Spring Theory

Some basic theory on spring:

# Spring WebMVC

* Spring Web MVC is the original web framework built on the Servlet API.
* The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.
* The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.
* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.
* Spring MVC, as many other web frameworks, is designed around the **front controller pattern** where a central Servlet, the DispatcherServlet, provides a shared algorithm for request processing, while actual work is performed by configurable delegate components.
* The DispatcherServlet, as any Servlet, needs to be declared and mapped according to the Servlet specification by using Java configuration or in web.xml.
* The DispatcherServlet delegates to special beans to process requests and render the appropriate responses. By “special beans” we mean Spring-managed Object instances that implement framework contracts. Those usually come with built-in contracts, but you can customize their properties and extend or replace them.
* Following are the special beans detected by the DispatcherServlet:

1. **HandlerMapping --** Map a request to a handler along with a list of [interceptors](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-handlermapping-interceptor) for pre- and post-processing.

The two main HandlerMapping implementations are **RequestMappingHandlerMapping** (which supports **@RequestMapping** annotated methods) and **SimpleUrlHandlerMapping** (which maintains explicit registrations of URI path patterns to handlers).

1. **HandlerAdapter –** Help the DispatcherServlet to invoke a handler mapped to a request, regardless of how the handler is actually invoked.
2. **HandlerExceptionResolver –** Strategy to resolve exceptions, possibly mapping them to handlers, to HTML error views, or other targets.
3. **ViewResolver –** Resolve logical String-based view names returned from a handler to an actual View with which to render to the response.
4. **LocaleResolver –** Resolve the Locale a client is using and possibly their time zone, in order to be able to offer internationalized views.
5. **ThemeResolver –** Resolve themes your web application can use — for example, to offer personalized layouts.
6. **MultipartResolver –** Abstraction for parsing a multi-part request (for example, browser form file upload) with the help of some multipart parsing library.
7. **FlashMapResolver --**Store and retrieve the “input” and the “output” FlashMap that can be used to pass attributes from one request to another, usually across a redirect.

* The **DispatcherServlet** processes requests as follows:
* All **HandlerMapping** implementations support handler interceptors that are useful when you want to apply specific functionality to certain requests — for example, checking for a principal.
* Spring MVC defines **the ViewResolver** and **View** interfaces that let you render models in a browser without tying you to a specific view technology.
* **ViewResolver** provides a mapping between view names and actual views.
* **View** addresses the preparation of data before handing over to a specific view technology.
* Browsers can submit form data only through HTTP GET or HTTP POST but non-browser clients can also use HTTP PUT, PATCH, and DELETE.
* The Servlet API requires ServletRequest.getParameter\*() methods to support form field access only for HTTP POST.
* Spring MVC provides an annotation-based programming model where **@Controller** and **@RestController** components use annotations to express request mappings, request input, exception handling, and more.
* Annotated controllers have flexible method signatures and do not have to extend base classes nor implement specific interfaces.
* **The @Controller** stereotype allows for auto-detection, aligned with Spring general support for detecting @Component classes in the classpath and auto-registering bean definitions for them. It also acts as a stereotype for the annotated class, indicating its role as a web component.
* **@RestController** is a [composed annotation](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/core.html#beans-meta-annotations) that is itself meta-annotated with **@Controller** and **@ResponseBody** to indicate a controller whose every method inherits the type-level **@ResponseBody** annotation and, therefore, writes directly to the response body versus view resolution and rendering with an HTML template.
* You can use the **@RequestMapping** annotation to map requests to controllers methods.
* It has various attributes to match by URL, HTTP method, request parameters, headers, and media types.
* You can use it at the class level to express shared mappings or at the method level to narrow down to a specific endpoint mapping.
* There are also HTTP method specific shortcut variants of **@RequestMapping:**
* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping
* The shortcuts are [Custom Annotations](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-composed) that are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using @RequestMapping, which, by default, matches to all HTTP methods. At the same, a @RequestMapping is still needed at the class level to express shared mappings.
* You can map requests by using glob patterns and wildcards:

|  |  |  |
| --- | --- | --- |
| ? | Matches one character | "/pages/t?st.html"  matches "/pages/test.html" and "/pages/t3st.html" |
| \* | Matches zero or more characters within a path segment | "/resources/\*.png" matches "/resources/file.png"  "/projects/\*/versions" matches "/projects/spring/versions" but does not match "/projects/spring/boot/versions" |
| \*\* | Matches zero or more path segments until the end of the path | "/resources/\*\*" matches "/resources/file.png" and "/resources/images/file.png" |
| {name} | Matches a path segment and captures it as a variable named "name" | "/projects/{project}/versions" matches "/projects/spring/versions" and captures project=spring |
| {name:[a-z]+} | Matches the regexp "[a-z]+" as a path variable named "name" | "/projects/{project:[a-z]+}/versions" matches "/projects/spring/versions" but not "/projects/spring1/versions" |

* Captured URI variables can be accessed with @PathVariable, as the following example shows:

@GetMapping("/owners/{ownerId}/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

* You can declare URI variables at the class and method levels, as the following example shows:

@Controller

@RequestMapping("/owners/{ownerId}")

public class OwnerController {

@GetMapping("/pets/{petId}")

public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {

// ...

}

}

* You can narrow the request mapping based on the Content-Type of the request, as the following example shows:

@PostMapping(path = "/pets", consumes = "application/json")

public void addPet(@RequestBody Pet pet) {

// ...

}

* Using a consumes attribute to narrow the mapping by the content type.
* The consumes attribute also supports negation expressions — for example, !text/plain means any content type other than text/plain.
* MediaType provides constants for commonly used media types, such as APPLICATION\_JSON\_VALUE and APPLICATION\_XML\_VALUE.
* You can narrow the request mapping based on the Accept request header and the list of content types that a controller method produces, as the following example shows:

@GetMapping(path = "/pets/{petId}", produces = "application/json")

@ResponseBody

public Pet getPet(@PathVariable String petId) {

// ...

}

* Using a produces attribute to narrow the mapping by the content type.

the media type can specify a character set. Negated expressions are supported — for example, !text/plain means any content type other than "text/plain"

* You can narrow request mappings based on request parameter conditions. You can test for the presence of a request parameter (myParam), for the absence of one (!myParam), or for a specific value (myParam=myValue). The following example shows how to test for a specific value:

@GetMapping(path = "/pets/{petId}", params = "myParam=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

Testing whether myParam equals myValue.

* You can also use the same with request header conditions, as the following example shows:

@GetMapping(path = "/pets", headers = "myHeader=myValue")

public void findPet(@PathVariable String petId) {

// ...

}

|  |  |
| --- | --- |
|  | Testing whether myHeader equals myValue. |

* Spring MVC supports the use of [composed annotations](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/core.html#beans-meta-annotations) for request mapping. Those are annotations that are themselves meta-annotated with @RequestMapping and composed to redeclare a subset (or all) of the @RequestMapping attributes with a narrower, more specific purpose.
* @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, and @PatchMapping are examples of composed annotations. They are provided because, arguably, most controller methods should be mapped to a specific HTTP method versus using @RequestMapping, which, by default, matches to all HTTP methods.
* @RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.
* The next table describes the supported controller method arguments. Reactive types are not supported for any arguments.

|  |  |
| --- | --- |
| **Controller method argument** | **Description** |
| WebRequest, NativeWebRequest | Generic access to request parameters and request and session attributes, without direct use of the Servlet API. |
| javax.servlet.ServletRequest, javax.servlet.ServletResponse | Choose any specific request or response type — for example, ServletRequest, HttpServletRequest, or Spring’s MultipartRequest, MultipartHttpServletRequest. |
| javax.servlet.http.HttpSession | Enforces the presence of a session. As a consequence, such an argument is never null. Note that session access is not thread-safe. Consider setting the RequestMappingHandlerAdapter instance’s synchronizeOnSession flag to true if multiple requests are allowed to concurrently access a session. |
| javax.servlet.http.PushBuilder | Servlet 4.0 push builder API for programmatic HTTP/2 resource pushes. Note that, per the Servlet specification, the injected PushBuilder instance can be null if the client does not support that HTTP/2 feature. |
| java.security.Principal | Currently authenticated user — possibly a specific Principal implementation class if known. |
| HttpMethod | The HTTP method of the request. |
| java.util.Locale | The current request locale, determined by the most specific LocaleResolver available (in effect, the configured LocaleResolver or LocaleContextResolver). |
| java.util.TimeZone + java.time.ZoneId | The time zone associated with the current request, as determined by a LocaleContextResolver. |
| java.io.InputStream, java.io.Reader | For access to the raw request body as exposed by the Servlet API. |
| java.io.OutputStream, java.io.Writer | For access to the raw response body as exposed by the Servlet API. |
| @PathVariable | For access to URI template variables. See [URI patterns](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-uri-templates). |
| @MatrixVariable | For access to name-value pairs in URI path segments. See [Matrix Variables](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-matrix-variables). |
| @RequestParam | For access to the Servlet request parameters, including multipart files. Parameter values are converted to the declared method argument type. See [@RequestParam](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestparam) as well as [Multipart](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-multipart-forms).  Note that use of @RequestParam is optional for simple parameter values. See “Any other argument”, at the end of this table. |
| @RequestHeader | For access to request headers. Header values are converted to the declared method argument type. See [@RequestHeader](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestheader). |
| @CookieValue | For access to cookies. Cookies values are converted to the declared method argument type. See [@CookieValue](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-cookievalue). |
| @RequestBody | For access to the HTTP request body. Body content is converted to the declared method argument type by using HttpMessageConverter implementations. See [@RequestBody](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestbody). |
| HttpEntity<B> | For access to request headers and body. The body is converted with an HttpMessageConverter. See [HttpEntity](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-httpentity). |
| @RequestPart | For access to a part in a multipart/form-data request, converting the part’s body with an HttpMessageConverter. See [Multipart](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-multipart-forms). |
| java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap | For access to the model that is used in HTML controllers and exposed to templates as part of view rendering. |
| RedirectAttributes | Specify attributes to use in case of a redirect (that is, to be appended to the query string) and flash attributes to be stored temporarily until the request after redirect. See [Redirect Attributes](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-redirecting-passing-data) and [Flash Attributes](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-flash-attributes). |
| @ModelAttribute | For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied. See [@ModelAttribute](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-method-args) as well as [Model](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-modelattrib-methods) and [DataBinder](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-initbinder).  Note that use of @ModelAttribute is optional (for example, to set its attributes). See “Any other argument” at the end of this table. |
| Errors, BindingResult | For access to errors from validation and data binding for a command object (that is, a @ModelAttribute argument) or errors from the validation of a @RequestBody or @RequestPart arguments. You must declare an Errors, or BindingResult argument immediately after the validated method argument. |
| SessionStatus + class-level @SessionAttributes | For marking form processing complete, which triggers cleanup of session attributes declared through a class-level @SessionAttributes annotation. See [@SessionAttributes](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattributes) for more details. |
| UriComponentsBuilder | For preparing a URL relative to the current request’s host, port, scheme, context path, and the literal part of the servlet mapping. See [URI Links](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-uri-building). |
| @SessionAttribute | For access to any session attribute, in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration. See [@SessionAttribute](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-sessionattribute) for more details. |
| @RequestAttribute | For access to request attributes. See [@RequestAttribute](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestattrib) for more details. |
| Any other argument | If a method argument is not matched to any of the earlier values in this table and it is a simple type (as determined by [BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.2.7.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-), it is a resolved as a @RequestParam. Otherwise, it is resolved as a @ModelAttribute. |

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.

* The next table describes the supported controller method return values. Reactive types are supported for all return values.

|  |  |
| --- | --- |
| **Controller method return value** | **Description** |
| @ResponseBody | The return value is converted through HttpMessageConverter implementations and written to the response. See [@ResponseBody](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-responsebody). |
| HttpEntity<B>, ResponseEntity<B> | The return value that specifies the full response (including HTTP headers and body) is to be converted through HttpMessageConverter implementations and written to the response. See [ResponseEntity](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-responseentity). |
| HttpHeaders | For returning a response with headers and no body. |
| String | A view name to be resolved with ViewResolver implementations and used together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method can also programmatically enrich the model by declaring a Model argument (see [Explicit Registrations](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-registration)). |
| View | A View instance to use for rendering together with the implicit model — determined through command objects and @ModelAttribute methods. The handler method can also programmatically enrich the model by declaring a Model argument (see [Explicit Registrations](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-requestmapping-registration)). |
| java.util.Map, org.springframework.ui.Model | Attributes to be added to the implicit model, with the view name implicitly determined through a RequestToViewNameTranslator. |
| @ModelAttribute | An attribute to be added to the model, with the view name implicitly determined through a RequestToViewNameTranslator.  Note that @ModelAttribute is optional. See "Any other return value" at the end of this table. |
| ModelAndView object | The view and model attributes to use and, optionally, a response status. |
| void | A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, an OutputStream argument, or an @ResponseStatus annotation. The same is also true if the controller has made a positive ETag or lastModified timestamp check (see [Controllers](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-caching-etag-lastmodified) for details).  If none of the above is true, a void return type can also indicate “no response body” for REST controllers or a default view name selection for HTML controllers. |
| DeferredResult<V> | Produce any of the preceding return values asynchronously from any thread — for example, as a result of some event or callback. See [Asynchronous Requests](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [DeferredResult](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async-deferredresult). |
| Callable<V> | Produce any of the above return values asynchronously in a Spring MVC-managed thread. See [Asynchronous Requests](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [Callable](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async-callable). |
| ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V> | Alternative to DeferredResult, as a convenience (for example, when an underlying service returns one of those). |
| ResponseBodyEmitter, SseEmitter | Emit a stream of objects asynchronously to be written to the response with HttpMessageConverter implementations. Also supported as the body of a ResponseEntity. See [Asynchronous Requests](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [HTTP Streaming](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async-http-streaming). |
| StreamingResponseBody | Write to the response OutputStream asynchronously. Also supported as the body of a ResponseEntity. See [Asynchronous Requests](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [HTTP Streaming](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async-http-streaming). |
| Reactive types — Reactor, RxJava, or others through ReactiveAdapterRegistry | Alternative to DeferredResult with multi-value streams (for example, Flux, Observable) collected to a List.  For streaming scenarios (for example, text/event-stream, application/json+stream), SseEmitter and ResponseBodyEmitter are used instead, where ServletOutputStream blocking I/O is performed on a Spring MVC-managed thread and back pressure is applied against the completion of each write.  See [Asynchronous Requests](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async) and [Reactive Types](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc-ann-async-reactive-types). |
| Any other return value | Any return value that does not match any of the earlier values in this table and that is a String or void is treated as a view name (default view name selection through RequestToViewNameTranslator applies), provided it is not a simple type, as determined by [BeanUtils#isSimpleProperty](https://docs.spring.io/spring-framework/docs/5.2.7.RELEASE/javadoc-api/org/springframework/beans/BeanUtils.html#isSimpleProperty-java.lang.Class-). Values that are simple types remain unresolved. |

* You can use the @RequestParam annotation to bind Servlet request parameters (that is, query parameters or form data) to a method argument in a controller.

@Controller

@RequestMapping("/pets")

public class EditPetForm {

// ...

@GetMapping

public String setupForm(@RequestParam("petId") int petId, Model model) {

Pet pet = this.clinic.loadPet(petId);

model.addAttribute("pet", pet);

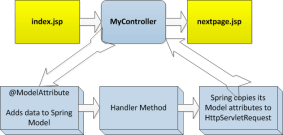
return "petForm";

}

// ...

}

* You can use the @RequestHeader annotation to bind a request header to a method argument in a controller.
* You can use the @CookieValue annotation to bind the value of an HTTP cookie to a method argument in a controller.
* You can use the @ModelAttribute annotation on a method argument to access an attribute from the model or have it be instantiated if not present.
* Model can supply attributes used for rendering views. To provide a view with usable data, we simply add this data to its Model object.
* In Spring MVC, the model works a container that contains the data of the application. Here, a data can be in any form such as objects, strings, information from the database, etc.
* It is required to place the **Model** interface in the controller part of the application. The object of **HttpServletRequest** reads the information provided by the user and pass it to the **Model** interface. Now, a view page easily accesses the data from the model part.
* There are several ways to add data or objects to Spring’s model.  Data or objects are typically added to Spring’s model via an @ModelAttribute annotated method in the controller.
* Spring model data stored?”  Is it stored in the standard Java request scope
* Spring does not add the model data to the request as an attribute until after the execution of the handler method and before presentation of the next view



* Spring model data created prior to (or during) the handler method execution has been copied to the HttpServletRequest before the next view is rendered.
* @SessionAttributes is used to store model attributes in the HTTP Servlet session between requests. It is a type-level annotation that declares the session attributes used by a specific controller.
* In actually, what @SessionAttributes allows you to do is tell Spring which of your model attributes will also be copied to HttpSession before rendering the view.

# Spring Boot

* The **SpringApplication** class provides a convenient way to bootstrap a Spring application that is started from a main() method.
* **@SpringBootApplication** is a convenience annotation that adds all of the following:
* **@Configuration**: Tags the class as a source of bean definitions for the application context.
* **@EnableAutoConfiguration**: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
  + For example, if spring-webmvc is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a DispatcherServlet.
* **@ComponentScan:** Tells Spring to look for other components, configurations, and services in the com/example package, letting it find the controllers.
* The main() method uses Spring Boot’s SpringApplication.run() method to launch an application.
* There is also **a CommandLineRunner** method marked as a @Bean, and this runs on start up. It retrieves all the beans that were created by your application or that were automatically added by Spring Boot. It sorts them and prints them out.

# SpringBoot WebMVC (spring-boot-starter-web)

* Spring Boot is well suited for web application development.
* You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty.
* Spring MVC lets you create **special @Controller or @RestController beans** to handle incoming HTTP requests.

Methods in your controller are mapped to HTTP by using **@RequestMapping** annotations.

* If a class is flagged with **@RestController**, meaning it is ready for use by Spring MVC to handle web requests.
* **@RestController** combines **@Controller** and **@ResponseBody**, two annotations that results in web requests returning data (pure text) rather than a view.
* If you want to keep those Spring Boot MVC customizations and make more [MVC customizations](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#mvc) (interceptors, formatters, view controllers, and other features), you can add your own **@Configuration** class of type **WebMvcConfigurer** but **without** **@EnableWebMvc**.
* If you want to take complete control of Spring MVC, you can add your own **@Configuration** annotated with **@EnableWebMvc**, or alternatively add your own **@Configuration-annotated DelegatingWebMvcConfiguration** as described in the Javadoc of @EnableWebMvc.
* By default, Spring Boot serves static content from a directory called /static (or /public or /resources or /META-INF/resources) in the classpath or from the root of the ServletContext
* Spring Boot supports both static and templated welcome pages. It first looks for an index.html file in the configured static content locations. If one is not found, it then looks for an index template. If either is found, it is automatically used as the welcome page of the application.
* Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, @GetMapping annotations on Controller methods).

# REST Clients

* Client side access to REST endpoints.
* The Spring Framework provides two choices for making calls to REST endpoints:

1. RestTemplate
2. WebClient

* **RestTemplate** is a synchronous client to perform HTTP requests.

It is the original Spring REST client and exposes a simple, template-method API over underlying HTTP client libraries.

* **WebClient** is a non-blocking, reactive client to perform HTTP requests.

It provides efficient support for synchronous and asynchronous, as well as streaming scenarios.

## RestTemplate –

If you need to call remote REST services from your application, you can use the Spring Framework’s [RestTemplate](https://docs.spring.io/spring/docs/5.2.7.RELEASE/javadoc-api/org/springframework/web/client/RestTemplate.html) class.

* The RestTemplate provides a higher level API over HTTP client libraries.
* It makes it easy to invoke REST endpoints in a single line. It exposes the following groups of overloaded methods:

|  |  |
| --- | --- |
| **Method group** | **Description** |
| getForObject | Retrieves a representation via GET. |
| getForEntity | Retrieves a ResponseEntity (that is, status, headers, and body) by using GET. |

* The default constructor uses java.net.HttpURLConnection to perform requests.
* You can switch to a different HTTP library with an implementation of ClientHttpRequestFactory. There is built-in support for the following:

1. Apache HttpComponents
2. Netty
3. OkHttp

* For example, to switch to Apache HttpComponents, you can use the following:

RestTemplate template = new RestTemplate(new HttpComponentsClientHttpRequestFactory());

* Each ClientHttpRequestFactory exposes configuration options specific to the underlying HTTP client library — for example, for credentials, connection pooling, and other details.
* Many of the RestTemplate methods accept a URI template and URI template variables, either as a String variable argument, or as Map<String,String>.

String result = restTemplate.getForObject(

"https://example.com/hotels/{hotel}/bookings/{booking}", String.class, "42", "21");

---

Map<String, String> vars = Collections.singletonMap("hotel", "42");

String result = restTemplate.getForObject(

"https://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);

* You can use the exchange() methods to specify request headers.
* You can obtain response headers through many RestTemplate method variants that return ResponseEntity.
* Objects passed into and returned from RestTemplate methods are converted to and from raw content with the help of an HttpMessageConverter.
* On a POST, an input object is serialized to the request body, as the following example shows:

URI location = template.postForLocation("https://example.com/people", person);

* On a GET, the body of the response is deserialized to an output Object, as the following example shows:

Person person = restTemplate.getForObject("https://example.com/people/{id}", Person.class, 42);

## WebClient –

* Spring WebFlux includes a reactive, non-blocking WebClient for HTTP requests
* The client has a functional, fluent API with reactive types for declarative composition.
* WebFlux client and server rely on the same non-blocking [codecs](https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web-reactive.html#webflux-codecs) to encode and decode request and response content.
* Internally WebClient delegates to an HTTP client library. By default, it uses [Reactor Netty](https://github.com/reactor/reactor-netty), there is built-in support for the Jetty [reactive HttpClient](https://github.com/jetty-project/jetty-reactive-httpclient), and others can be plugged in through a ClientHttpConnector.
* The simplest way to create a WebClient is through one of the static factory methods:
  + WebClient.create()
  + WebClient.create(String baseUrl)

The above methods use the Reactor Netty HttpClient with default settings and expect io.projectreactor.netty:reactor-netty to be on the classpath.

You can also use WebClient.builder() with further options.

* Once built, a WebClient instance is immutable. However, you can clone it and build a modified copy without affecting the original instance.
* The retrieve() method is the easiest way to get a response body and decode it. The following example shows how to do so:

WebClient client = WebClient.create("https://example.org");

Mono<Person> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.retrieve()

.bodyToMono(Person.class);

* The exchange() method provides more control than the retrieve method. The following example is equivalent to retrieve() but also provides access to the ClientResponse:

Mono<Person> result = client.get()

.uri("/persons/{id}", id).accept(MediaType.APPLICATION\_JSON)

.exchange()

.flatMap(response -> response.bodyToMono(Person.class));

* The request body can be encoded from any asynchronous type handled by ReactiveAdapterRegistry, like Mono or Kotlin Coroutines Deferred as the following example shows:
* To send form data, you can provide a MultiValueMap<String, String> as the body. Note that the content is automatically set to application/x-www-form-urlencoded by the FormHttpMessageWriter.

## SpringBoot RestTemplate

* Since RestTemplate instances often need to be customized before being used, Spring Boot does not provide any single auto-configured RestTemplate bean.
* It does, however, auto-configure a RestTemplateBuilder, which can be used to create RestTemplate instances when needed.
* The auto-configured RestTemplateBuilder ensures that sensible HttpMessageConverters are applied to RestTemplate instances.

@Service

public class MyService {

private final RestTemplate restTemplate;

public MyService(RestTemplateBuilder restTemplateBuilder) {

this.restTemplate = restTemplateBuilder.build();

}

public Details someRestCall(String name) {

return this.restTemplate.getForObject("/{name}/details", Details.class, name);

}

}

* RestTemplateBuilder includes a number of useful methods that can be used to quickly configure a RestTemplate. For example, to add BASIC auth support, you can use builder.basicAuthentication("user", "password").build().
* There are three main approaches to RestTemplate customization, depending on how broadly you want the customizations to apply.
* To make the scope of any customizations as narrow as possible, inject the auto-configured RestTemplateBuilder and then call its methods as required. Each method call returns a new RestTemplateBuilder instance, so the customizations only affect this use of the builder.
* To make an application-wide, additive customization, use a RestTemplateCustomizer bean. All such beans are automatically registered with the auto-configured RestTemplateBuilder and are applied to any templates that are built with it.
* Finally, the most extreme (and rarely used) option is to create your own RestTemplateBuilder bean. Doing so switches off the auto-configuration of a RestTemplateBuilder and prevents any RestTemplateCustomizer beans from being used.

## SpringBoot WebClient

# Common Facts: