1. Using Apache wink Refer -https://www.ibm.com/support/knowledgecenter/en/SSAW57\_8.0.0/com.ibm.websphere.nd.doc/info/ae/ae/twbs\_jaxrs\_configwebxml.html
2. https://www.ibm.com/developerworks/library/wa-apachewink1/index.html
3. Using Spring boot then Refer -https://spring.io/guides/gs/rest-service/
4. <https://stackoverflow.com/questions/671118/what-exactly-is-restful-programming>
5. <http://www.springboottutorial.com/creating-rest-service-with-spring-boot>
6. <https://dzone.com/articles/spring-boot-restful-api-documentation-with-swagger>
7. https://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/

You can configure the web.xml file for your web application to enable the JAX-RS application code. You can specify an IBM specific JAX-RS servlet to use to run your JAX-RS code. The web.xml file provides configuration and deployment information for the web components that comprise a web application.

When using servlets, any servlet path that is defined in the web.xml is appended to the base URL. For example, if a root resource has a @javax.ws.rs.Path value of myresource and a servlet path of myservletpath, the final URL of the resource is http://<your\_hostname>:<your Web\_container\_port>/<context\_root\_of\_Web\_application>//myservletpath/myresource.

Using spring boot

If you are using Maven, you can run the application using ./mvnw spring-boot:run. Or you can build the JAR file with ./mvnw clean package. Then you can run the JAR file:

java -jar target/gs-rest-service-0.1.0.jar

|  |  |
| --- | --- |
|  | The procedure above will create a runnable JAR. You can also opt to [build a classic WAR file](https://spring.io/guides/gs/convert-jar-to-war/) instead. |

This code uses Spring 4’s new [@RestController](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation, which marks the class as a controller where every method returns a domain object instead of a view. It’s shorthand for @Controller and @ResponseBody rolled together.

<https://dzone.com/articles/spring-boot-restful-api-documentation-with-swagger>

The Swagger 2 specification, which is known as [OpenAPI specification](https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md" \t "_blank), has several implementations. Currently, [Springfox](https://springfox.github.io/springfox/" \t "_blank) that has replaced Swagger-SpringMVC (Swagger 1.2 and older) is popular for Spring Boot applications. Springfox supports both Swagger 1.2 and 2.0.

 The current version defines a set HTML, JavaScript, and CSS assets to dynamically generate documentation from a Swagger-compliant API

 Besides rendering documentation, Swagger UI allows other API developers or consumers to interact with the API’s resources without having any of the implementation logic in place.

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.6.1</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.6.1</version>

<scope>compile</scope>

</dependency>

. . .

**Configuring Swagger 2 in the Application**

For our application, we will create a Docket bean in a Spring Boot configuration to configure Swagger 2 for the application. A Springfox Docket instance provides the primary API configuration with sensible defaults and convenience methods for configuration. Our Spring Boot configuration class, SwaggerConfig is this.

. . .

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket productApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.select() .apis(RequestHandlerSelectors.basePackage("guru.springframework.controllers"))

.paths(regex("/product.\*"))

.build();

}

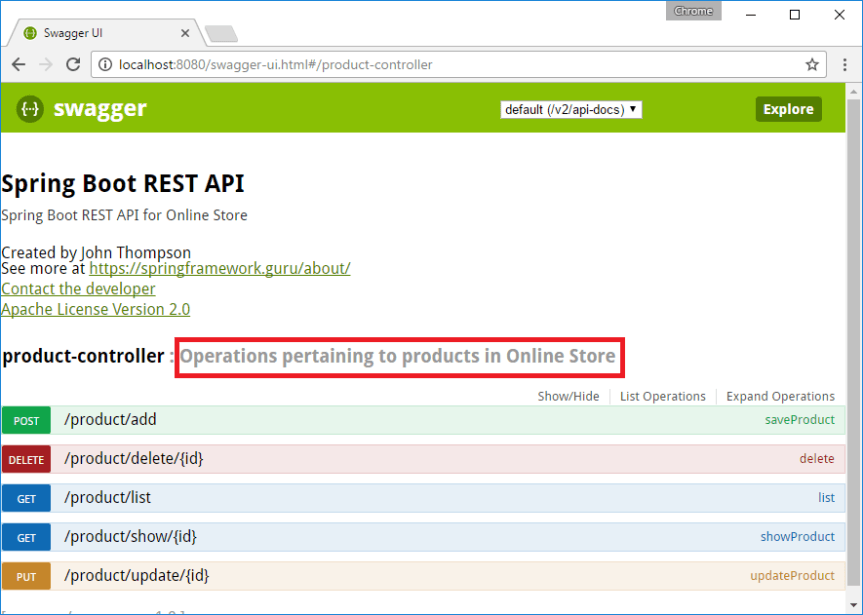
}

. . .

In this configuration class, the @EnableSwagger2 annotation enables Swagger support in the class. The select() method called on the Docket bean instance returns an ApiSelectorBuilder, which provides the apis() and paths() methods that are used to filter the controllers and methods that are being documented using String predicates.

1. We can use the @Api annotation on our ProductController class to describe our API.

@RestController @RequestMapping("/product") @Api(value="onlinestore", description="Operations pertaining to products in Online Store") public class ProductController { . . . . }



1. For each of our operation endpoints, we can use the @ApiOperation annotation to describe the endpoint and its response type, like this:

. . .

@ApiOperation(value = "View a list of available products", response = Iterable.class)

@RequestMapping(value = "/list", method= RequestMethod.GET,produces = "application/json")

public Iterable list(Model model){

Iterable productList = productService.listAllProducts();

return productList;

}

. . .

1. Swagger 2 also allows overriding the default response messages of HTTP methods. You can use the @ApiResponse annotation to document other responses, in addition to the regular HTTP 200 OK, like this.
2. . . .
3. @ApiOperation(value = "View a list of available products", response = Iterable.class)
4. @ApiResponses(value = {
5. @ApiResponse(code = 200, message = "Successfully retrieved list"),
6. @ApiResponse(code = 401, message = "You are not authorized to view the resource"),
7. @ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),
8. @ApiResponse(code = 404, message = "The resource you were trying to reach is not found")
9. }
10. )
11. @RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")
12. public Iterable list(Model model){
13. Iterable productList = productService.listAllProducts();
14. return productList;
15. }
16. . . .

ProductController.java:

package guru.springframework.controllers;

import guru.springframework.domain.Product;

import guru.springframework.services.ProductService;

import io.swagger.annotations.Api;

import io.swagger.annotations.ApiOperation;

import io.swagger.annotations.ApiResponse;

import io.swagger.annotations.ApiResponses;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/product")

@Api(value="onlinestore", description="Operations pertaining to products in Online Store")

public class ProductController {

private ProductService productService;

@Autowired

public void setProductService(ProductService productService) {

this.productService = productService;

}

@ApiOperation(value = "View a list of available products",response = Iterable.class)

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")

public Iterable<Product> list(Model model){

Iterable<Product> productList = productService.listAllProducts();

return productList;

}

@ApiOperation(value = "Search a product with an ID",response = Product.class)

@RequestMapping(value = "/show/{id}", method= RequestMethod.GET, produces = "application/json")

public Product showProduct(@PathVariable Integer id, Model model){

Product product = productService.getProductById(id);

return product;

}

@ApiOperation(value = "Add a product")

@RequestMapping(value = "/add", method = RequestMethod.POST, produces = "application/json")

public ResponseEntity saveProduct(@RequestBody Product product){

productService.saveProduct(product);

return new ResponseEntity("Product saved successfully", HttpStatus.OK);

}

@ApiOperation(value = "Update a product")

@RequestMapping(value = "/update/{id}", method = RequestMethod.PUT, produces = "application/json")

public ResponseEntity updateProduct(@PathVariable Integer id, @RequestBody Product product){

Product storedProduct = productService.getProductById(id);

storedProduct.setDescription(product.getDescription());

storedProduct.setImageUrl(product.getImageUrl());

storedProduct.setPrice(product.getPrice());

productService.saveProduct(storedProduct);

return new ResponseEntity("Product updated successfully", HttpStatus.OK);

}

@ApiOperation(value = "Delete a product")

@RequestMapping(value="/delete/{id}", method = RequestMethod.DELETE, produces = "application/json")

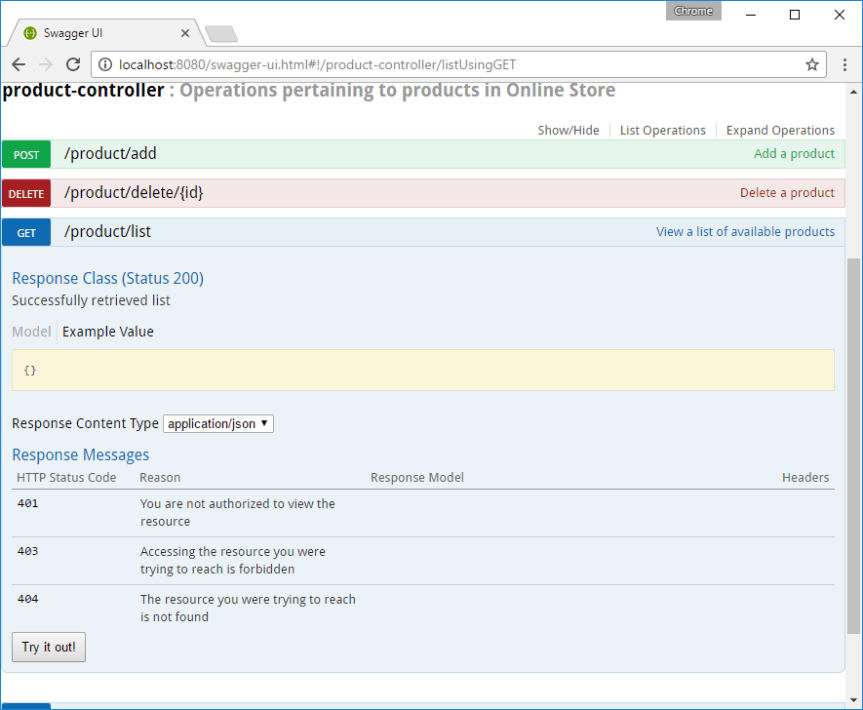
public ResponseEntity delete(@PathVariable Integer id){

productService.deleteProduct(id);

return new ResponseEntity("Product deleted successfully", HttpStatus.OK);

}

}



*@GetMapping("/owners/{ownerId}")*

**public** String findOwner(**@PathVariable** String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

**return** "displayOwner";

}

The URI Template " /owners/{ownerId}`" specifies the variable name `ownerId. When the controller handles this request, the value of ownerId is set to the value found in the appropriate part of the URI. For example, when a request comes in for /owners/fred, the value of ownerId is fred.

|  |
| --- |
| [Tip] |
| To process the @PathVariable annotation, Spring MVC needs to find the matching URI template variable by name. You can specify it in the annotation:  *@GetMapping("/owners/{ownerId}")*  **public** String findOwner(**@PathVariable("ownerId")** String theOwner, Model model) {  *// implementation omitted*  }  Or if the URI template variable name matches the method argument name you can omit that detail. As long as your code is compiled with debugging information or the -parameters compiler flag on Java 8, Spring MVC will match the method argument name to the URI template variable name:  *@GetMapping("/owners/{ownerId}")*  **public** String findOwner(**@PathVariable** String ownerId, Model model) {  *// implementation omitted*  } |

#### Composed @RequestMapping Variants

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

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

By default use of @EnableWebMvc or <mvc:annotation-driven> automatically registers Bean Validation support in Spring MVC through the LocalValidatorFactoryBean when a Bean Validation provider such as Hibernate Validator is detected on the classpath.