Laboratorio 10

Swagger: Documenta APIs REST - Cómo construir microservicios con Spring Boot

Swagger es un framework que resulta muy útil para **documentar, visualizar y consumir** servicios **REST** . El objetivo de *Swagger* es que la documentación del ***API RESTFul*** se vaya actualizando cada vez que se realicen cambios en el servidor.
Este framework ofrece una ***interfaz visual*** a modo de *sandbox* que permite probar las llamadas a las operaciones del API RESTFul así como consultar su documentación (métodos, parámetros y estructura JSON del modelo)

1 - Pom.xml

Para trabajar con *Swagger,* lo primero de todo será añadir las siguientes dependencias en el fichero ***pom.xml*** del proyecto:

```
<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger2</artifactId>
   <version>${springfox.version}</version>
</dependency>

<dependency>
   <groupId>io.springfox</groupId>
   <artifactId>springfox-swagger-ui</artifactId>
   <version>${springfox.version}</version>
</dependency>
```

2 - Configuración de Swagger

Añadir al proyecto la siguiente clase:

```
package net.maint.microservices.users.config;

import com.google.common.base.Predicate;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
```

```
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import static springfox.documentation.builders.PathSelectors.regex;


@EnableSwagger2
@Configuration
public class SwaggerConfiguration {

  /**
   * Publish a bean to generate swagger2 endpoints
   * @return a swagger configuration bean
   */
  @Bean
  public Docket usersApi() {
    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(usersApiInfo())
        .select()
        .paths(userPaths())
        .apis(RequestHandlerSelectors.any())
        .build()
        .useDefaultResponseMessages(false);
  }

  private ApiInfo usersApiInfo() {
    return new ApiInfoBuilder()
        .title("Service User")
        .version("1.0")
        .license("Apache License Version 2.0")
        .build();
  }

  private Predicate<String> userPaths() {
    return regex("/user.*");
  }
}
```

Para documentar el API, indicamos que se utilice la ***especificación 2.0*** de *Swagger*

3 - Documentación del modelo

Modificamos la clase del modelo *User* de la siguiente manera:

```
@ApiModel("Model User")
public class User{

  @Id
```

```
    @NotNull
    @ApiModelProperty(value = "the user's id", required = true)
    private String userId;

    @NotNull
    @ApiModelProperty(value = "the user's name", required = true)
    private String name;
}
```

@ApiModel proporciona información adicional sobre modelos *Swagger* mientras que la anotación ***@ApiModelProperty*** permite describir una propiedad de una clase del modelo e indicar, por ejemplo, si éste es o no obligatorio (en nuestro ejemplo estamos usando la anotación *@NotNull* para indicar que ambas propiedades del modelo no pueden contener un valor nulo).

4 - Documentación API RESTFul

Actualizamos la clase *UsersController* para documentar las operaciones *REST:*

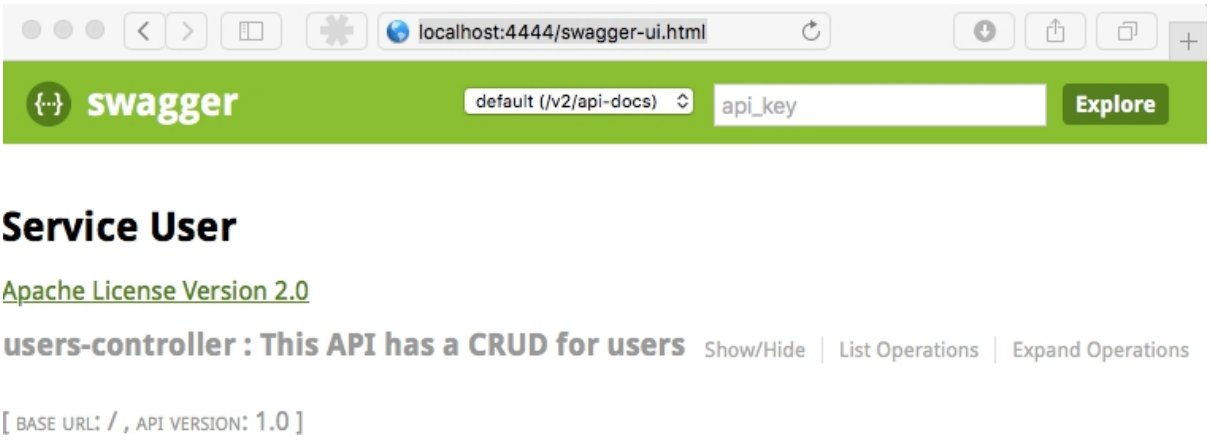
```
@RestController
@RequestMapping("users")
@Api(value = "Users microservice", description = "This API has a CRUD for users")
public class UsersController {


    private static final Log log = LogFactory.getLog(UsersController.class);

    private final UserService usersService;
    private User user;

    @Autowired
    public UsersController(UserService usersService) {
        this.usersService = usersService;
    }

    /**
     * Get a User by userId
     * @param userId
     * @return a controller
     */
    @RequestMapping(value="/{userId}",method = RequestMethod.GET)
    @ApiOperation(value = "Find an user", notes = "Return a user by Id" )
    public ResponseEntity<User> userById(@PathVariable String userId)  throws
UserNotFoundException{
        log.info("Get userById");
```

```
    try{
        user = usersService.findByUserId(userId);
    }catch(UserNotFoundException e){
        user = null;
    }
    return ResponseEntity.ok(user);

}

/**
 * Get all Users
 * @return a controller
 */
@RequestMapping(method = RequestMethod.GET)
@ApiOperation(value = "Find all user", notes = "Return all users" )
public ResponseEntity<List<User>> userById(){
    log.info("Get allUsers");
    return ResponseEntity.ok(usersService.findAll());
}


/**
 * Delete an user by Id
 * @param userId
 * @return empty response
 */
@RequestMapping(value="/{userId}",method = RequestMethod.DELETE)
@ApiOperation(value = "Delete an user", notes = "Delete a user by Id")
public ResponseEntity<Void> deleteUser(@PathVariable String userId){
        log.info("Delete user " + userId);
    usersService.deleteUser(userId);
    return ResponseEntity.noContent().build();
}

/**
 * Save a new user
 * @param user
 * @return
 */
@RequestMapping(method=RequestMethod.POST)
@ApiOperation(value = "Create an user", notes = "Create a new user")
public  ResponseEntity<User> saveUser(@RequestBody @Valid User user){
        log.info("Save new user");
     return ResponseEntity.ok(usersService.saveUser(user));
}

/**
```

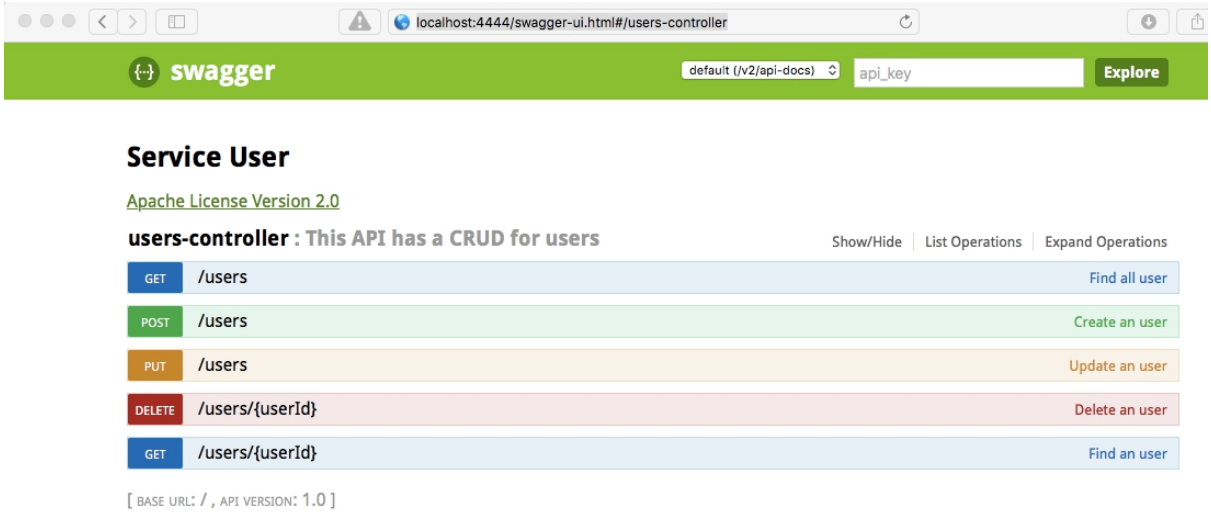
```
     * Update an user
     * @param user
     * @return empty response
     */
    @RequestMapping(method = RequestMethod.PUT)
    @ApiOperation(value = "Update an user", notes = "Update an user by Id")
    public ResponseEntity<Void> updateUser(@RequestBody @Valid User user){
            log.info("update user " + user.getUserId());
        usersService.updateUser(user);
        return ResponseEntity.noContent().build();
    }

}
```

5 - Visualizar y Probar API RESTful

Por último, ya sólo queda ejecutar el microservicio y acceder al API generado por Swagger. Para ello, bastará con añadir al *endpoint* del servicio "***/swagger-ui.html***". Si todo fue bien, deberías ver una pantalla como la siguiente:

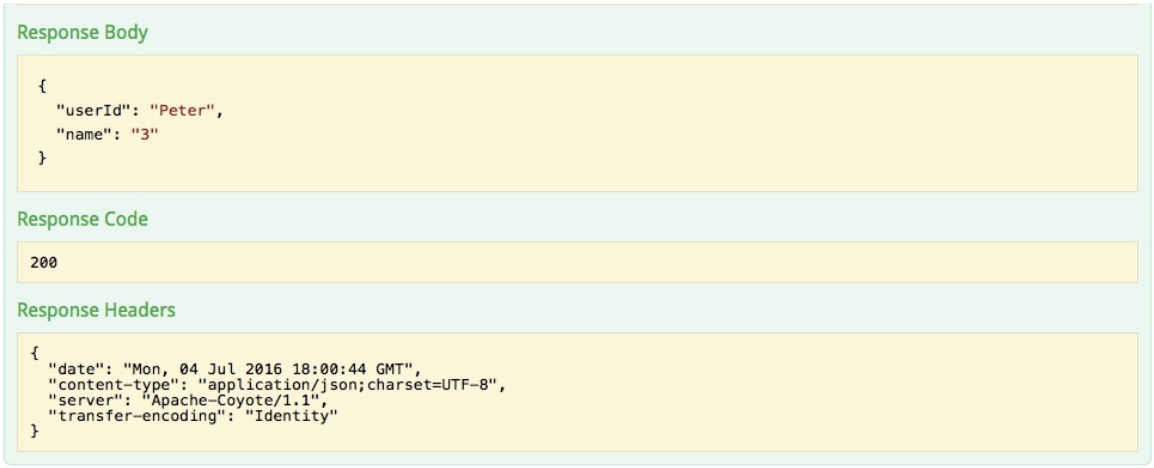


En este punto, puedes hacer clic sobre el API *users-controller* para ver la documentación generada y probar las operaciones de dicho API.



Puedes hacer clic sobre cada operación y ver su documentación detallada (descripción, parámetros de entrada, esquema JSON del modelo, etc)

Si quieres **probar una operación**, es tan sencillo como rellenar los parámetros de entrada siguiendo el *Model Schema* y hacer clic en el botón *Try it out!*. A continuación, te muestro un ejemplo del resultado a la hora de dar de alta un nuevo usuario mediante un *POST*



Te animo a seguir probando el resto de operaciones REST y familiarizándote con este interesante framework de documentación de APIs REST