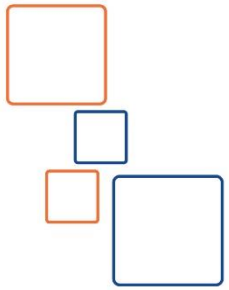
Spring Framework

THE RIGHT TECHNOLOGY STACK FOR THE JOB AT HAND



Java™ Education and Technology Services

Invest In Yourself, Develop Your Career

Course Outline

- Lesson 1: Spring MVC Introduction
    - (DispatcherServlet, Application and Web Application Context)
- Lesson 2: Spring MVC Request Lifecycle
- Lesson 3: Spring MVC Hello World Example
- Lesson 4: Handler Mappings
- Lesson 5: Controllers

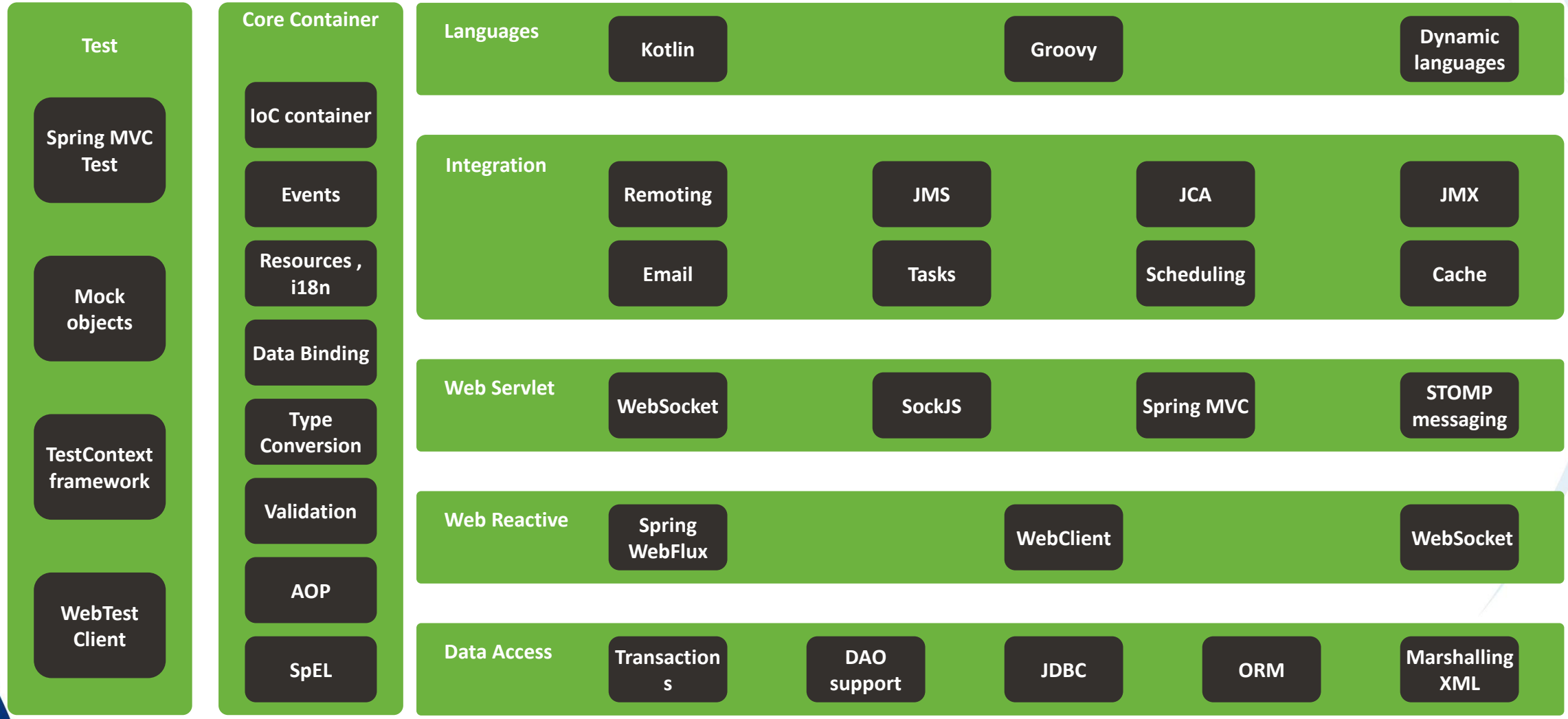
*** **References & Recommended Reading**

Lesson 1

Spring MVC Introduction

(DispatcherServlet, Application and Web Application Context)

Spring Framework Modules



Spring Web MVC

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

Spring Web MVC (Ex.)

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

Spring Application and Web Application Context

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

Spring Application and Web Application Context

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

Spring Application and Web Application Context

- The configuration would look something like this:

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

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

Spring Application and Web Application Context

- The purpose of the ContextLoaderListener is two function:

    1. To tie the lifecycle of the ApplicationContext to the lifecycle of the ServletContext.

    2. To automate the creation of the ApplicationContext, so you don't have to write explicit code to do create it - it's a convenience function.

Spring Application and Web Application Context

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

Spring Application and Web Application Context

- The configuration would look something like this:

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

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

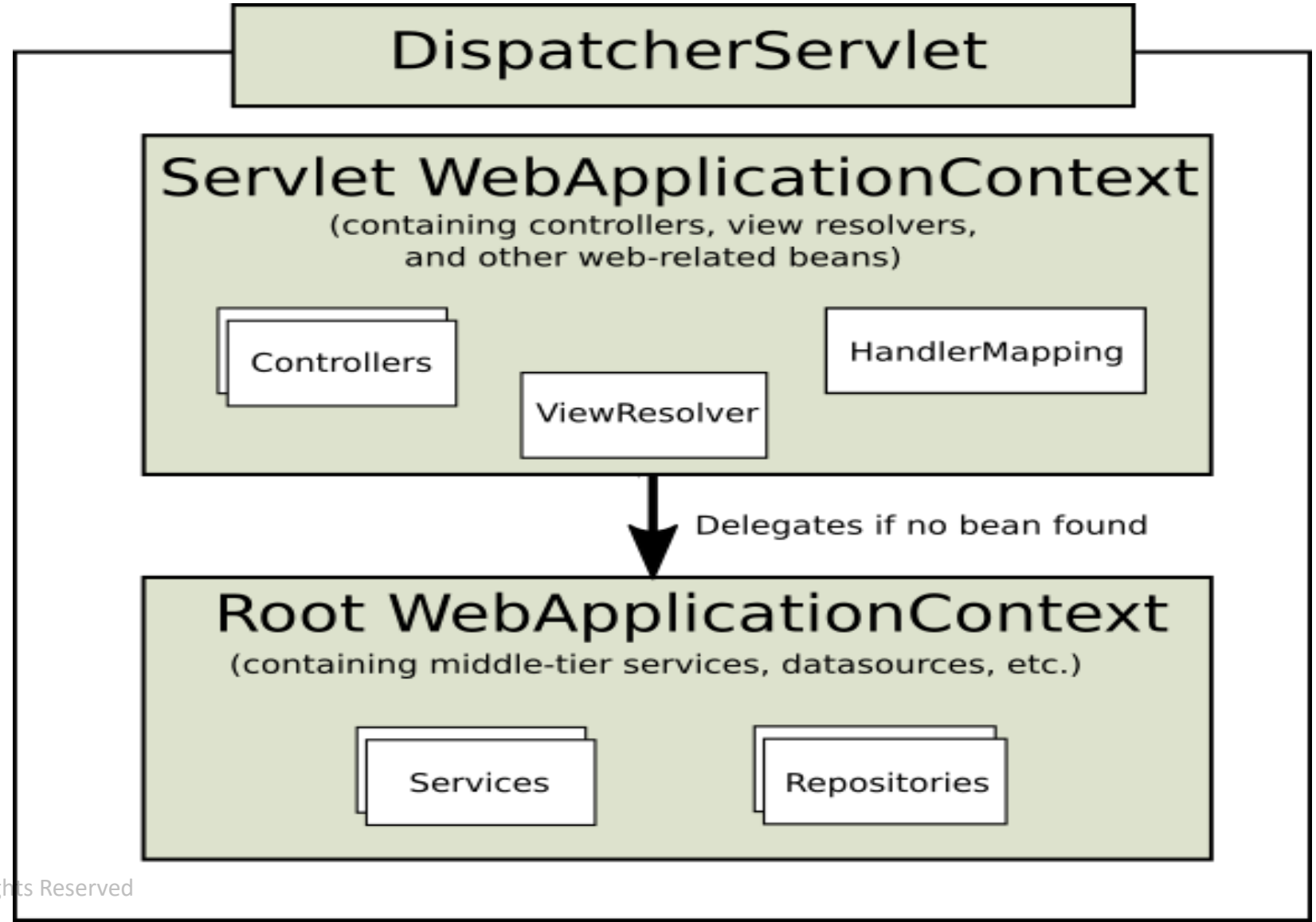
Spring Application and Web Application Context

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

Spring Application and Web Application Context

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

Spring Application and Web Application Context



Spring Application and Web Application Context

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

Using DispatcherServlet

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

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

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

Using DispatcherServlet (Ex.)

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

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

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

Using DispatcherServlet (Ex.)

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

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

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

Using DispatcherServlet (Ex.)

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

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

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

Using WebApplicationInitializer

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

Using WebApplicationInitializer (Ex.)

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

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

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

Using WebApplicationInitializer (Ex.)

- If your configuration is Java Based, you could user AnnotationConfigWebApplicationContext to create context as root configuration using @Configuration:

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {

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

Using WebApplicationInitializer (Ex.)

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

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

Using AbstractAnnotationConfigDispatcherServletInitializer

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

Using WebApplicationInitializer (Ex.)

```
public class MyWebApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

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

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

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

Using AbstractAnnotationConfigDispatcherServletInitializer

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

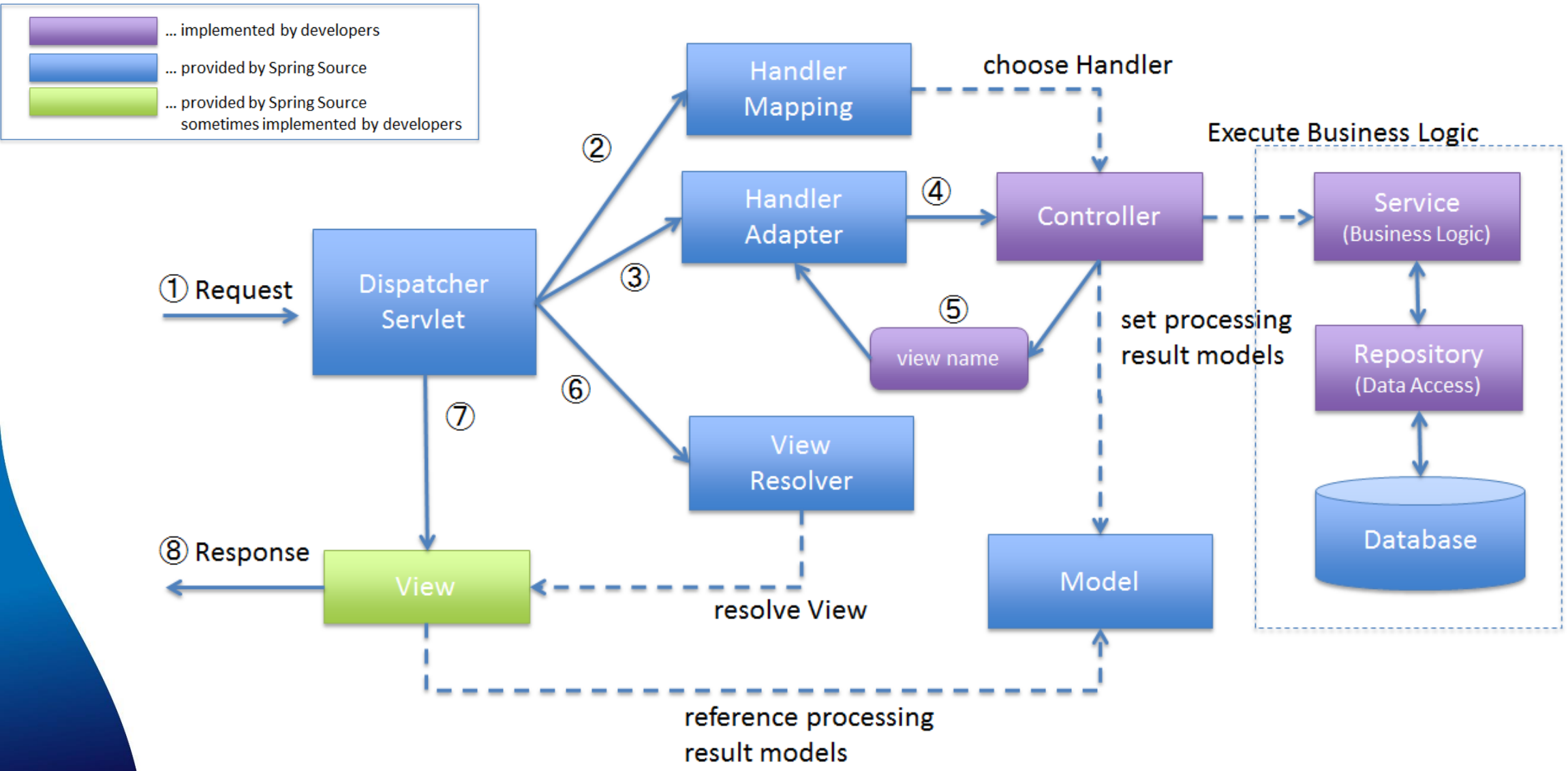
```
public class MyWebApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

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

Lesson 2

Spring MVC Request Lifecycle

Request Life Cycle



Request Life Cycle (Ex.)

- When the request leaves the browser, it carries information (e.g. the requested URL, the information submitted in a form ,etc..)

1. The first step in the request's life cycle is Spring's DispatcherServlet receives the request
   - It act as façade class for user request handling.
2. DispatcherServlet dispatches (delegate) the task of selecting an appropriate controller to HandlerMapping.
   - HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.

Request Life Cycle (Ex.)

3. DispatcherServlet dispatches (delegate) the task of executing of business logic of Controller to HandlerAdapter.

4. HandlerAdapter calls the business logic process of Controller.

   - Also responsible for data binding and conversion between the row request and the controller data binding.

5. Controller

   - Executes the business logic.
   - Sets the processing result in Model.
   - Returns the logical name of view to HandlerAdapter.

Request Life Cycle (Ex.)

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

Lesson 4

Handler Mappings

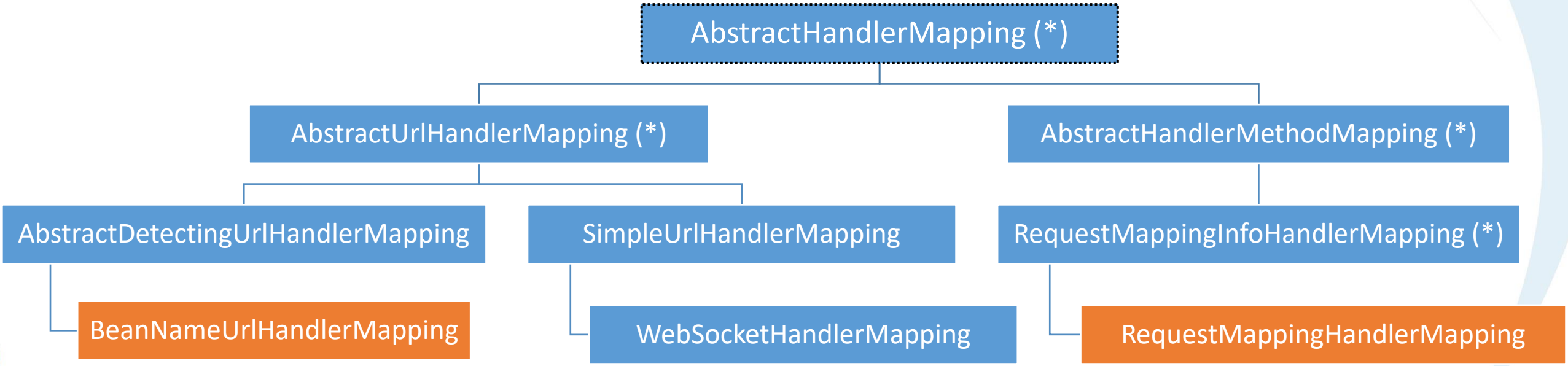
Handler Mappings

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

Handler Mappings (Ex.)

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

Handler Mappings Types



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

Handler Mappings Types (Ex.)

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

Handler Mappings Types (Ex.)

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

Handler Mappings Types (Ex.)

- ControllerClassNameHandlerMapping
- As the following example:

```
<bean id="urlMapping"
      class="org.springframework.web.servlet.mvc.ControllerClassNameHandlerMapping"/>

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

- The previous example define this controller HelloWorldController will response to the URL

  /helloworld.htm

Including Multiple Handler Mappings

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

Including Multiple Handler Mappings (Ex.)

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

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

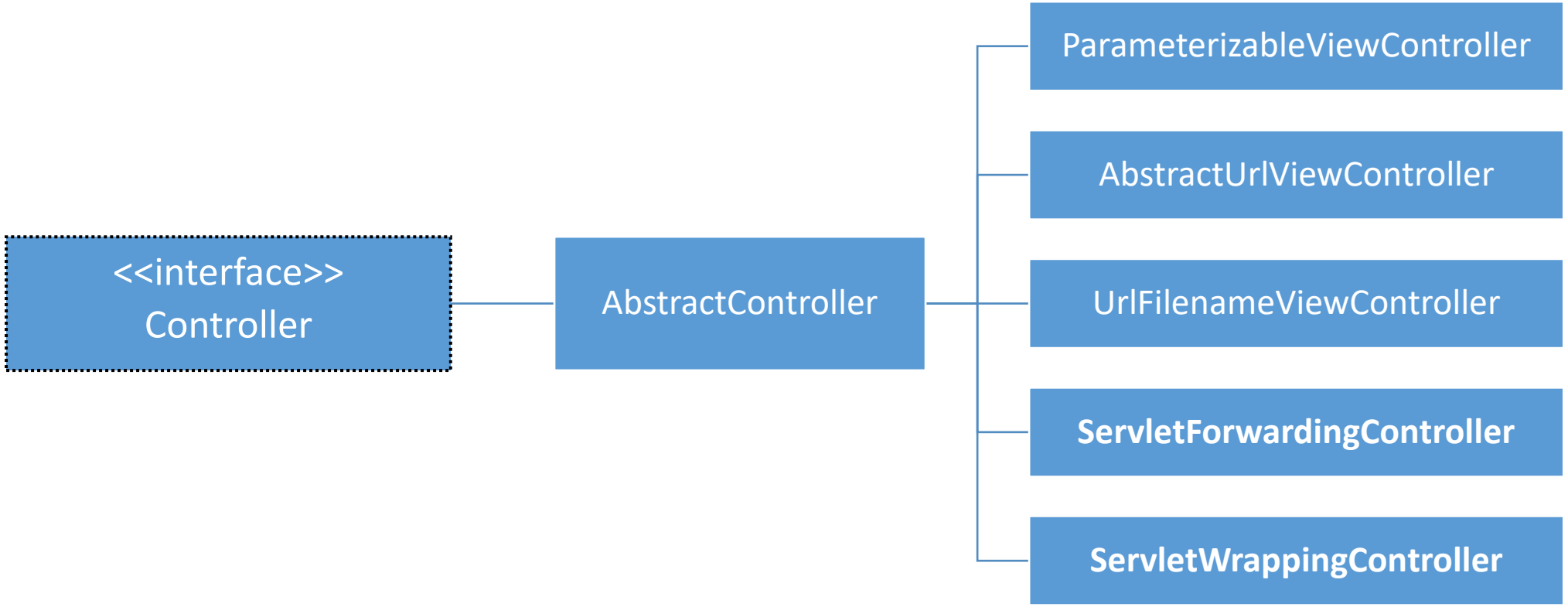
Lesson 5

Controllers

What is Controllers?

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

Controllers Hierarchy



Enabling URL Mapping by Annotation

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

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

Enabling URL Mapping by Annotation (Ex.)

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

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

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

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

Enabling URL Mapping by Annotation (Ex.)

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

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

Enabling URL Mapping by Annotation (Ex.)

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

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

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

Enabling URL Mapping by Annotation (Ex.)

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

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

Spring MVC Annotations

- @Controller
- @RequestMapping
- @RequestParam
- @PathVariable
- @ResponseBody
- @RequestBody

- @RestController
- @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping
- @ResponseStatus

@Controller

- This annotation is used to make a class as a web controller, which can handle client requests and send a response back to the client.

- This is a class-level annotation, which is put on top of your controller class.
- Similar to @Service and @Repository it is also a stereotype annotation.

```
@Controller
public class HelloController {
   // handler methods
}
```

- This is a simple controller class which contains handler methods to handle HTTP request for different URLs.

@RequestMapping

- It provides the mapping between the request path and the handler method.

```
@Controller
@ResponseBody
@RequestMapping("/api")
public class HelloController {

   @RequestMapping("/hi")
   public String hi() {
      return "Response for '/hi' with method GET";
   }

   @RequestMapping(method = RequestMethod.GET, path = "/hello")
   public String helloGet() {
      return "Response for '/hello' with method GET";
   }

   @RequestMapping(method = RequestMethod.POST, path = "/hello")
   public String helloPost() {
      return "Response for '/hello' with method POST";
   }

}
```

@RequestParam

- Used to bind HTTP parameters into method arguments of handler methods.
- In Spring MVC, "request parameters" map to query parameters, form data, and parts in multipart requests.

```
@Controller
@ResponseBody
public class HelloController {


   @RequestMapping("/hello")
   public String hello(@RequestParam String myParam) {
      return "Param Value: " + myParam;
   }


}
```

@PathVariable

- Enables the controller to handle a request for parameterized URLs like URLs that have variable input as part of their path.

```
@Controller
@ResponseBody
public class HelloController {


   @RequestMapping("/hello/{myPathVariable}")
   public String hello(@PathVariable String myPathVariable) {
      return "Path Variable Value: " + myPathVariable;
   }


}
```

@ResponseBody

- The @ResponseBody annotation is one of the most useful annotations for developing RESTful web service using Spring MVC.
- This annotation is used to transform a Java object returned from the controller to a resource representation requested by a REST client.
- ***It can completely bypass the view resolution part.***

```
@Controller
public class CoursesController {

    @RequestMapping(method = RequestMethod.POST, consumes = "application/json")
    public @ResponseBody Course saveCourse(@RequestBody Course course) {
        return coursesRepository.save(course);
    }

}
```

@RequestBody

- This annotation can convert inbound HTTP data into Java objects passed into the controller's handler method.
- Just as @ResponseBody tells the Spring MVC to use a message converter when sending a response to the client, ***the @RequestBody annotations tell the Spring to find a suitable message converter to convert a resource representation coming from a client into an object***.

```
@Controller
public class CoursesController {

   @RequestMapping(method = RequestMethod.POST, consumes = "application/json")
   public @ResponseBody Course saveCourse(@RequestBody Course course) {
      return coursesRepository.save(course);
   }

}
```

@RestController

- This is a **convenience annotation** for developing a RESTful web service with the Spring MVC framework.
- The ***@RestController*** is a combination of ***@Controller*** and ***@ResponseBody***, which was introduced in the Spring 3.4 version.

- When you annotate a controller class with @RestController it does two purposes,
- First, it says that the controller class is handling a request for REST APIs
- Second, you don't need to annotate each method with the @ResposneBody annotation to signal that the response will be converted into a Resource using various HttpMessageConverers.

@RestController

```
@Controller
@ResponseBody
public class HelloController {
   @RequestMapping("/hello")
   public String hello() {
      return "Hello, REST";
   }
}
```

```
@Controller
public class HelloController {
   @RequestMapping("/hello")
   public @ResponseBody String hello() {
      return "Hello, REST";
   }
}
```

```
@RestController
public class HelloController {

   @RequestMapping("/hello")
   public String hello() {
      return "Hello, REST";
   }

}
```

Java™ Education and Technology Services

@XxxMapping

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping

```
@RestController
public class BooksController {

   @GetMapping("/books/{id}")
   public Book getBookById(@RequestParam String id) {
      return bookService.findBook(id);
   }

}
```

```
@RestController
public class BooksController {

   @RequestMapping(method = RequestMethod.GET, path = "/books/{id}")
   public Book getBookById(@RequestParam String id) {
      return bookService.findBook(id);
   }

}
```

@ResponseStatus

- This annotation can be used to override the HTTP response code for a response.
- You can use this annotation for error handling while developing a web application or RESTful web service using Spring.

```
@ExceptionHandler(BookNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
public Error bookNotFound(BookNotFoundException ex) {
   long ISBN = ex.getISBN();
   return new Error(4, "Book [" + ISBN + "] not found");
}
```

- Here we are returning an error if the book for the given ISBN is not found in our library, and we have used @ResponseStatus to override the HTTP status code as 404 to indicate the client a missing resource.

Response Metadata

Java™ Education and Technology Services

ResponseEntity

- ResponseEntity represents the whole HTTP response: status code, headers, and body. As a result, we can use it to fully configure the HTTP response.

- If we want to use it, we have to return it from the endpoint; Spring takes care of the rest.

- ResponseEntity is a generic type.
  Consequently, we can use any type as the response body.

```
@GetMapping("/hello")
public ResponseEntity<String> hello() {
   return new ResponseEntity<>("Hello World!", HttpStatus.OK);
}
```

ResponseEntity

- Since we specify the response status programmatically,
  we can return with different status codes for different scenarios:

```
@GetMapping("/age")
public ResponseEntity<String> age(@RequestParam("yearOfBirth") int yearOfBirth) {

    if (isInFuture(yearOfBirth)) {
        return new ResponseEntity<>("YOB cannot be in the future", HttpStatus.BAD_REQUEST);
    }

    return new ResponseEntity<>("Your age is " + calculateAge(yearOfBirth), HttpStatus.OK);
}
```

ResponseEntity

- Additionally, we can set HTTP headers:

```
@GetMapping("/customHeader")
public ResponseEntity<String> customHeader() {
    HttpHeaders headers = new HttpHeaders();
    headers.add("Custom-Header", "foo");

    return new ResponseEntity<>("Custom header set", headers, HttpStatus.OK);
}
```

Java™ Education and Technology Services

ResponseEntity – Builder Interfaces

- Furthermore, ResponseEntity provides two nested builder interfaces:
  - HeadersBuilder and its sub-interface, BodyBuilder.
- We can access their capabilities through the static methods of ResponseEntity.

- The simplest case is a response with a body and HTTP 200 response code:

```
@GetMapping("/hello")
public ResponseEntity<String> hello() {
   return ResponseEntity
                          .ok("Hello World!");
}
```

```
// For the most popular HTTP status codes
// we get static methods:

BodyBuilder accepted();
BodyBuilder badRequest();
BodyBuilder created(java.net.URI location);
HeadersBuilder<?> noContent();
HeadersBuilder<?> notFound();
BodyBuilder ok();

BodyBuilder status(HttpStatus status);
BodyBuilder status(int status);
```

ResponseEntity – Builder Interfaces

ResponseEntity<T> BodyBuilder.body(T body) we can set the HTTP response body:

Since BodyBuilder.body() returns a ResponseEntity instead of BodyBuilder, it should be the last call.

```
@GetMapping("/age")
public ResponseEntity<String> age(@RequestParam("yearOfBirth") int yearOfBirth) {
   if (isInFuture(yearOfBirth)) {
      return ResponseEntity
                        .badRequest()
                        .body("Year of birth cannot be in the future");
   }
   return ResponseEntity
                        .status(HttpStatus.OK)
                        .body("Your age is " + calculateAge(yearOfBirth));
}
```

ResponseEntity – Builder Interfaces

We can also set custom headers:

Since BodyBuilder.body() returns a ResponseEntity instead of BodyBuilder, it should be the last call.

```
@GetMapping("/customHeader")
public ResponseEntity<String> customHeader() {
    return ResponseEntity.ok()
        .header("Custom-Header", "foo")
        .body("Custom header set");
}
```

Java™ Education and Technology Services

Exception Handling

Error Handling for REST with Spring

- Before Spring 3.2, the two main approaches to handling exceptions in a Spring MVC application were ***HandlerExceptionResolver*** or the ***@ExceptionHandler*** annotation. Both have some clear downsides.

- Since 3.2, we've had the ***@ControllerAdvice*** annotation to address the limitations of the previous two solutions and to promote a unified exception handling throughout a whole application.

- Now Spring 5 introduces the ***ResponseStatusException*** class — a fast way for basic error handling in our REST APIs.

- All of these do have one thing in common: They deal with the ***separation of concerns*** very well. The app can throw exceptions normally to indicate a failure of some kind, which will then be handled separately.

Error Handling for REST with Spring

- ***Solution 1:*** The Controller-Level @ExceptionHandler
- Solution 2: The HandlerExceptionResolver
- ***Solution 3:*** @ControllerAdvice
- Solution 4: ResponseStatusException (Spring 5 and Above)

The Controller-Level @ExceptionHandler

- This approach has a major drawback:
  The @ExceptionHandler annotated method is only active for that particular Controller, not globally for the entire application.
- Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.

```
public class FooController {

   // Handler methods ...

   @ExceptionHandler({ CustomException1.class, CustomException2.class })
   public void handleException() {
      //
   }

}
```

@ControllerAdvice

- The @ControllerAdvice annotation allows us to consolidate our multiple, scattered @ExceptionHandlers from before into a single, global error handling component.

- The actual mechanism is extremely simple but also very flexible:
  - It gives us full control over the body of the response as well as the status code.
  - It provides mapping of several exceptions to the same method, to be handled together.
  - It makes good use of the newer RESTful ResposeEntity response.

- One thing to keep in mind here is to match the exceptions declared with @ExceptionHandler to the exception used as the argument of the method.

@ControllerAdvice

```
public class ErrorDetails {
   private String errorCode;
   private String errorMessage;
   private String devErrorMessage;
   private Map<String, Object> additionalData = new HashMap<>();
   // setters & getters
}
```

```
@ControllerAdvice
public class MyGlobalExceptionHandler {

   @ExceptionHandler(MyCustomRuntimeException.class)
   public ResponseEntity<?> handleMyCustomException(MyCustomRuntimeException ex) {
      ErrorDetails errorDetails = new ErrorDetails();
      return new ResponseEntity<>(errorDetails, HttpStatus.INTERNAL_SERVER_ERROR);
   }

}
```

References & Recommended Reading

References & Recommended Reading

- Spring in Action 5th Edition
- Cloud Native Java
- Learning Spring Boot 2.0
- Spring 5 Recipes: A Problem-Solution Approach