. In such scenarios, you can annotate your controller class with @SessionAttributes annotation that specifies the model attributes that are stored in HttpSession between requests.

If @SessionAttributes annotation is used, a @ModelAttribute annotated method is invoked only if the model attribute specified by the @ModelAttribute annotation is not found in the HttpSession. Also, @ModelAttribute annotation on a method argument will result in creation of a new instance of model attribute only if the model attribute is not found in the HttpSession.