***Here is an example of the @RequestMapping annotation applied to both class and methods.***

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping("/")

String get(){

//mapped to hostname:port/home/

return "Hello from get";

}

@RequestMapping("/index")

String index(){

//mapped to hostname:port/home/index/

return "Hello from index";

}

}

}

@RequestMapping with Multiple URIs

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value={"", "/page", "page\*","view/\*,\*\*/msg"})

String indexMultipleMapping(){

return "Hello from index multiple mapping.";

}

}

Example code:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/id")

String getIdByValue(@RequestParam("id") String personId){

System.out.println("ID is "+personId);

return "Get ID from query string of URL with value element";

}

@RequestMapping(value = "/personId")

String getId(@RequestParam String personId){

System.out.println("ID is "+personId);

return "Get ID from query string of URL without value element";

}

}

In Line 6 of this code, the request param id will be mapped to the personId parameter personId of the getIdByValue() handler method.

An example URL is this:

**localhost:8090/home/id?id=5**

The value element of @RequestParam can be omitted if the request param and handler method parameter names are the same as shown in Line 11.

An example URL is this:

**localhost:8090/home/personId?personId=5**

References:

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-requestmapping

https://springframework.guru/spring-requestmapping-annotation/

<https://springframework.guru/spring-requestmapping-annotation/>

**Discuss various HTTP methods**

Example code:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(method = RequestMethod.GET)

String get(){

return "Hello from get";

}

@RequestMapping(method = RequestMethod.DELETE)

String delete(){

return "Hello from delete";

}

@RequestMapping(method = RequestMethod.POST)

String post(){

return "Hello from post";

}

@RequestMapping(method = RequestMethod.PUT)

String put(){

return "Hello from put";

}

@RequestMapping(method = RequestMethod.PATCH)

String patch(){

return "Hello from patch";

}

}

In the code snippet above, the method element of the @RequestMapping annotations indicates the HTTP method type of the HTTP request.

All the handler methods will handle requests coming to the same URL ( /home), but will depend on the HTTP method being used.

For example, a POST request to /home will be handled by the post() method. While a DELETE request to /home will be handled by the delete() method.

You can see how Spring MVC will map the other methods using this same logic.

Example code:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/prod", produces = {"application/JSON"})

@ResponseBody

String getProduces(){

return "Produces attribute";

}

@RequestMapping(value = "/cons", consumes = {"application/JSON", "application/XML"})

String getConsumes(){

return "Consumes attribute";

}

}

In this code, the getProduces() handler method produces a JSON response. The getConsumes() handler method consumes JSON as well as XML present in requests.

References:

https://springframework.guru/spring-requestmapping-annotation/

https://springframework.guru/spring-requestmapping-annotation

Discuss Headers, Request Parameters, and Dynamic URIs

Example code:

@RequestMapping with Headers

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/head", headers = {"content-type=text/plain"})

String post(){

return "Mapping applied along with headers";

}

}

In the above code snippet, the headers attribute of the @RequestMapping annotation narrows down the mapping to the post() method. With this, the post() method will handle requests to /home/head whose content-typeheader specifies plain text as the value.

You can also indicate multiple header values like this:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/head", headers = {"content-type=text/plain", "content-type=text/html"}) String post(){

return "Mapping applied along with headers";

}

}

Here it implies that both text/plain as well as text/html are accepted by the post() handler method.

@RequestMapping with Request Parameters

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/fetch", params = {"personId=10"})

String getParams(@RequestParam("personId") String id){

return "Fetched parameter using params attribute = "+id;

}

@RequestMapping(value = "/fetch", params = {"personId=20"})

String getParamsDifferent(@RequestParam("personId") String id){

return "Fetched parameter using params attribute = "+id;

}

}

In this code snippet, both the getParams() and getParamsDifferent() methods will handle requests coming to the same URL ( /home/fetch) but will execute depending on the params element.

For example, when the URL is /home/fetch?id=10 the getParams() handler method will be executed with the id value 10.. For the URL, localhost:8080/home/fetch?personId=20, the getParamsDifferent() handler method gets executed with the id value 20.

Example code:

@RestController

@RequestMapping("/home")

public class IndexController {

@RequestMapping(value = "/fetch/{id}", method = RequestMethod.GET)

String getDynamicUriValue(@PathVariable String id){

System.out.println("ID is "+id);

return "Dynamic URI parameter fetched";

}

@RequestMapping(value = "/fetch/{id:[a-z]+}/{name}", method = RequestMethod.GET)

String getDynamicUriValueRegex(@PathVariable("name") String name){

System.out.println("Name is "+name);

return "Dynamic URI parameter fetched using regex";

}

}

In this code, the method getDynamicUriValue() will execute for a request to localhost:8080/home/fetch/10. Also, the id parameter of the getDynamicUriValue() handler method will be populated with the value 10 dynamically.

The method getDynamicUriValueRegex() will execute for a request to localhost:8080/home/fetch/category/shirt. However, an exception will be thrown for a request to /home/fetch/10/shirt as it does not match the regular expression.

@PathVariable works differently from @RequestParam. You use @RequestParam to obtain the values of the query parameters from the URI. On the other hand, you use @PathVariable to obtain the parameter values from the URI template.

References:

https://springframework.guru/spring-requestmapping-annotation/

https://springframework.guru/spring-requestmapping-annotation/

Discuss RequestMapping Shortcuts

Example code:

@RestController

@RequestMapping("/home")

public class IndexController {

@GetMapping("/person")

public @ResponseBody ResponseEntity<String> getPerson() {

return new ResponseEntity<String>("Response from GET", HttpStatus.OK);

}

@GetMapping("/person/{id}")

public @ResponseBody ResponseEntity<String> getPersonById(@PathVariable String id){

return new ResponseEntity<String>("Response from GET with id " +id,HttpStatus.OK); }

@PostMapping("/person")

public @ResponseBody ResponseEntity<String> postPerson() {

return new ResponseEntity<String>("Response from POST method", HttpStatus.OK);

}

@PutMapping("/person")

public @ResponseBody ResponseEntity<String> putPerson() {

return new ResponseEntity<String>("Response from PUT method", HttpStatus.OK);

}

@DeleteMapping("/person")

public @ResponseBody ResponseEntity<String> deletePerson() {

return new ResponseEntity<String>("Response from DELETE method", HttpStatus.OK);

}

@PatchMapping("/person")

public @ResponseBody ResponseEntity<String> patchPerson() {

return new ResponseEntity<String>("Response from PATCH method", HttpStatus.OK);

}

In this code, each of the handler methods is annotated with the composed variants of @RequestMapping. Although each variant can be used interchangeably with @RequestMapping with the method attribute, it’s considered a best practice to use the composed variant. Primarily because the composed annotations reduce the configuration metadata on the application side and the code is more readable.

References:

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-requestmapping

<https://springframework.guru/spring-requestmapping-annotation/>

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++=

@RequestBody can be used in combination with javax.validation.Valid, or Spring’s @Validated annotation, which causes Standard Bean Validation to be applied. By default validation errors cause a MethodArgumentNotValidException which is turned into a 400 (BAD\_REQUEST) response. Alternatively, validation errors can be handled locally within the controller through an Errors or BindingResult argument:

@PostMapping("/accounts")

public void handle(@Valid @RequestBody Account account, BindingResult result) {

// ...

}

https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#rest-message-conversion

**Message Conversion**

The spring-web module contains the HttpMessageConverter contract for reading and writing the body of HTTP requests and responses via InputStream and OutputStream. HttpMessageConverter's are used on the client-side, e.g. in the RestTemplate, and also on the server-side, e.g. in Spring MVC REST controllers.

Concrete implementations for the main media (MIME) types are provided in the framework and are registered by default with the RestTemplate on the client-side and with RequestMethodHandlerAdapter on the server-side (see Configuring Message Converters).

**The implementations of HttpMessageConverters are described in the following sections.**

* StringHttpMessageConverter - An HttpMessageConverter implementation that can read and write Strings from the HTTP request and response. By default, this converter supports all text media types ( text/\*) and writes with a Content-Type of text/plain.
* FormHttpMessageConverter - An HttpMessageConverter implementation that can read and write form data from the HTTP request and response. By default, this converter reads and writes the media type application/x-www-form-urlencoded. Form data is read from and written into a MultiValueMap<String, String>.
* ByteArrayHttpMessageConverter - An HttpMessageConverter implementation that can read and write byte arrays from the HTTP request and response. By default, this converter supports all media types ( \*/\*), and writes with a Content-Type of application/octet-stream. This can be overridden by setting the supportedMediaTypes property, and overriding getContentType(byte[]).
* MarshallingHttpMessageConverter - An HttpMessageConverter implementation that can read and write XML using Spring’s Marshaller and Unmarshaller abstractions from the org.springframework.oxm package. This converter requires a Marshaller and Unmarshaller before it can be used. These can be injected via constructor or bean properties. By default, this converter supports ( text/xml) and ( application/xml).
* MappingJackson2HttpMessageConverter - An HttpMessageConverter implementation that can read and write JSON using Jackson’s ObjectMapper. JSON mapping can be customized as needed through the use of Jackson’s provided annotations. When further control is needed, a custom ObjectMapper can be injected through the ObjectMapper property for cases where custom JSON serializers/deserializers need to be provided for specific types. By default, this converter supports ( application/json).
* MappingJackson2XmlHttpMessageConverter - An HttpMessageConverter implementation that can read and write XML using [Jackson XML](https://github.com/FasterXML/jackson-dataformat-xml) extension’s XmlMapper. XML mapping can be customized as needed through the use of JAXB or Jackson’s provided annotations. When further control is needed, a custom XmlMapper can be injected through the ObjectMapper property for cases where custom XML serializers/deserializers need to be provided for specific types. By default, this converter supports ( application/xml).
* SourceHttpMessageConverter - An HttpMessageConverter implementation that can read and writejavax.xml.transform.Source from the HTTP request and response. Only DOMSource, SAXSource, and StreamSource are supported. By default, this converter supports ( text/xml) and ( application/xml).
* BufferedImageHttpMessageConverter - An HttpMessageConverter implementation that can read and writejava.awt.image.BufferedImage from the HTTP request and response. This converter reads and writes the media type supported by the Java I/O API.

References:

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-requestbody

https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#rest-message-conversion

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

References:

https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html

http://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation

**Model Attribute at Method Level:**

@ModelAttribute

public void addAttributes(Model model) {

model.addAttribute("msg", "Welcome to the StackRoute!");

}

Model Attribute as a Method Argument:

@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)

public String submit(@ModelAttribute("employee") Employee employee) {

// Code that uses the employee object

return "employeeView";

}

Model

References:

https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html

http://www.baeldung.com/spring-mvc-and-the-modelattribute-annotation

Discuss below Examples

Model Attribute at Method Level:

@ModelAttribute

public void addAttributes(Model model) {

model.addAttribute("msg", "Welcome to the StackRoute!");

}

Model Attribute as a Method Argument:

@RequestMapping(value = "/addEmployee", method = RequestMethod.POST)

public String submit(@ModelAttribute("employee") Employee employee) {

// Code that uses the employee object

return "employeeView";

}