

# Implementación de Swagger en SpringBoot

En esta guía se va a realizar una breve explicación de como implementar el flujo de "API-first development" en un proyecto usando SpringBoot.

- [API-first development - Creando endpoints y modelos a partir de una especificación en Swagger](#)
  - [Creando el proyecto con SpringBoot inicializr](#)
  - [Preparando el entorno](#)
    - [Comentarios sobre estos archivos y esta estructura.](#)
    - [Explicando el pom.xml](#)
  - [Posible organización del proyecto](#)
  - [Creando el proyecto](#)
  - [Visualizando la nueva estructura del proyecto VS la antigua](#)
  - [Ejecutando el proyecto SpringBoot](#)
  - [Implementado la lógica](#)
  - [Modificando la documentación](#)
  - [¿Qué faltaría documentar/discutir?](#)
- [Swagger-UI: Creando una web con la documentación](#)

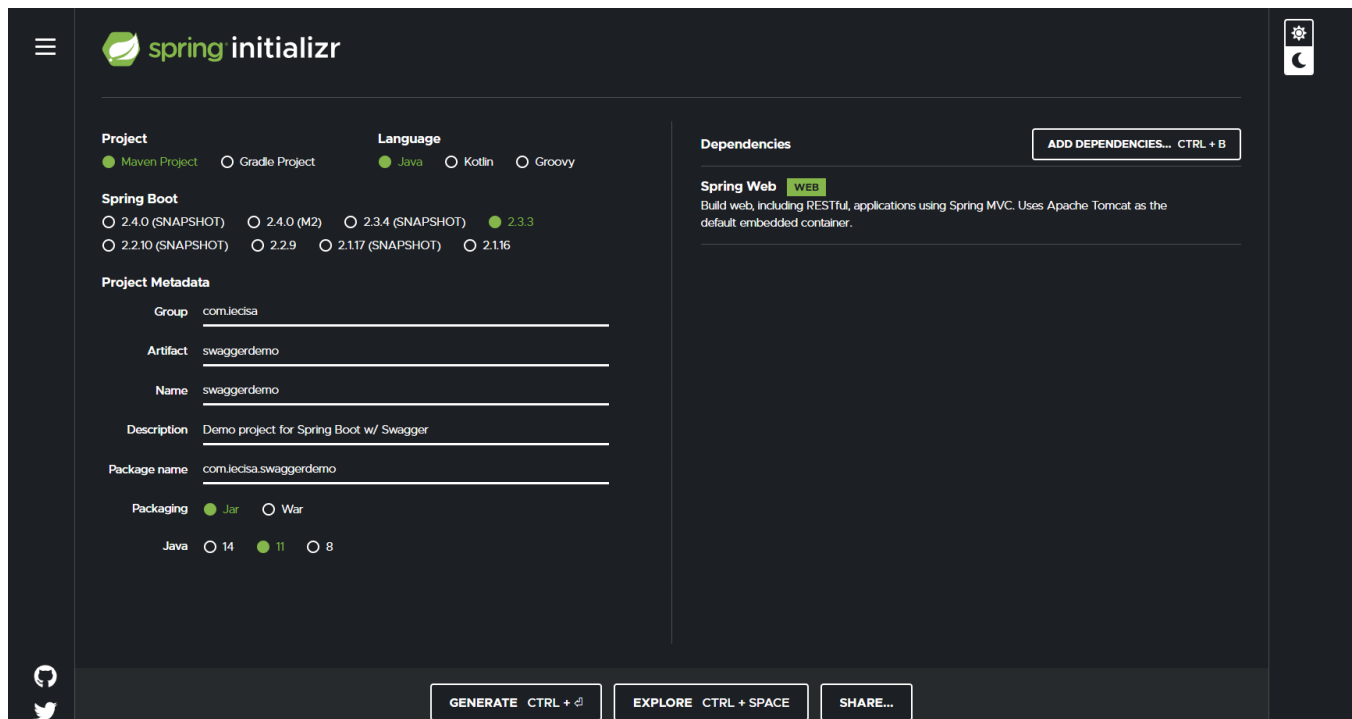
[Código de ejemplo para esta guía](#)

## API-first development - Creando endpoints y modelos a partir de una especificación en Swagger

### Creando el proyecto con SpringBoot inicializr

Para generar el código, lo más cómodo es tener ya una aplicación de SpringBoot creada. Recomiendo usar lo que muestro a continuación ya que tendrá las versiones más actualizadas

Usando la herramienta oficial de SpringBoot Initializr [aquí](#), se puede especificar versiones, nombre, librerías... que se quieran utilizar. En este caso, usaremos la librería de Spring Web para poder crear una aplicación RESTful.



The screenshot shows the Spring Initializr web application interface. It has a dark theme with a sidebar on the left containing a menu icon and social media links. The main content area is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for various versions: 2.4.0 (SNAPSHOT), 2.4.0 (M2), 2.3.4 (SNAPSHOT), **2.3.3** (selected), 2.2.10 (SNAPSHOT), 2.2.9, 2.1.17 (SNAPSHOT), and 2.1.16.
- Project Metadata:** Includes input fields for **Group** (com.iesda), **Artifact** (swaggerdemo), **Name** (swaggerdemo), **Description** (Demo project for Spring Boot w/ Swagger), and **Package name** (com.iesda.swaggerdemo).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Java:** Includes radio buttons for 14, **11** (selected), and 8.
- Dependencies:** Includes a section for **Spring Web** with a **WEB** tag and a description: "Build web, including RESTful applications using Spring MVC. Uses Apache Tomcat as the default embedded container." There is an **ADD DEPENDENCIES... CTRL + B** button.

At the bottom, there are three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

### Preparando el entorno

Ahora que tenemos el proyecto, tendremos que crear la especificación. **A partir de esta especificación** conteniendo la documentación de nuestro servicio **crearemos nuestros modelos y controladores para el proyecto automáticamente**. Para ello, recomiendo tener todo en una carpeta como muestro a continuación.

```
.
specification
  mvnw
  mvnw.cmd
  pom.xml
  specification.yml
```

Esta estructura de proyecto base se puede descargar aquí:

[skeleton.zip](#)

Aunque lo **recomendable**, es sólo descargarse el pom.xml. Los archivos mvnw y mvnw.cmd son opcionales.

[pom.xml](#) - He usado las últimas versiones a Agosto 2020, pero se pueden editar si se quiere. Para más detalles sobre este fichero se puede mirar el apartado [Explicando el pom.xml](#)

### Comentarios sobre estos archivos y esta estructura.

- Se facilitan el wrapper de maven (mvnw o mvnw.cmd) por si se quisiese usar el wrapper de maven.
- La especificación de la documentación esta en formato .yml, pero puede ser perfectamente un archivo .yaml. Además, un archivo .yml o .yaml puede contener un formato .json y funcionar perfectamente. Eso es a gusto del desarrollador.

### Explicando el pom.xml

Esta es la estructura básica que está en nuestro [pom.xml](#) y que nos permite crear el esqueleto del proyecto en SpringBoot. He metido varias opciones como ejemplo, pero se pueden depende de que ajustes se quieren realizar.

```
<plugin>
  <!--https://github.com/OpenAPITools/openapi-generator/tree/master/modules/openapi-generator-maven-plugin-->
  <groupId>org.openapitools</groupId>
  <artifactId>openapi-generator-maven-plugin</artifactId>
  <version>4.2.3</version>
  <executions>
    <execution>
      <goals>
        <goal>generate</goal>
      </goals>
      <configuration>

        <inputSpec>${project.basedir}/src/main/resources/specification.yml</inputSpec> 1
        <generatorName>spring</generatorName> 2
        <output>${project.basedir}/../app</output> 3
        <apiPackage>com.swaggerdemo.rest</apiPackage> 4
        <modelPackage>com.swaggerdemo.model</modelPackage> 5

        <supportingFilesToGenerate>ApiUtil.java</supportingFilesToGenerate>
        <configOptions>
          <!--https://github.com/OpenAPITools/openapi-generator/blob/master/docs/generators/spring.md-->
          <title>Demo API</title>
          <java8>true</java8> 6
          <delegatePattern>true</delegatePattern>
          <artifactDescription>Small API service</artifactDescription>
        </configOptions>
      </configuration>
    </execution>
  </executions>
</plugin>

<plugin>
```

1. Ruta al fichero de especificación del Swagger. El valor de este atributo debería ser: `${project.baseDir}/specification.yml` para nuestro ejemplo
2. Tecnología del proyecto. Springboot por defecto aunque puede ser Go, Node, C...
3. Directorio donde se encontrará el proyecto con el código

4. Donde se guardarán los controladores creados automáticamente. En este ejemplo están mal, debería ser algo como: com.ORGANIZACION.NOMBRE\_APP.rest
5. Donde se guardarán los modelos creados automáticamente. En este ejemplo están mal, debería ser algo como: com.ORGANIZACION.NOMBRE\_APP.model
6. "Use Java8 classes instead of third party equivalents. Starting in 5.x, JDK8 is the default and the support for JDK7, JDK6 has been dropped"

Se pueden ver más opciones para añadir a este archivo aquí:

- <https://github.com/OpenAPITools/openapi-generator/tree/master/modules/openapi-generator-maven-plugin>
- <https://github.com/OpenAPITools/openapi-generator/blob/master/docs/generators/spring.md>

## Posible organización del proyecto

Yo organizaría el proyecto a partir de aquí con una sola carpeta con el nombre del proyecto que contuviese las siguientes carpetas:

1. specification: para la documentación y generación de código
2. app: Implementación de la lógica del código.

En mi caso quedaría de la siguiente forma, aunque el diseño de estructura es a gusto del desarrollador, siempre y cuando se configuren los paths correspondientes:

```
NOMBRE_PROYECTO
.git # Todo el proyecto se subiría a un solo repositorio
.gitignore
specification
  mvnw
  mvnw.cmd
  pom.xml
  specification.yml
app
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
  src
    main
      java
        com
          ORGANIZACION
            swaggerdemo
              SwaggerdemoApplication.java
            resources
              application.properties
              static
              templates
```

## Creando el proyecto

Una vez con esta estructura, asegúrate que el `pom.xml` tiene los atributos `output`, `apiPackage` y `modelPackage` son los correctos para que coincidan con las rutas adecuadas. Una vez editados los atributos podemos crear los controladores y los beans a partir de la documentación usando las herramientas de OpenAPI generator. Para hacer esto simplemente ejecutamos el siguiente comando en la carpeta `specification`:

```
mvn install
```

Para leer la documentación oficial del generador de código podemos ver: <https://github.com/OpenAPITools/openapi-generator> o si se quiere ver una lista de blogs o videos explicando este proceso: <https://github.com/OpenAPITools/openapi-generator#5---presentationsvideostutorialsbooks>

Con este comando, conseguimos que la librería de **OpenAPI genere todos los controladores y beans/modelos de nuestro proyecto**. En mi caso, se genera los controladores en un paquete llamado `rest` y los beans/modelos en un paquete llamado `model` como podemos ver aquí:

```
app
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
  src
    main
      java
```

```

com
  ORGANIZACION
    swaggerdemo
      model
        ApiError.java
        UserBean.java
        User.java
      rest
        ApiUtil.java
        UsersApiController.java
        UsersApi.java
        UsersApiDelegate.java
        SwaggerdemoApplication.java
  org
    openapitools
      configuration
resources
  application.properties
  static
  templates

```

Las clases generadas en el paquete model, son Beans básicos que contienen todas las anotaciones que hayamos especificado en el .yaml. Es decir, si decimos que un atributo es de tipo email, habrá una anotación donde indicará que es de tipo email. Para ello usa la librería javax.annotations. Es por ello que importante escribir bien la especificación. Para ello se puede usar la documentación oficial de Swagger: <https://swagger.io/docs/specification/about/>. Además estos Beans vienen con setters y getters y funciones auxiliares como toString() y toEquals().

Veamos un ejemplo de como escribir buena documentación. Digamos que tenemos un endpoint GET que sea /user/{email}. Está claro que esperamos un email en la URI y que ese email se valide. Por ello lo tendríamos que dejar reflejado que esa variable es de tipo email. ¿Cómo podemos conseguir esto? Simplemente leyendo la documentación de Swagger - Data Type vemos que habría indicar que el email es de tipo string con formato email como se puede ver a continuación.

#### GET /users en la especificación

```

'/users/{email}':
  get:
    tags:
      - user-controller
    summary: Find an user by email
    description: Returns the user
    operationId: getUserByEmail
    produces:
      - application/json
    parameters:
      - name: email
        in: path
        description: The unique email to identify the entity
        required: true
        type: string
        format: email
    responses:
      '200':
        description: OK
        schema:
          $ref: '#/definitions/User'
      '404':
        description: User not found
        schema:
          $ref: '#/definitions/ApiError'
    deprecated: false

```

Hay más tipos de parámetro. Supongamos que queremos trabajar con una fecha. Haciendo una consulta a la documentación oficial vemos que podemos usar: date o date-time en el campo format.

Si por algún motivo tenemos que añadir alguna función a los Beans o añadir constructores, deberemos crear una clase secundaria que herede del Bean al que queremos añadir nuevos métodos. Sugiero como norma añadir el sufijo 'Impl' para nombrar estas clases. Por ejemplo, el Bean User, quedaría UserImpl o el Bean EmployeeBean quedaría EmployeeBeanImpl.

Por otro lado vemos que en el paquete rest tenemos multitud de archivos. Imaginemos que tuviésemos dos controladores en nuestra aplicación llamado Users y Foo. Al generar los controladores obtendríamos lo siguiente:

```

rest
  ApiUtil.java
  FooApiController.java
  FooApi.java
  FooApiDelegate.java
  UsersApiController.java
  UsersApi.java
  UsersApiDelegate.java

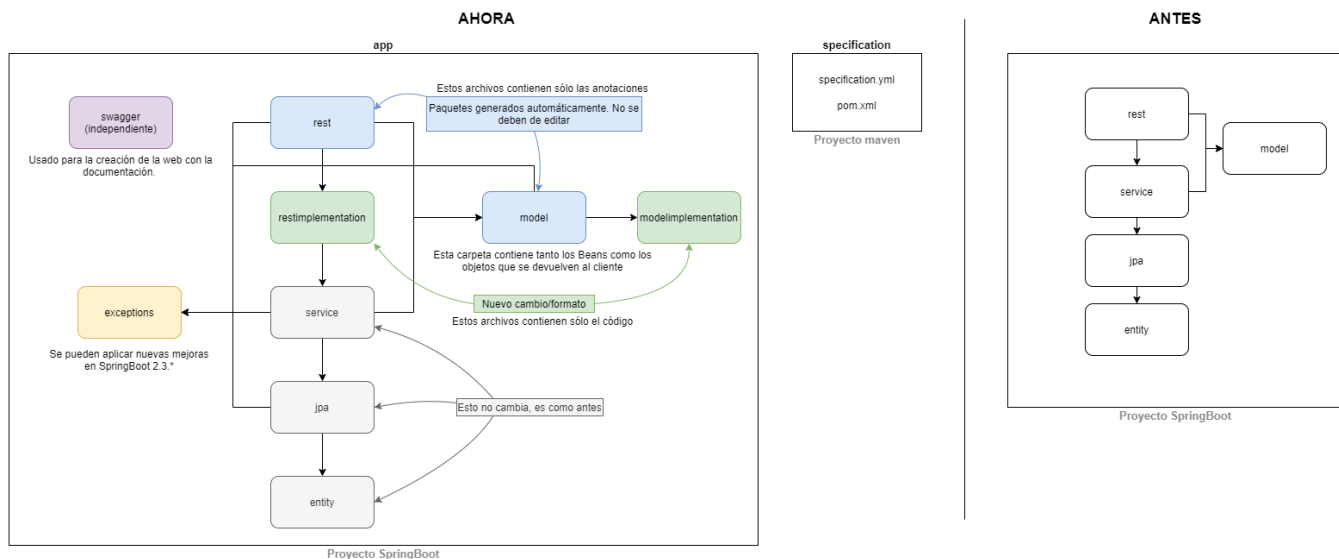
```

El ApiUtil es general para todos los controladores. Simplemente se encarga de crear una response de ejemplo. Por otro lado, por cada controlador que tengamos en nuestra aplicación tendremos tres archivos:

- \*ApiController.java: Este será el punto donde se define el controlador como tal. Implementa los \*Api.java. Su función es recibir las peticiones y delegarlas a la clase de \*Delegate.java
- \*Api.java: Es una interfaz que define el controlador. Contiene todas las anotaciones para SpringBoot y de Swagger.
- \*Delegate.java: Es una interfaz que se ejecuta cuando el \*ApiController.java lo invoca. Por defecto devuelve un 501 NOT\_IMPLEMENTED si no se ha hecho nada. Esta interfaz es la que implementaremos en otro paquete.

## Visualizando la nueva estructura del proyecto VS la antigua

Añadiendo los paquetes relacionados con Swagger, hemos añadido bastantes archivos nuevos y una nueva estructura. A continuación se muestra un gráfico de como los distintos paquetes se relacionan entre ellos y las nuevas actualizaciones.



Editable: [updates.drawio](#)

## Ejecutando el proyecto SpringBoot

Y ejecutando maven por terminal o con el IDE (en la carpeta del proyecto):

```

mvn clean install
java -jar target/*.jar

```

NOTA: Recuerda añadir el resto de dependencias que necesites a tú proyecto (como por ejemplo la dependencia para la conexión a tu bbdd).

Para probar que todo va bien podemos realizar una petición HTTP y ver el status de la respuesta:

```

$max curl -I http://localhost:8080/users/
HTTP/1.1 501
Content-Length: 0
Date: Tue, 01 Sep 2020 11:53:43 GMT
Connection: close

```

Si repasamos nuestro código vemos que todos nuestros endpoints pueden recibir las request y crear los beans correctamente, pero al no haber implementado aún el código, el servidor nos devuelve correctamente un 501 - NOT\_IMPLEMENTED.

## Implementado la lógica

Para implementar la lógica, lo que haremos será implementar una interfaz por cada controlador. En concreto implementaremos los \*Delegate para los controladores y los todos los modelos/beans que nos hiciesen falta. Además, para tener una estructura del código más limpia, lo realizaremos en un paquete a parte que se puede llamar como se desee, en mi caso, `restimplementation` y `modelimplementation` respectivamente. Para implementar una interfaz, podemos ver alguno de los dos ejemplos:

### UsersApiDelegateImpl.java

```
package com.ORGANIZACION.swaggerdemo.restimplementation;

// imports ...

@Component
public class UsersApiDelegateImpl implements UsersApiDelegate {
    @Override
    public ResponseEntity<User> getUser(Integer userId) {
        User user = new User();
        user.setId(userId);
        user.setName("Petros");
        user.setEmail("petros@mail.com");
        user.setAge(20);
        return ResponseEntity.ok(user);
    }
}
```

### UserImpl.java

```
package com.ORGANIZACION.swaggerdemo.modelimplementation;

// imports ...

public class UserImpl extends User {

    public UserImpl() {
        super();
    }

    // Esto es solo un ejemplo de algún método que se debe de implementar...
    public UserImpl fromEntity(UserEntity user) {
        this.setName(user.getName());
        this.setEmail(user.getEmail());
        this.setAge(user.getAge());
        return this;
    }
}
```

Nos podemos dar cuenta, que al estar creando todas las anotaciones a partir de la especificación de Swagger, no debemos **preocuparnos nunca** más de tener algo mal ahí **ni de tener que EDITARLO**. Si queremos cambiar algo de algún Bean o endpoint lo deberemos cambiar siempre en la especificación. Nosotros solo nos preocuparemos a partir de este punto de:

1. Escribir la lógica del código.
2. Modificar la documentación si hay que realizar algún ajuste y volver a ejecutar el openapi-generator para generar las clases correspondientes.

Ahora si ejecutamos la llamada GET a `/user/10`, nos devolverá el objeto User que hemos puesto explícitamente en el código.

Veamos un ejemplo de cómo podemos implementar el código. Digamos que tengo un GET `/user/{id}` en mi aplicación, por lo tanto tendré un model User.java que se habrá creado automáticamente y además un endpoint en los controladores también creado automáticamente. Suponemos que el id es un Integer, por lo que si intentamos realizar un `/user/String` nos devuelve un error porque las anotaciones están funcionando perfectamente. Sin embargo para implementar el código tendríamos que tener un servicio con la lógica y un archivo que implementase el rest. Aquí podemos ver un ejemplo de ambos archivos:

### UsersApiDelegateImpl.java

```

package com.ORGANIZACION.swaggerdemo.restimplementation;

import com.ORGANIZACION.swaggerdemo.rest.UsersApiDelegate;
import com.ORGANIZACION.swaggerdemo.services.UserService;
// Omitimos el resto imports

@Component
public class UsersApiDelegateImpl implements UsersApiDelegate {

    @Autowired
    private UserService userService;

    @Override
    public ResponseEntity<User> getUser(Integer userId) {
        return ResponseEntity.ok(userService.getUser(userId));
    }

    // Rest de implementaciones
}

```

### UserService.java

```

package com.ORGANIZACION.swaggerdemo.services;

// Omitimos imports

@Service
public class UserService {
    private UserRepository repository;

    UserService(UserRepository repository) {
        this.repository = repository;
    }

    public User getUser(Integer userId) {
        return repository.findById(userId)
            .orElseThrow(() -> new UserNotFoundException(userId));
    }
}

```

## Modificando la documentación

Según se va desarrollando el la aplicación, nos vamos dando cuenta de que es necesario ir realizando cambios o añadir nuevas funciones que implican la modificación de los endpoints o de los modelos. Para ello, simplemente deberemos modificar la especificación Swagger, ejecutar el proyecto maven que tenemos en nuestra carpeta `specification`, que creara de nuevo todos los archivos en `model` y en `rest`. Sin embargo, dejará el resto de paquetes sin modificar. Posteriormente, nosotros podremos añadir las distintas implementaciones: o bien a `restimplementation` o a `modelimplementation`.

## ¿Qué faltaría documentar/discutir?

Lo explicado hasta ahora, cubre bastante nuestras necesidades, pero habría que seguir investigando en ciertos aspectos:

- ¿Cómo se trabaja con un JSON multinivel? Si nuestra API recibe un objeto con varios niveles de profundidad porque lo queremos realizar así, ¿habría que cambiar algo? ¿Se crearía un HashMap dentro de un HashMap automático, lo haría usando una clase por cada nivel de profundidad o de otro modo?
- Si usamos este proceso para la creación de código a partir de ahora, hay que ser más conscientes y más perfeccionistas al crear las especificaciones de la documentación, ya que afectará directamente al código. Por ejemplo, si ponemos que un atributo email es de tipo String, sería correcto pero no se traducirá en que el Bean tenga que validar que el atributo es de tipo email. Para ello, hay que indicarle en la especificación de que el atributo tiene un formato de email.

## Swagger-UI: Creando una web con la documentación

Lo que hemos hecho hasta ahora es que a partir de una especificación Swagger crear un código base e implementar la lógica. ¿Pero si queremos la documentación a partir del código? También podemos hacer esto. Para ello, usaremos dos librerías que se pueden añadir al `pom.xml` del proyecto:

## pom.xml - En el proyecto con la implementación

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>

<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

La primera librería nos crea automáticamente un endpoint([http://example.org/BASE\\_PATH/v2/api-docs](http://example.org/BASE_PATH/v2/api-docs)) que responde con un JSON conteniendo la especificación de nuestro código. Esto lo hace leyendo las anotaciones en los controladores y en los beans. Por otro lado, la librería Swagger UI, crea una web que consume el endpoint y lo muestra gráficamente. Además, esta web nos permite interactuar con la API de forma directa sin tener que estar usando Swagger, dando clicks, se pueden cargar los modelos ejemplo, editarlos y realizar la petición mucho más rápido que en Postman.

Además de añadir las dos librerías mencionadas anteriormente, hay que añadir un archivo de configuración de Swagger que lo guardaremos en un paquete nuevo con el nombre que se quiera, por ejemplo, swagger. El archivo de configuración lo podéis copiar de internet o usando como base el que muestro a continuación:

## SwaggerConfiguration.java

```
package com.swagger.swaggerdemoapi.swagger;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.service.Contact;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

import java.util.Collections;

@Configuration
@EnableSwagger2
public class SwaggerConfiguration {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2).select()
            .apis(RequestHandlerSelectors.basePackage("com.swagger.swaggerdemoapi.rest")) // Se puede usar
            un array para indicar varios
            .paths(PathSelectors.any())
            .build()
            .apiInfo(apiInfo())
            .useDefaultResponseMessages(false);
    }

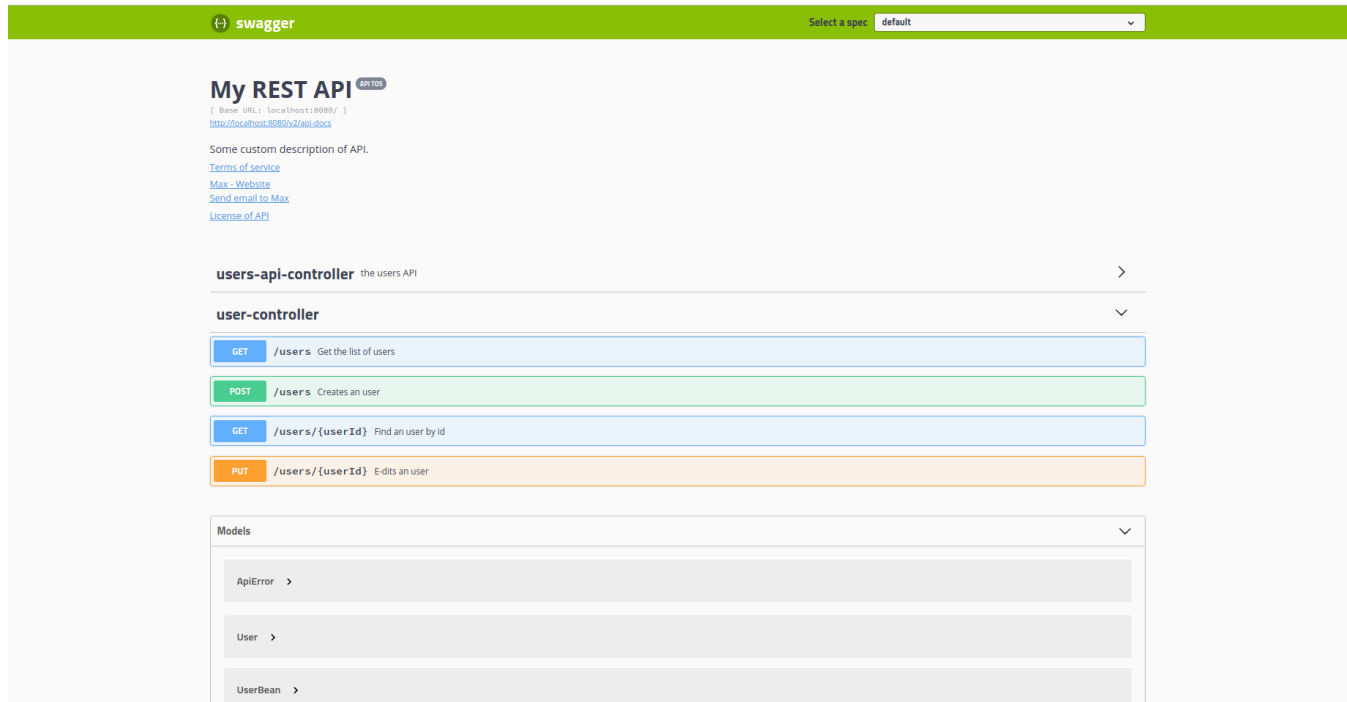
    private ApiInfo apiInfo() {
        ApiInfo apiInfo = new ApiInfo("My REST API", "Some custom description of API.",
            "API TOS", "Terms of service",
            new Contact("Max", "www.github.com", "myeaddress@company.com"),
            "License of API", "API license URL", Collections.emptyList());
        return apiInfo;
    }
}
```

Recuerda editar los paquetes y Strings que necesites para que funcione en tu proyecto. Una vez hecho esto, compilamos y ejecutamos:



```
mvn clean install
java -jar target/*.jar
```

Y yendo a la ruta [http://localhost:8000/BASE\\_PATH/swagger-ui.html](http://localhost:8000/BASE_PATH/swagger-ui.html) deberíamos ver la interfaz de Swagger con los controladores que le especificamos. Si no especificamos el BASE\_PATH, solo poner `/swagger-ui.html`. En esta página veríamos algo como esto:



Desde aquí podríamos realizar las peticiones como si estuviésemos en Postman pero sin preocuparnos de las URI o de los headers. Solo del payload o Path params. Además, habría que editar solo las peticiones porque habría ya valores por defecto para cada variable que hay que enviar

## Código de ejemplo para esta guía

El código usado en este proyecto se puede ver aquí:

<https://cis.ieci-servicios.com/bitbucket/projects/BTF/repos/testswaggar/browse>

Además he añadido una carpeta con un proyecto Swagger implementando nuevas funcionalidades de SpringBoot:

- En los servicios, el uso de repositorios es mejor y actualizado a la versión 2.3.3
- En las excepciones se ha creado una cierta estructura:
  - Una clase común llamada `ApiError`
  - Una clase para cada tipo de error:
    - Error de servidor
    - Error a la creando beans
    - Error general como un error en SQL (`NotFound`)