Documentar Servicios API Rest con Swagger

Secretaria de relciones exteriores y cooperacion internacional | Bo. El Centro antiguo edificio del Banco Central de Honduras

Implementar swagger en spring boot

Nahum Martinez

2018

DOCUMENTAR UN SERVICIO REST, CON SWAGGER

**ACTUALIZACIÓN 21-12-2015:** La librería ‘swagger-springmvc’ ha sido deprecada, ahora ha pasado a llamarse **SpringFox**. He escrito un [nuevo post](https://anotherdayanotherbug.wordpress.com/2015/12/21/documentar-un-servicio-rest-con-swagger-2-springfox/) sobre como documentar un servicio REST con Swagger usando esa nueva librería SpringFox.

¡Ahora sí! Esta vez si que vamos a usar Swagger, para documentar un servicio REST implementado con Spring, y sobre Spring Boot. Para ello usaremos el servicio que ha preparamos en el [post anterior](https://anotherdayanotherbug.wordpress.com/2014/09/08/implementar-un-servicio-rest-con-spring/).

**Concepto:**  
Dado el auge de los servicios REST, se ha convertido en casi una obligación tener una buena documentación de esos mismos servicios, para que el cliente pueda fácilmente usarlo. Entre otras, [Swagger](http://swagger.io/) parece que está destacando bastante últimamente, como librería para documentar fácilmente servicios REST en java.  
Para más información de Swagger, aquí unos links interesantes:  
<http://swagger.io/>  
<https://github.com/wordnik/swagger-spec>  
<https://github.com/wordnik/swagger-ui>  
<https://github.com/martypitt/swagger-springmvc>

**Entorno usado:**  
Java JDK 1.8  
Maven 3.2.1  
Git 1.9.4  
IDE Intellij 13.1.4 Community version

**Pasos:**

1. Como comenté antes, vamos a usar como base el proyecto que hicimos en el [último post](https://anotherdayanotherbug.wordpress.com/2014/09/08/implementar-un-servicio-rest-con-spring/), en el que teníamos ya implementado un sencillo servicio REST en Spring, sobre un proyecto Spring Boot. Podéis bajaros el proyecto directamente de [github](https://github.com/edwise/java-pocs/tree/master/spring-rest-example" \o "github spring-rest-example" \t "_blank).

2. Añadimos las dependencias maven necesarias a nuestro pom.xml, para poder usar swagger, en concreto swagger-springmvc, que es el que necesitamos para montarlo sobre Spring Boot.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ...     <dependency>         <groupId>com.mangofactory</groupId>         <artifactId>swagger-springmvc</artifactId>         <version>0.8.8</version>     </dependency>  ... |

Con esta dependencia es suficiente para toda la parte ‘back’ de swagger.

3. Para que Swagger funcione, necesitamos definir en nuestro contexto de Spring un bean ‘[SwaggerSpringMvcPlugin](https://github.com/martypitt/swagger-springmvc/blob/master/swagger-springmvc/src/main/java/com/mangofactory/swagger/plugin/SwaggerSpringMvcPlugin.java" \o "SwaggerSpringMvcPlugin" \t "_blank)‘. Para ello, podemos definirlo directamente, en nuestra clase ‘Application.java’ (que la tenemos marcada como [@Configuration](http://docs.spring.io/spring-framework/docs/current/javadoc-api/index.html?org/springframework/context/annotation/Configuration.html)) o crear una nueva clase de configuración. Haremos esto último, para que todo este un poco más organizado:

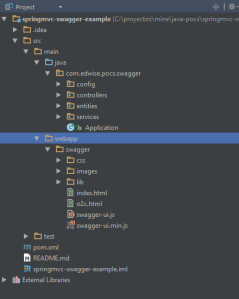
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | package com.edwise.pocs.swagger.config;    import com.mangofactory.swagger.configuration.SpringSwaggerConfig;  import com.mangofactory.swagger.plugin.SwaggerSpringMvcPlugin;  import com.wordnik.swagger.model.ApiInfo;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;    @Configuration  public class SwaggerSpringMvcConfig {        private SpringSwaggerConfig springSwaggerConfig;        @Autowired      public void setSpringSwaggerConfig(SpringSwaggerConfig springSwaggerConfig) {          this.springSwaggerConfig = springSwaggerConfig;      }        @Bean      public SwaggerSpringMvcPlugin customImplementation() {          return new SwaggerSpringMvcPlugin(this.springSwaggerConfig)                  .apiInfo(apiInfo())                  .includePatterns(".\*/\*.\*");      }        private ApiInfo apiInfo() {          return new ApiInfo(                  "Users API",                  "Your user database!",                  "<http://userweb.userapi.com/Terms_of_service>",                  "userapi.manager@gmail.com",                  "Apache 2.0",                  "<http://www.apache.org/licenses/LICENSE-2.0.html>"          );      }  } |

Principalmente lo que hacemos en esta clase de configuración es cargar el bean **SwaggerSpringMvcPlugin**, en el método ‘customImplementation’, añadiéndole cierta información básica de configuración:  
– Con la clase ‘**ApiInfo**‘ le metemos varios textos generales para nuestra api.  
– con el método ‘**includePatterns**‘ filtramos los servicios que queremos que muestre nuestra documentación Swagger (en este caso mostramos todos).

4. Añadimos a nuestra clase base para Spring Boot ‘Application.java’ la anotación [@EnableSwagger](https://github.com/martypitt/swagger-springmvc/blob/b58f20ba5d54837c4ebbd96c89b1e338f0007ebc/swagger-springmvc/src/main/java/com/mangofactory/swagger/plugin/EnableSwagger.java). Tambien podriamos añadirla en la clase creada antes. El único requisito es que la clase sea de configuración (@Configuration).

5. Arrancamos nuestra aplicación Spring Boot como siempre (mvn spring-boot:run o desde la tool window de maven en Intellij). Si accedemos a la url <http://localhost:8080/api-docs> nos devolverá un json con cierta información de nuestros servicios REST (entre otras cosas los datos que le pasamos en ApiInfo en nuestra clase de configuración). Ya tenemos swagger funcionando realmente, pero ahora faltaría documentar un poco nuestro REST, y poder verlo en alguna web un poco en condiciones…

6. Vamos a empezar por bajarnos la interfaz web para swagger. Para ello podemos, o añadir la dependencia maven, o bajarnos toda la interfaz y copiarla en nuestro proyecto. Dado que necesitamos modificarle alguna cosa (ahora lo veremos), lo mejor es lo segundo. Para ello, nos descargamos el repositorio git de [aquí](https://github.com/wordnik/swagger-ui) y copiamos todo el contenido de la carpeta ‘dist’ a una subcarpeta en nuestro proyecto, dentro de ‘src/main/webapp’ llamada ‘swagger’. Nuestra estructura de carpetas quedaría tal que así:

[](https://anotherdayanotherbug.files.wordpress.com/2014/09/screenclip.png)

*Estructura de carpetas*

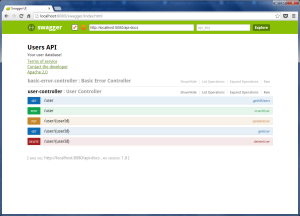
7. Vamos a ‘costumizar’ un poco la interfaz, ya que por defecto apunta al swagger de ejemplo que tienen en la web (la petstore esa…). Abrimos el archivo index.html que tenemos directamente en nuestra subcarpeta ‘swagger’ y buscamos la siguiente linea en código javascript:

|  |  |
| --- | --- |
| 1  2  3 | ...  url = "<http://petstore.swagger.wordnik.com/v2/swagger.json>";      ... |

Y sustituimos esa url por la nuestra, que será así: <http://localhost:8080/api-docs> . O podemos también no hardcodear la url de vuestro server, y dejarlo un poco más bonito sustituyéndolo por algo así:

|  |  |
| --- | --- |
| 1  2  3 | ...  url = window.location.protocol + "//" + window.location.host + "/api-docs";      ... |

Con esto ya podemos arrancar nuestro server y accediendo a <http://localhost:8080/swagger/index.html>, ver la interfaz web de swagger funcionando contra nuestro servicio. Veremos algo como esto:

[](https://anotherdayanotherbug.files.wordpress.com/2014/09/swaggerui_01.png)

*Swagger por defecto*

Entre otras cosas veremos que a nuestro servicio lo ha llamado ‘user-controller’ y que hay otro servicio más, basic-error-controller. Este último es parte de Spring, no le hagamos mucho caso.  
Nos sigue faltando documentar en condiciones nuestro servicio REST (por ahora swagger nos muestra todo con textos por defecto), y ya de paso filtrar los servicios para que muestre solo el nuestro. ¡Vamos con ello!

7. Vamos a filtrar los servicios que mostramos. Podríamos filtrar por el nombre de nuestro servicio, pero para poder añadir otros, vamos a filtrar por la ruta del servicio, que sea ‘/api/loquesea’. Nuestro servicio actual es directamente /user, así que lo primero que vamos a hacer es modificar su url. En nuestra clase UserController modificamos el RequestMapping para dejarlo así:

|  |  |
| --- | --- |
| UserController.java | |
| 1  2  3  4  5 | ...  @RestController  @RequestMapping("/api/user")  public class UserController {  ... |

Y ahora modificamos nuestra clase de configuración de Swagger, SwaggerSpringMvcConfig, para que filtre los servicios por un patrón:

|  |  |
| --- | --- |
| SwaggerSpringMvcConfig.java | |
| 1  2  3  4  5  6  7  8 | ...  @Bean  public SwaggerSpringMvcPlugin customImplementation() {      return new SwaggerSpringMvcPlugin(this.springSwaggerConfig)              .apiInfo(apiInfo())              .includePatterns(".\*api/\*.\*");  }  ... |

Si arrancamos el servidor y accedemos otra vez a <http://localhost:8080/swagger/index.html> veremos que ahora ya no nos muestra nada más que nuestro servicio. (Ojo, la url para acceder al servicio REST ahora es: <http://localhost:8080/api/user>)

8. Ahora vamos a documentar nuestro servicio en condiciones, directamente en nuestras clases java (nada de xmls, ni historias de esas). Para ello swagger proporciona distintas anotaciones java. Vamos a ver y añadir algunas a nuestro servicio:  
Primero documentaremos nuestro modelo, la clase User, quedaría así:

|  |  |
| --- | --- |
| User.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | ...  import com.wordnik.swagger.annotations.ApiModel;  import com.wordnik.swagger.annotations.ApiModelProperty;  import org.joda.time.LocalDate;    @ApiModel(value = "User entity", description = "Complete info of a entity user")  public class User {        @ApiModelProperty(value = "The id of the user")      private Long id;        @ApiModelProperty(value = "The name of the user", required = true)      private String name;        @ApiModelProperty(value = "The type of the user", required = true)      private Integer type;        @ApiModelProperty(value = "The phone of the user", required = false)      private String phone;        @ApiModelProperty(value = "The birthDay of the user", required = false)      private LocalDate birthDate;        // getters, setters, etc  } |

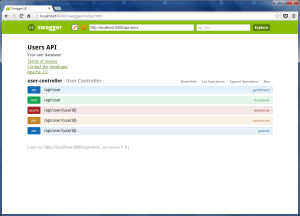
Las anotaciones son bastante ‘autoexplicativas’, pero por aclarar:  
– **@ApiModel** documenta la entity, con una descripción corta (value) y una descripción más larga (description).  
– **@ApiModelProperty** documenta las propiedades, con una descripción (value) y si es obligatoria (required). También contiene otros parámetros de posibles valores (allowableValues), ocultarlo a swagger (hidden) y otros.

Ahora documentamos nuestro controller y cada uno de sus métodos, quedando así:

|  |  |
| --- | --- |
| UserController.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | ...  // imports...    @RestController  @RequestMapping("/api/user")  @Api(value = "users", description = "Users API")  public class UserController {        private static final int RESPONSE\_CODE\_OK = 200;      private static final int RESPONSE\_CODE\_CREATED = 201;      private static final int RESPONSE\_CODE\_NOCONTENT = 204;      private static final int RESPONSE\_CODE\_NOTFOUND = 404;        @Autowired      private UserService userService;        @RequestMapping(method = RequestMethod.GET, produces = MediaType.APPLICATION\_JSON\_VALUE)      @ApiOperation(value = "Get Users", response = List.class, notes = "Returns all users")      @ApiResponses({              @ApiResponse(code = RESPONSE\_CODE\_OK, message = "All users in db")      })      public List getAllUsers() {          return userService.findAll();      }        @RequestMapping(value = "/{userId}", method = RequestMethod.GET, produces = MediaType.APPLICATION\_JSON\_VALUE)      @ApiOperation(value = "Get one User", response = ResponseEntity.class, notes = "Returns one user")      @ApiResponses({              @ApiResponse(code = RESPONSE\_CODE\_OK, message = "Exists this user"),              @ApiResponse(code = RESPONSE\_CODE\_NOTFOUND, message = "No exists this user")      })      public ResponseEntity getUser(@PathVariable long userId) {          User user = userService.findById(userId);          ResponseEntity response;          if (user == null) {              response = new ResponseEntity<>(HttpStatus.NOT\_FOUND);          } else {              response = new ResponseEntity<>(user, HttpStatus.OK);          }          return response;      }        @RequestMapping(method = RequestMethod.POST)      @ResponseStatus(HttpStatus.CREATED)      @ApiOperation(value = "Insert User", notes = "Insert a complete new User")      @ApiResponses({              @ApiResponse(code = RESPONSE\_CODE\_CREATED, message = "User inserted")      })      public void insertUser(@RequestBody User user) {          userService.save(user);      }        @RequestMapping(value = "/{userId}", method = RequestMethod.PUT)      @ResponseStatus(HttpStatus.CREATED)      @ApiOperation(value = "Update User", notes = "Updates a user")      @ApiResponses({              @ApiResponse(code = RESPONSE\_CODE\_CREATED, message = "User updated")      })      public void updateUser(@PathVariable long userId, @RequestBody User user) {          User userOld = userService.findById(userId);          if (userOld != null) {              userService.update(userOld.copyFrom(user));          }      }        @RequestMapping(value = "/{userId}", method = RequestMethod.DELETE)      @ResponseStatus(HttpStatus.NO\_CONTENT)      @ApiOperation(value = "Delete User", notes = "Deletes a user")      @ApiResponses({              @ApiResponse(code = RESPONSE\_CODE\_NOCONTENT, message = "Deleted user")      })      public void deleteUser(@PathVariable long userId) {          userService.delete(userId);      }  } |

Aunque aquí también son bastante legibles las anotaciones, explico un poco:  
– **@Api** documenta el servicio REST en si. Va a ser la descripción que salga en el listado, entre otras cosas.  
– **@ApiOperation** documenta cada método del servicio.  
– **@ApiResponses** documenta las posibles respuestas del método, con mensaje explicativo.  
(Hay alguna anotación más, para los parámetros, etc, pero estas son las que me parecen más interesantes)

9. Arrancamos el servidor, con mvn spring-boot:run o desde la tool window de maven en Intellij. Accedemos a <http://localhost:8080/swagger/index.html> y veremos algo como antes, pero mucho más detallado y documentado

[](https://anotherdayanotherbug.files.wordpress.com/2014/09/swaggerui_02.png)

*Swagger UI filtrada y documentada*

Swagger permite probar el servicio desde su propia interfaz, consultar el modelo a enviar o recibido, etc. Jugad un poco con él