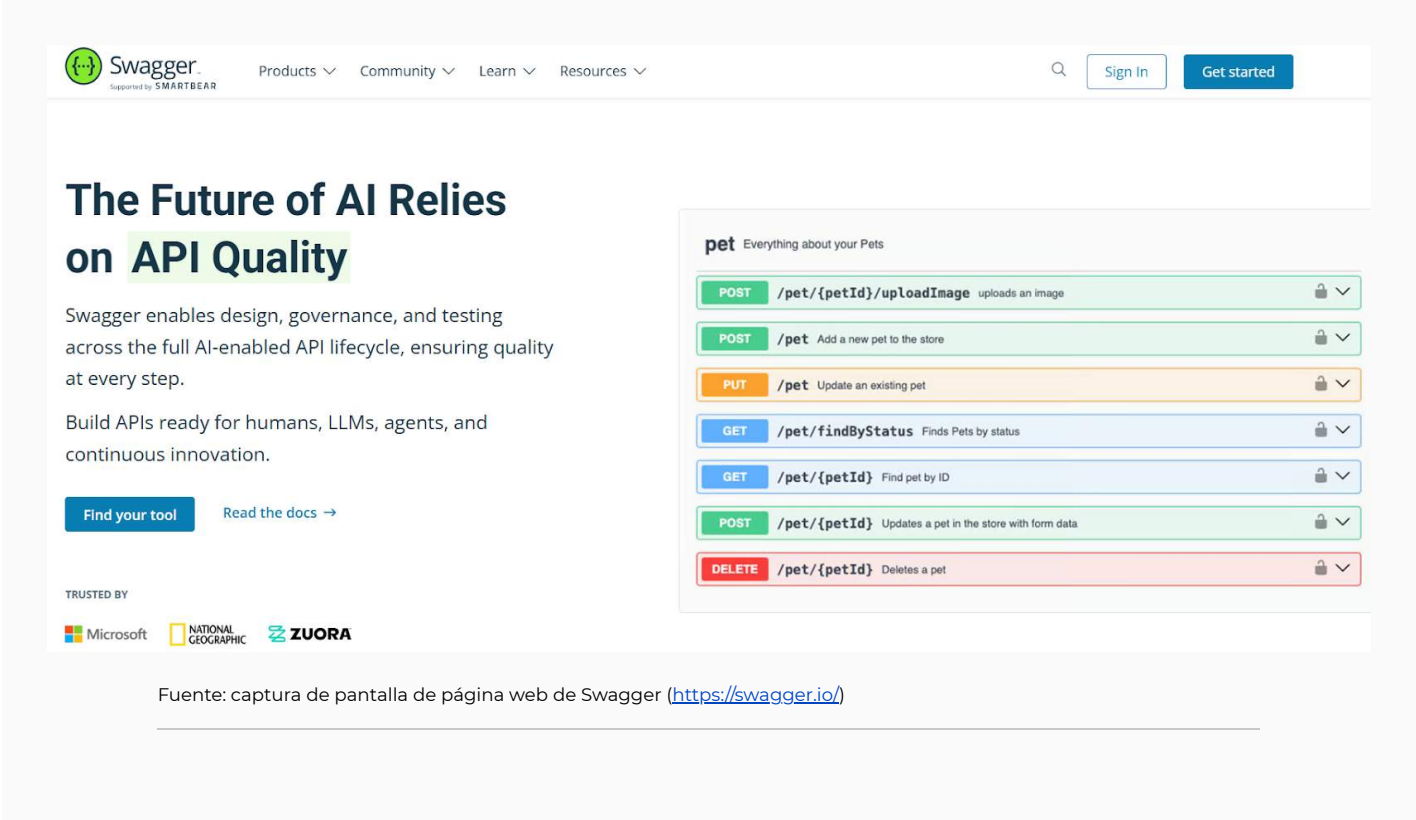
Módulo 2. Swagger

CÓRDOBA CREA FUTURO

1. Introducción

En programación, Swagger (actualmente conocido como la especificación OpenAPI) es un conjunto de herramientas de código abierto para diseñar, construir, documentar y consumir API *RESTful*. Permite a los desarrolladores comprender y utilizar servicios web sin necesidad de acceder al código fuente, genera documentación interactiva y estandarizada, y facilita la colaboración mediante la definición de las capacidades de una API en formatos como YAML o JSON.

Figura 1. Página principal de Swagger



Fuente: captura de pantalla de página web de Swagger (https://swagger.io/)

Con Swagger, se pueden definir los *endpoints* de una API —interfaz de programación de aplicaciones, es decir, un conjunto de reglas y protocolos que permiten a las aplicaciones

comunicarse entre sí a través de Internet (*application programming interface*)—, los parámetros que acepta, los tipos de datos que devuelve y los errores que puede generar.

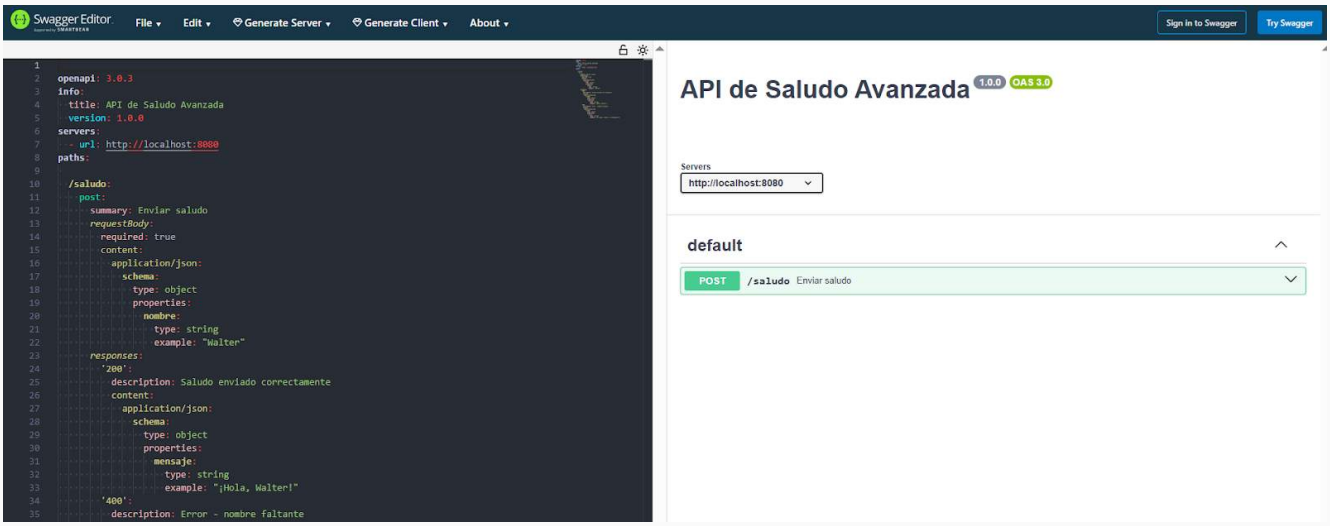
El nombre Swagger suele atribuirse a la idea de que la herramienta proporciona una forma segura y confiada de presentar y gestionar las API de una aplicación.

Componentes clave de Swagger (OpenAPI)

Swagger se compone de diversas herramientas y especificaciones que facilitan el diseño, la documentación y la prueba de API. A continuación, se describen sus principales componentes:

- **OpenAPI Specification (OAS)**. Es un documento —generalmente en formato openapi.yaml u openapi.json— que describe la API que se está desarrollando. Incluye información sobre *endpoints*, parámetros, cuerpos de solicitud y respuesta, códigos de estado, mecanismos de seguridad, entre otros aspectos. Se utiliza con un editor en el navegador que permite redactar y validar archivos de definición OpenAPI en formato YAML o JSON.

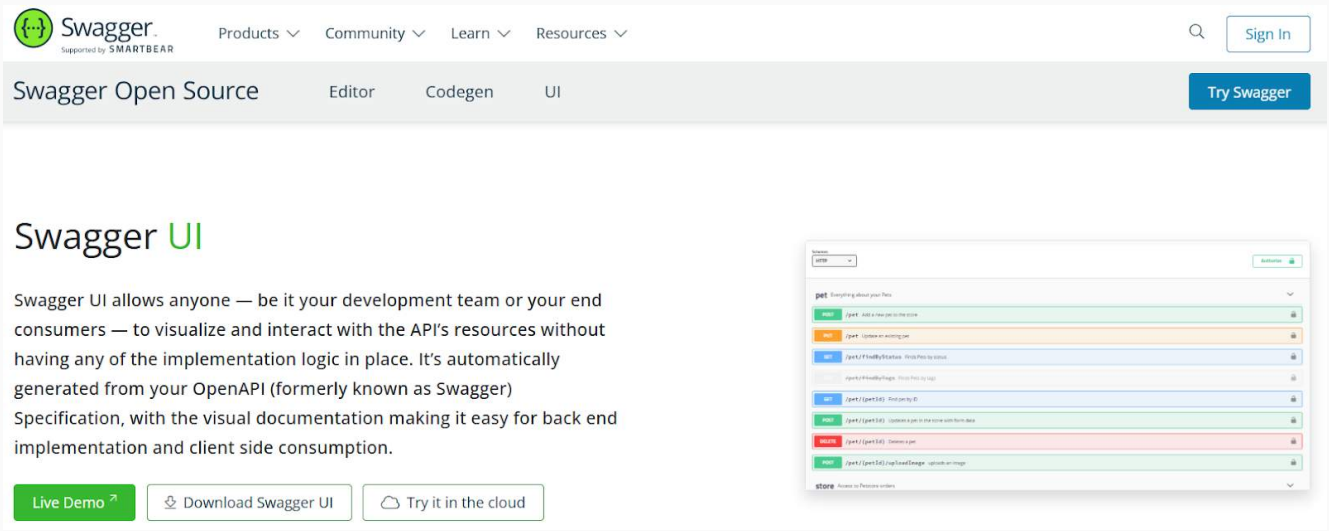
Figura 2. Editor Swagger



Fuente: captura de pantalla de página web de Swagger (https://swagger.io/)

- **Swagger UI**: es una interfaz web que interpreta la especificación OpenAPI y permite probar la API de forma interactiva mediante el botón «Try it out». A partir del documento, genera una documentación interactiva que facilita la exploración y evaluación de los *endpoints* definidos.

Figura 3. Swagger UI



Fuente: captura de pantalla de página web de Swagger (https://swagger.io/)

- **Swagger/OpenAPI *tooling***: incluye herramientas para automatizar distintos aspectos del desarrollo. Entre sus funcionalidades se encuentran la generación automática de documentación a partir de anotaciones o del entorno de ejecución, y la validación de solicitudes y respuestas conforme al contrato establecido en la especificación.
- **Codegen**: permite generar clientes en diferentes lenguajes (como Java, JavaScript o Python) y *stubs* de servidor a partir del documento OAS. De esta manera, se agiliza la implementación de las API y se asegura la coherencia con la especificación.

¿Para qué sirve?

Swagger ofrece múltiples beneficios en el desarrollo y gestión de API. Entre sus principales usos, se encuentran los siguientes:

- **Documentación automática**. Crea páginas de ayuda claras y actualizadas automáticamente para las API, lo que facilita su comprensión y mantenimiento.
- **Diseño primero (*design first*)**: permite definir la API antes de escribir el código, lo que mejora la colaboración entre equipos y asegura una planificación más precisa.
- **Pruebas interactivas**: los desarrolladores pueden probar métodos y parámetros en tiempo real a través de Swagger UI, lo que agiliza la validación y el ajuste de funcionalidades.
- **Generación de código**: acelera el desarrollo al generar código cliente y de servidor directamente desde la especificación.
- **Estandarización**: proporciona un formato universal (OpenAPI) para describir API, independiente del lenguaje de programación utilizado.

¿Cuál es la diferencia entre Swagger y rest API?

Swagger es una herramienta de documentación y un *framework* de generación de código para API, mientras que una API REST es un estilo arquitectónico utilizado para el diseño de servicios web.

Swagger resulta útil para gestionar el ciclo de vida de las API, ya que hace que el desarrollo y la integración de servicios web sean más eficientes y menos propensos a errores. Ofrece herramientas que abarcan desde el diseño hasta la generación de código y la documentación interactiva.

CONTINUAR

2. Stubs de servidores

Los stubs de servidores son simulaciones de un servidor real que se utilizan en pruebas de software. Funcionan como sustitutos funcionales que permiten aislar el código a probar, devolver respuestas predefinidas y manejar llamadas a funciones remotas sin depender del sistema original.

Gracias a estos *stubs*, los desarrolladores pueden probar un cliente o una aplicación de forma aislada, verificando la lógica sin la complejidad ni la necesidad de disponer de un servidor en funcionamiento. Esto resulta especialmente útil en sistemas distribuidos, donde es necesario simular servicios remotos.

Funciones principales

Los *stubs* de servidores ofrecen varias funcionalidades clave en entornos de prueba y desarrollo. Entre ellas, se destacan las siguientes:

- **Simulación**. Reemplazan funcionalmente a un servidor o servicio real, presentándose de manera idéntica ante el cliente que los consume.
- **Aislamiento**: permiten probar componentes de software de forma aislada, sin depender de servicios externos, bases de datos o sistemas en funcionamiento.
- **Respuestas controladas**: devuelven datos o respuestas estáticas predefinidas, independientemente de la entrada recibida, lo que facilita la verificación del comportamiento esperado.
- **Simplificación**: ocultan la complejidad de la comunicación de red en sistemas distribuidos, lo que hace que el desarrollo sea más sencillo.

Ejemplos de uso

Los *stubs* de servidores pueden aplicarse en distintas etapas del desarrollo y la prueba de software. A continuación, se presentan algunos ejemplos:

- **Pruebas unitarias**. Un *stub* de servidor puede simular una API externa, devolviendo un archivo JSON específico. Esto permite que una aplicación cliente pruebe su lógica de manejo de datos sin realizar una llamada real a Internet.
- **Desarrollo en paralelo**: un equipo puede trabajar en el *frontend* utilizando *stubs* para API que aún no están disponibles, lo que permite continuar con el desarrollo sin interrupciones.

En resumen, un *stub* de servidor es un componente temporal y simplificado que imita el comportamiento de un servicio real con fines de desarrollo y pruebas, permitiendo un control total sobre las respuestas y el entorno de ejecución.

OpenAPI Specification (OAS)

En el contexto de API y *stubs* de servidores, OAS hace referencia a la *OpenAPI Specification*, un formato estándar (JSON/YAML) que describe API *RESTful*. Este formato permite generar automáticamente *stubs* de servidor (esqueletos de código) y clientes, lo que facilita la creación de interfaces, documentación y pruebas sin necesidad de acceder al código fuente. Esto se detalla en documentos como los provistos por Swagger.

¿Qué es OAS y cómo se relaciona con los stubs?

La OpenAPI Specification (OAS) es un lenguaje agnóstico utilizado para describir la interfaz de una API HTTP. Define sus *endpoints*, métodos (como GET o POST), parámetros, respuestas y esquemas de datos. El documento OAS es el archivo (en formato JSON o YAML) que contiene esta descripción detallada de la API. A partir de este documento, se pueden generar *stubs* de servidor, es decir, plantillas de código que incluyen la estructura básica de las funciones del servidor. Estas plantillas permiten a los desarrolladores implementar la lógica de negocio sobre una base ya definida, lo que acelera el desarrollo y garantiza la consistencia con la especificación.

De esta manera, el documento OAS actúa como el plano de la API, y las herramientas de generación de código utilizan ese plano para crear los *stubs* de servidor. Estos *stubs* sirven como punto de partida para construir el servidor real, asegurando que se respete el diseño definido en la especificación.

Dependencias Maven/Gradle en Java

El uso de dependencias Maven o Gradle en Java permite automatizar la gestión de librerías externas, eliminando la necesidad de descargar archivos JAR manualmente, resolviendo conflictos de versiones, estandarizando la estructura del proyecto y facilitando tanto la compilación como las pruebas en distintos entornos. Además, mejora la colaboración entre desarrolladores al centralizar la configuración de dependencias en un archivo declarativo (pom.xml o build.gradle).

Ambas son herramientas de construcción (*build tools*) que simplifican considerablemente el ciclo de vida de un proyecto Java, desde la descarga de dependencias hasta el empaquetado final.

¿Maven o Gradle?

La elección entre Maven y Gradle depende del tipo de proyecto y las necesidades del equipo de desarrollo.

- **Maven**. Es una buena opción para proyectos que siguen convenciones establecidas y requieren una configuración mínima. Se caracteriza por su robustez, una comunidad amplia y una gran variedad de *plugins* maduros, lo que lo vuelve especialmente adecuado para desarrollos medianos y con estructuras estándar. Por ejemplo, al trabajar con Spring Boot, se pueden declarar las dependencias en el archivo *pom.xml* de la siguiente manera:

```
<!-- SpringFox (Swagger 2) - Para Spring Boot 2.x -->

<dependency>

  <groupId>io.springfox</groupId>

  <artifactId>springfox-boot-starter</artifactId>

  <version>3.0.0</version>

</dependency>
```

```
<!-- O SpringDoc OpenAPI (recomendado) - Para Spring Boot 3.x -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

- **Gradle.** Brinda mayor flexibilidad y rendimiento, especialmente en proyectos grandes, gracias a su compilación incremental y sistema de caché. Utiliza un lenguaje específico de dominio (DSL), como Groovy o Kotlin, que permite definir lógica de construcción personalizada de forma más expresiva que el XML utilizado por Maven. En este caso, las dependencias se definen en el archivo *build.gradle*, como se muestra a continuación:

```
// SpringDoc (OpenAPI 3)
implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.5.0'
```

Beneficios clave de usar gestores de dependencias

El uso de gestores de dependencias como Maven o Gradle aporta múltiples ventajas en el desarrollo de proyectos Java. Entre los principales beneficios, se encuentran los siguientes:

- **Automatización de descargas**. Descargan automáticamente las librerías (JAR) y sus dependencias transitivas desde repositorios centrales, como Maven Central.

- **Gestión de versiones**: resuelven de forma automática qué versiones de librerías utilizar, lo que evita conflictos e incompatibilidades que requerirían intervención manual.

- **Estandarización**: imponen una estructura de proyecto común, lo que hace que el código sea más predecible y fácil de comprender para otros desarrolladores.

- **Independencia del IDE**: permiten compilar y construir el proyecto de manera uniforme en distintos sistemas operativos o entornos de desarrollo, como Eclipse, IntelliJ o VS Code.

- **Ciclo de vida definido**: gestionan fases del proyecto como la compilación, prueba, empaquetado (JAR/WAR) y despliegue, de forma consistente y automatizada.

Build tools

Las *build tools* en Java son programas que automatizan tareas repetitivas dentro del proceso de desarrollo de software. Entre sus funciones principales se encuentran la compilación del código fuente a *bytecode*, la gestión de dependencias externas (librerías), la ejecución de pruebas unitarias y el empaquetado de la aplicación en archivos ejecutables (JAR, WAR) o desplegables.

Maven y Gradle son las herramientas más utilizadas en este ámbito, ya que facilitan la gestión de proyectos grandes y complejos, asegurando consistencia y eficiencia a lo largo de todo el ciclo de vida del *software*.

¿Qué hacen las build tools?

Las *build tools* automatizan diversas tareas dentro del ciclo de vida del desarrollo de software en Java. Entre sus principales funciones, se encuentran las siguientes:

- **Compilación**. Convierten el código fuente Java (.java) en *bytecode* (.class), listo para ser ejecutado por la máquina virtual de Java.
- **Gestión de dependencias**: descargan y administran automáticamente las librerías de terceros que el proyecto necesita para funcionar correctamente.
- **Pruebas**: ejecutan pruebas unitarias de forma automática para verificar la calidad y estabilidad del código.
- **Empaquetado**: generan artefactos como archivos JAR (bibliotecas) o WAR (aplicaciones web), listos para ser desplegados.
- **Despliegue**: permiten implementar la aplicación en servidores o publicarla en repositorios de artefactos.
- **Integración continua (CI)**: desempeñan un rol clave en procesos de integración y entrega continua (CI/CD), construyendo y probando el código automáticamente en cada cambio.

¿Cómo usar Swagger?

A continuación —con base en la documentación oficial de Swagger (2025)— se describen los pasos para el uso de esta herramienta:

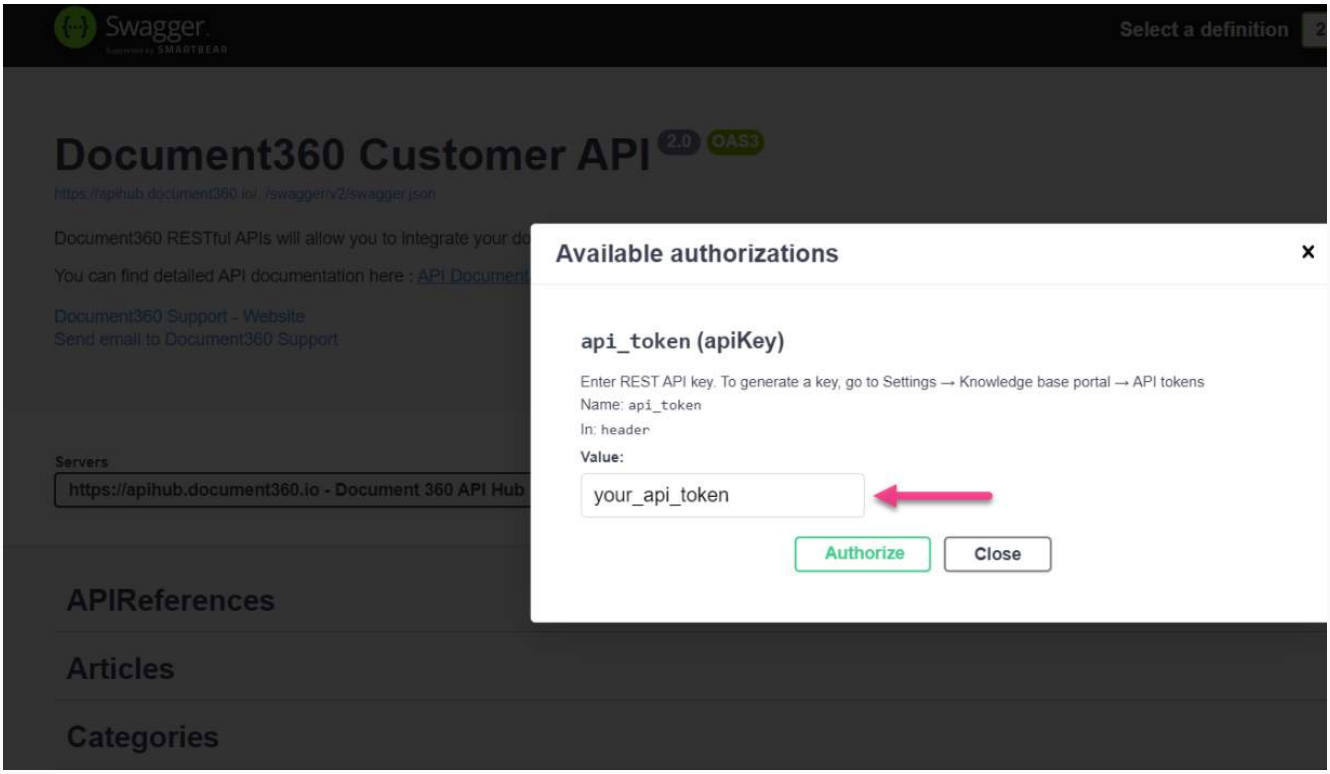
1. Abrir el sitio de documentación de Swagger: https://apihub.document360.io/

2. Para autenticar la documentación, hacer clic en «Autorizar». Se abrirá el panel «Autorizaciones disponibles».
3. Introducir el token de API en el campo «Valor».

Nota: se puede seleccionar un token existente o generar uno nuevo, según sea necesario. Para generar un nuevo token, seguir estos pasos:

- Ir a «Configuración» en la barra de navegación izquierda del portal de la base de conocimientos.
- En el panel de navegación izquierdo, acceder a «Portal de la base de conocimientos» > «Tokens de API».
- Escribir un nombre para el token en el campo «Escriba el nombre del token».
- Seleccionar los métodos HTTP necesarios. En «Método(s) permitido(s)» se ofrecen cuatro opciones: GET, PUT, POST y DELETE.
- El *token* generado solo podrá utilizarse con los métodos seleccionados. No será válido para otros métodos.
- Hacer clic en «Copiar» para copiar el *token* de API.
- Hacer clic en el botón «Autorizar».
- Una vez completado el proceso, hacer clic en el botón «Cerrar». La autorización se habrá realizado correctamente.

Figura 4. Generación de *token* en Swagger



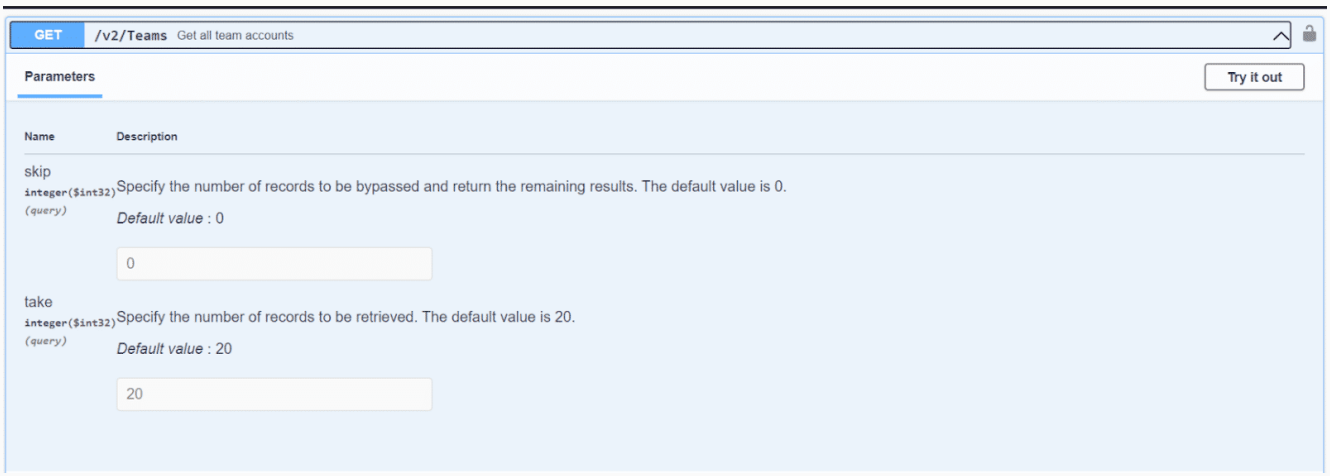
Fuente: Swagger, 2025, https://goo.su/gHnCj

Nota: si se desea cambiar a un token diferente, hacer clic en «Autorizar», luego en «Cerrar sesión», ingresar el nuevo token de API, hacer clic nuevamente en «Autorizar» y, finalmente, en «Cerrar».

Uso del método GET

El método GET se utiliza para recuperar información. A continuación, se presenta un ejemplo de acción GET para consultar todos los registros asociados a la cuenta del equipo.

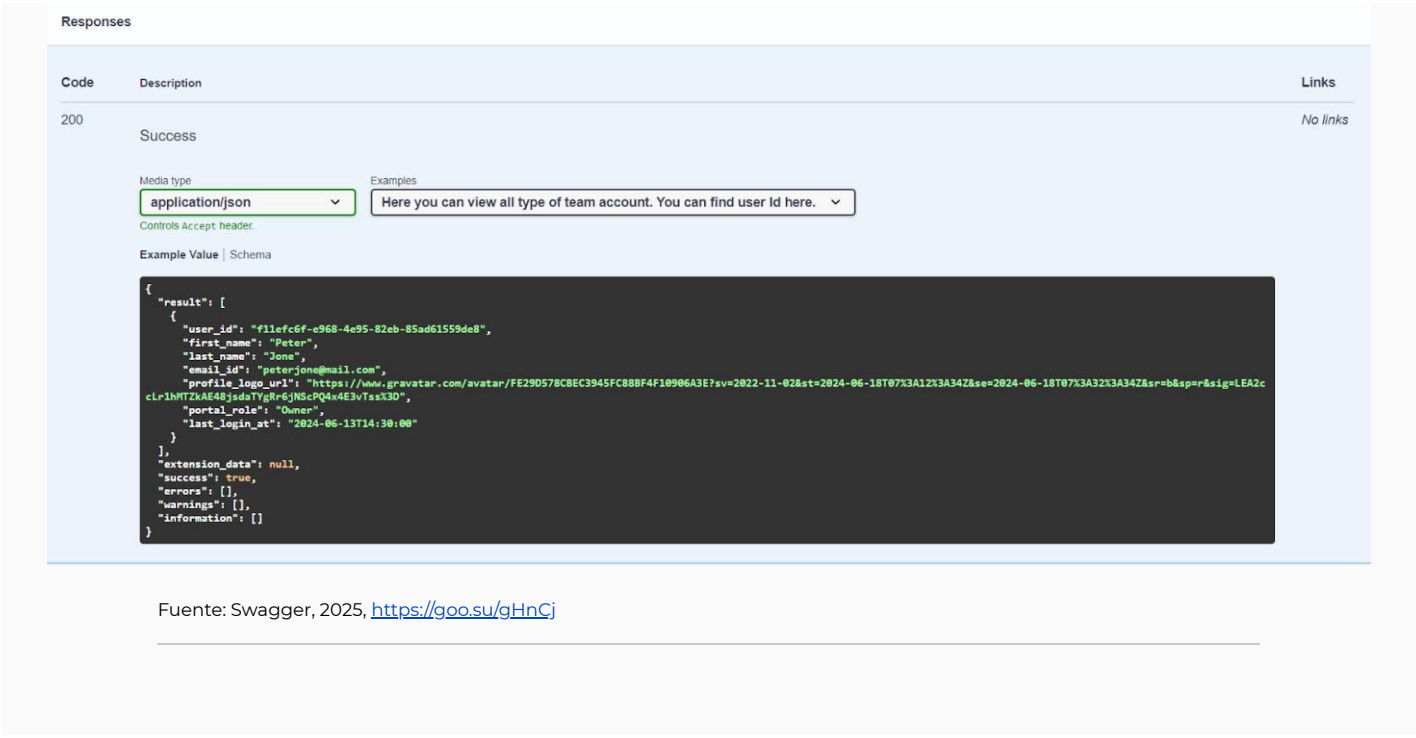
Figura 5. Ejemplo de uso del método GET para consultar todas las cuentas de equipo mediante el *endpoint* /v2/Teams



Fuente: Swagger, 2025, https://goo.su/gHnCj

Para ejecutar la consulta, se deben completar los campos correspondientes y hacer clic en el botón «Ejecutar». Una vez realizada la solicitud, la información relacionada con la cuenta del equipo estará disponible en la sección «Respuestas».

Figura 6. Respuesta del *endpoint* GET /v2/Teams con datos de una cuenta de equipo en formato JSON

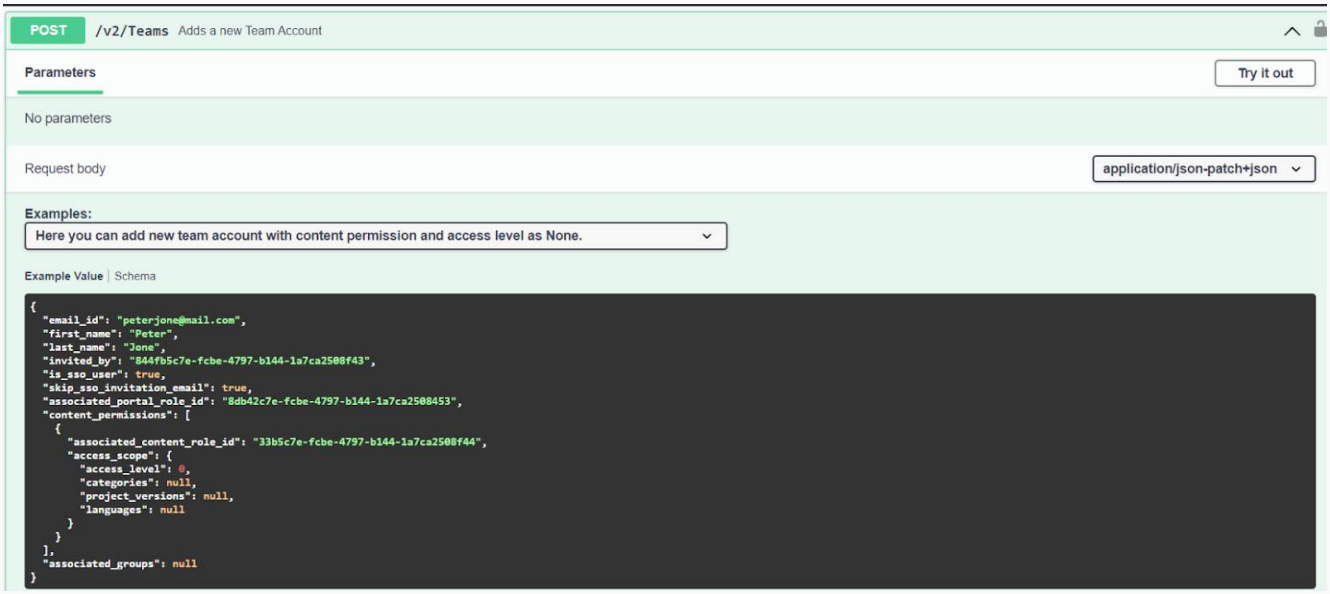


Fuente: Swagger, 2025, https://goo.su/gHnCj

Uso del método POST

El método POST se utiliza para agregar un nuevo elemento. A continuación, se presenta un ejemplo de acción POST para añadir una cuenta de equipo a un proyecto.

Figura 7. Solicitud POST al *endpoint* `/v2/Teams` para agregar una nueva cuenta de equipo



Fuente: Swagger, 2025, https://goo.su/gHnCj

Se debe completar la información de la cuenta del equipo, incluyendo los campos «email_id», «first_name» y «last_name», para agregar un usuario al proyecto.

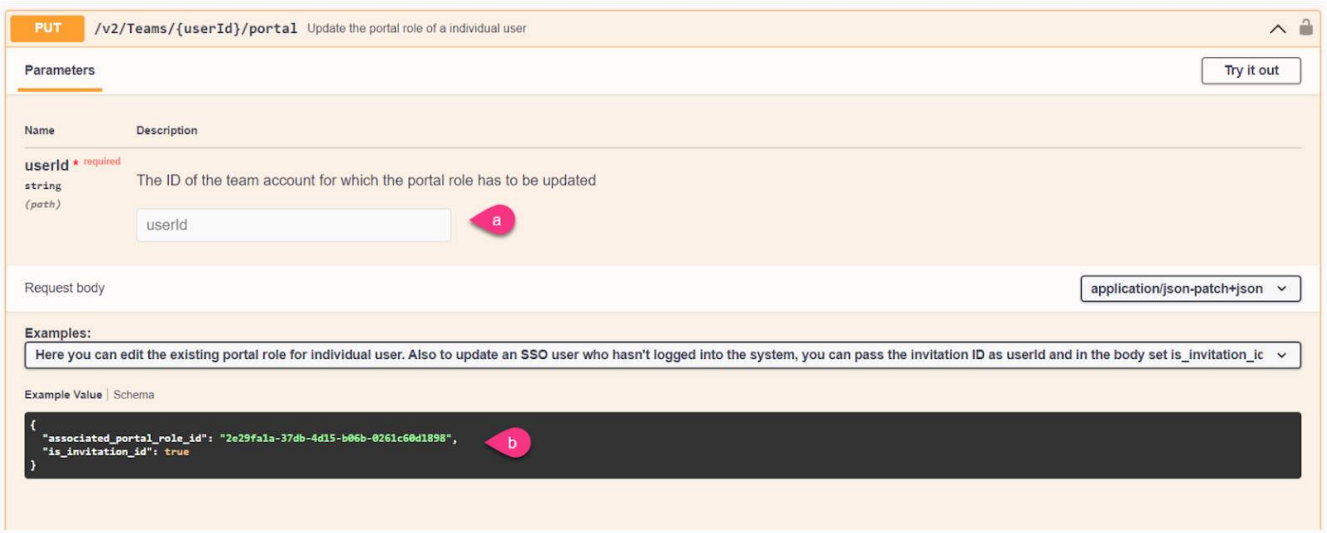
Una vez ingresados los datos, hacer clic en el botón «Ejecutar».

La cuenta de equipo se añadirá correctamente al proyecto si la respuesta devuelve un estado 200 y el valor de «success» es *true*.

Uso del método PUT

El método PUT se utiliza para actualizar un elemento existente. A continuación, se presenta un ejemplo de acción PUT para modificar el rol de un usuario en el proyecto.

Figura 8. Solicitud PUT al *endpoint* /v2/Teams/{userId}/portal para actualizar el rol de un usuario en el portal



Fuente: Swagger, 2025, https://goo.su/gHnCj

Para realizar la actualización, se debe ingresar el valor de «userId» en el campo correspondiente. Luego, se actualiza el rol de usuario utilizando el identificador numérico correspondiente a cada tipo de rol. Los valores asignados a cada rol son los siguientes:

- Administrador
- Editor
- Redactor de borradores
- Lector
- Propietario

Una vez completados los datos, hacer clic en el botón «Ejecutar». La actualización se considerará exitosa si la respuesta devuelve un estado 200 y el valor de «success» es *true*.

Uso del método DELETE

El método DELETE se utiliza para eliminar un elemento. A continuación, se presenta un ejemplo de acción DELETE para eliminar un usuario identificado por su `userId`.

Figura 9. Solicitud DELETE al *endpoint* `/v2/Teams/{userId}` para eliminar una cuenta de equipo específica

Fuente: Swagger, 2025, https://goo.su/gHnCj

Para ejecutar la acción, se debe ingresar el userId en el campo correspondiente. Una vez hecho esto, hacer clic en el botón «Ejecutar». La eliminación se considerará exitosa si la respuesta devuelve un estado 200 y el valor de success es true.

Spring Boot

Spring Boot es una extensión del *framework* Spring que simplifica notablemente el desarrollo de aplicaciones web y microservicios listos para producción en Java. Elimina configuraciones complejas y permite a los desarrolladores centrarse en la lógica de negocio mediante mecanismos como la autoconfiguración, el uso de dependencias predefinidas y la capacidad de crear aplicaciones independientes y ejecutables.

En esencia, Spring Boot hace que el desarrollo en Java sea más rápido y sencillo. Entre sus características principales, se destacan las siguientes:

- **Autoconfiguración (*auto-configuration*)**. Configura automáticamente la aplicación en función de las librerías incluidas en el proyecto. Esto reduce la necesidad de declarar manualmente *beans* y configuraciones en XML o Java.
- **Dependencias *starters***: ofrece paquetes de dependencias preconfiguradas (como `spring-boot-starter-web`), que incluyen todo lo necesario para comenzar rápidamente con una funcionalidad específica, como el desarrollo web.
- **Aplicaciones independientes (*standalone*)**: permite crear aplicaciones que incluyen un servidor web embebido (como Tomcat), lo que evita tener que desplegar un archivo WAR en un servidor externo.
- **Convención sobre configuración (*convention over configuration*)**: adopta un enfoque basado en convenciones predefinidas (como nombres de archivos y patrones de código), lo que reduce la cantidad de decisiones de configuración que debe tomar el desarrollador.

En comparación, **Spring Framework** es la base: un marco completo para construir aplicaciones empresariales en Java. **Spring Boot**, en cambio, actúa como una capa superior que simplifica el uso de Spring, facilitando el arranque y la configuración rápida

de proyectos sin la complejidad inicial que caracteriza al enfoque tradicional. Por estas razones, Spring Boot resulta especialmente adecuado en situaciones como el desarrollo ágil de API *RESTful*, la creación de microservicios y la implementación de aplicaciones listas para producción en la nube o en entornos basados en contenedores.

Se recomienda consultar la siguiente publicación, donde se explica esta herramienta en mayor detalle:

> **Fuente: Álvarez Caules, C.** (2024). Spring Boot ¿Qué es y cómo funciona? *Arquitectura Java*. https://www.arquitecturajava.com/spring-boot-que-es/

¿Cómo se usa en Java (Spring Boot)?

En proyectos modernos desarrollados con Spring Boot, la forma más común de documentar API es utilizando **SpringDoc OpenAPI**, una herramienta que reemplaza al antiguo *springfox-swagger*. Esta biblioteca permite generar automáticamente el documento OpenAPI y mostrar una interfaz interactiva a través de Swagger UI, sin necesidad de realizar configuraciones complejas.

1. **Agregar dependencias**

Para usar SpringDoc OpenAPI en un proyecto con Maven, se debe incluir la siguiente dependencia en el archivo `pom.xml`:

```
<dependencies>

<!-- Genera OpenAPI y expone Swagger UI -->

<dependency>

  <groupId>org.springdoc</groupId>
```

```
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

  <version>2.6.0</version> <!-- usa la última estable -->

</dependency>

</dependencies>
```

2. **Crear un controlador de ejemplo**

A continuación, se muestra un ejemplo de un controlador en Java que gestiona cursos. El controlador utiliza anotaciones de la biblioteca `io.swagger.v3.oas.annotations` para documentar automáticamente cada *endpoint*:

```
import org.springframework.http.ResponseEntity;

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

import io.swagger.v3.oas.annotations.Operation;

import io.swagger.v3.oas.annotations.tags.Tag;

import io.swagger.v3.oas.annotations.media.Schema;

import java.util.List;

@Tag(name = "Cursos", description = "Gestión de cursos")

@RestController

@RequestMapping("/api/cursos")

public class CursoController {
```

```
    @Operation(summary = "Listar cursos", description = "Retorna todos los cursos disponibles")
  @GetMapping
  public ResponseEntity<List<CursoDTO>> listar() {
    // lógica...
    return ResponseEntity.ok(List.of());
  }
  @Operation(summary = "Crear curso")
  @PostMapping
  public ResponseEntity<CursoDTO> crear(@RequestBody CursoDTO dto) {
    // lógica...
    return ResponseEntity.ok(dto);
  }
  public static class CursoDTO {
    @Schema(example = "Java Avanzado")
    public String nombre;
    @Schema(example = "Arquitectura en capas, excepciones, APIs, persistencia")
    public String descripcion;
```

```
  }

}
```

3. **Acceso a la documentación**

Una vez configurado el proyecto, se pueden acceder a las siguientes rutas predeterminadas:

- Documento OpenAPI en formato JSON: `GET /v3/api-docs`
- Interfaz Swagger UI: `GET /swagger-ui.html` o `/swagger-ui/index.html`

4. **Anotaciones más comunes**

Estas son algunas de las anotaciones más utilizadas en SpringDoc OpenAPI:

- `@Operation`: describe un *endpoint*, incluyendo resumen, descripción y etiquetas.
- `@Schema`: documenta las propiedades de un modelo de datos (como atributos de una clase).
- `@Parameter`: documenta los parámetros que se envían en la solicitud.
- `@ApiResponse` y `@ApiResponses`: definen las posibles respuestas del *endpoint*.
- `@SecurityRequirement`: especifica los requisitos de seguridad (como autenticación con JWT).

Flujo típico de trabajo

El flujo de trabajo con SpringDoc OpenAPI suele seguir estos pasos:

1. Escribir la API (controladores y DTO).
2. Añadir anotaciones (opcional, pero recomendable para enriquecer la documentación).

3. springdoc-openapi genera automáticamente el documento en /v3/api-docs.
4. Swagger UI muestra la documentación y permite probar los *endpoints*.
5. Si se necesitan SDK cliente, se puede usar OpenAPI Generator para generar código a partir del archivo JSON o YAML.

Consejos y buenas prácticas

Al utilizar SpringDoc OpenAPI, se recomienda seguir estas buenas prácticas para mejorar la calidad de la documentación:

1. Mantener nombres y descripciones claras en los atributos summary y description de la anotación @Operation.

- Definir los esquemas de datos utilizando @Schema, incluyendo ejemplos siempre que sea posible.
- Documentar los errores comunes (400, 401, 404, 422, 500) mediante la anotación @ApiResponses.
- Incorporar medidas de seguridad, como autenticación con Bearer JWT, en la configuración global de OpenAPI.
- Versionar la API (por ejemplo, /api/v1/...) y reflejar la versión en el título del documento generado.

Nota: el *token* tipo Bearer se basa en un estándar (RFC 7519) conocido como JWT (*JSON Web Token*), utilizado comúnmente en API para la autenticación sin estado. El término «Bearer» indica que quien posea el token puede acceder a recursos protegidos, sin necesidad de

mantener una sesión en el servidor. Un JWT está compuesto por tres partes:

- ***Header*. Contiene el algoritmo de firma utilizado.**
- ***Payload*: incluye los datos del usuario o las reclamaciones (*claims*).**
- **Firma digital: garantiza que el contenido no ha sido alterado.**

En Java, este tipo de autenticación se gestiona habitualmente mediante librerías como Spring Security, lo que permite proteger los endpoints de forma escalable.

Ejemplo Java

El siguiente ejemplo, elaborado con ayuda de *vide coding* mediante Deepseek, muestra cómo documentar API REST en proyectos Java utilizando Swagger, actualmente implementado mediante la especificación OpenAPI. Este enfoque permite generar documentación interactiva, acceder a los *endpoints* desde el navegador y mantener una estructura clara y centralizada.

1. **Dependencias Maven/Gradle**

Para comenzar, es necesario agregar las dependencias en el proyecto, según se use Maven o Gradle:

- **Maven:**

<!-- SpringFox (Swagger 2) - Para Spring Boot 2.x -->

```
<dependency>
  <groupId>io.springfox</groupId>
  <artifactId>springfox-boot-starter</artifactId>
  <version>3.0.0</version>
</dependency>

<!-- O SpringDoc OpenAPI (recomendado) - Para Spring Boot 3.x -->
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

Gradle:

```
// SpringDoc (OpenAPI 3)
implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.5.0'
```

2. **Configuración básica**

Se define una clase de configuración para personalizar el documento generado:

```
import io.swagger.v3.oas.models.OpenAPI;

import io.swagger.v3.oas.models.info.Info;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;



@Configuration

public class OpenApiConfig {



  @Bean

  public OpenAPI customOpenAPI() {

    return new OpenAPI()

      .info(new Info()

        .title("API de Ejemplo")

        .version("1.0")

        .description("Documentación de la API")
```

```
            .termsOfService("http://swagger.io/terms/")
            .license(new License()
               .name("Apache 2.0")
               .url("http://springdoc.org")));
  }
}
```

3. **Anotaciones en los controladores**

El siguiente ejemplo muestra un controlador REST documentado con anotaciones OpenAPI:

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.Parameter;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import io.swagger.v3.oas.annotations.tags.Tag;

@RestController
@RequestMapping("/api/usuarios")
@Tag(name = "Usuarios", description = "Operaciones CRUD para usuarios")
```

```
public class UsuarioController {



  @GetMapping("/{id}")

  @Operation(

     summary = "Obtener usuario por ID",

     description = "Retorna un usuario específico basado en su ID"

  )

  @ApiResponse(

     responseCode = "200",

     description = "Usuario encontrado"

  )

  @ApiResponse(

     responseCode = "404",

     description = "Usuario no encontrado"

  )

  public Usuario getUsuario(

     @Parameter(
```

```
        description = "ID del usuario",
        required = true,
        example = "123"
    )
    @PathVariable Long id
) {
    // lógica del método
}

@PostMapping
@Operation(summary = "Crear un nuevo usuario")
public Usuario crearUsuario(
    @io.swagger.v3.oas.annotations.parameters.RequestBody(
        description = "Objeto usuario a crear",
        required = true
    )
    @RequestBody Usuario usuario
```

```
    ) {
        // lógica del método
    }
}
```

4. **Anotaciones en modelos (DTO)**

El modelo `Usuario` puede documentarse así:

```
import io.swagger.v3.oas.annotations.media.Schema;

@Schema(description = "Entidad que representa un usuario")
public class Usuario {

    @Schema(
        description = "ID único del usuario",
        example = "123",
        accessMode = Schema.AccessMode.READ_ONLY
```

```
)
private Long id;

@Schema(
    description = "Nombre completo del usuario",
    example = "Juan Pérez",
    required = true
)
@NotBlank
private String nombre;

@Schema(
    description = "Correo electrónico",
    example = "juan@email.com",
    required = true
)
@Email
```

```
    private String email;


    // getters y setters
}
```

5. Configuración en application.properties o application.yml

```
# SpringDoc OpenAPI config
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui.html
springdoc.swagger-ui.enabled=true

# Personalizar UI
springdoc.swagger-ui.operationsSorter=method
springdoc.swagger-ui.tagsSorter=alpha
springdoc.swagger-ui.display-request-duration=true
```

6. **Acceso a la documentación**

Una vez ejecutado el proyecto, se puede acceder a las siguientes rutas:

- Swagger UI: http://localhost:8080/swagger-ui.html
- Documento OpenAPI en JSON: http://localhost:8080/v3/api-docs
- Documento personalizado (si se configuró): http://localhost:8080/api-docs

7. **Configuración de seguridad (si se usa Spring Security)**

Para permitir acceso a Swagger UI sin autenticación, se puede definir una configuración como esta:

```
@Configuration
public class SecurityConfig {


  @Bean
  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
      .authorizeHttpRequests(auth -> auth
        .requestMatchers(
          "/swagger-ui/**",
```

```
                "/v3/api-docs/**",
                "/api-docs/**"
            ).permitAll()
            .anyRequest().authenticated()
        );

    return http.build();
  }
}
```

8. **Agrupar API por áreas (configuración avanzada)**

Para separar la documentación por secciones, se pueden definir grupos de *endpoints* con `GroupedOpenApi`.

```
@Configuration
public class OpenApiGroupConfig {

  @Bean
```

```
public GroupedOpenApi publicApi() {
    return GroupedOpenApi.builder()
        .group("public")
        .pathsToMatch("/api/public/**")
        .build();
}

@Bean
public GroupedOpenApi adminApi() {
    return GroupedOpenApi.builder()
        .group("admin")
        .pathsToMatch("/api/admin/**")
        .addOpenApiCustomizer(openApi ->
            openApi.info(new Info().title("API Admin"))
        )
        .build();
}
```

```
}
```

9. **Controlador de ejemplo completo**

A continuación, se muestra un controlador completo con operaciones básicas de una API REST documentadas mediante anotaciones OpenAPI.

```
@RestController

@RequestMapping("/api/productos")

@Tag(name = "Productos", description = "Gestión de productos")

public class ProductoController {


  @Autowired

  private ProductoService productoService;



  @GetMapping

  @Operation(summary = "Listar todos los productos")

  @ApiResponse(responseCode = "200", description = "Lista de productos")

  public List<Producto> listarProductos() {

    return productoService.obtenerTodos();
```

```
}

@GetMapping("/{id}")
@Operation(summary = "Buscar producto por ID")
@ApiResponses({
    @ApiResponse(responseCode = "200", description = "Producto encontrado"),
    @ApiResponse(responseCode = "404", description = "Producto no encontrado")
})
public Producto obtenerProducto(
    @Parameter(description = "ID del producto", required = true)
    @PathVariable Long id
) {
    return productoService.obtenerPorId(id);
}

@PostMapping
@Operation(summary = "Crear nuevo producto")
```

```
    @ApiResponse(responseCode = "201", description = "Producto creado")
    @ResponseStatus(HttpStatus.CREATED)
    public Producto crearProducto(
        @RequestBody
        @Valid
        @io.swagger.v3.oas.annotations.parameters.RequestBody(
            description = "Datos del nuevo producto",
            required = true
        )
        Producto producto
    ) {
        return productoService.crear(producto);
    }
}
```

CONTINUAR

Referencias

Álvarez Caules, C. (2024). Spring Boot ¿Qué es y cómo funciona? *Arquitectura Java*. https://www.arquitecturajava.com/spring-boot-que-es/

Swagger, (2025). *¿Cómo usar Swagger?* https://docs.document360.com/docs/es/how-to-use-swagger

Referencias bibliográficas de consulta

Chakray, (2020). *Swagger y Swagger UI: ¿Qué es y por qué es imprescindible para las APIS?* https://www.chakray.com/es/swagger-y-swagger-ui-por-que-es-imprescindible-para-tus-apis/

Deitel, H. y Deitel, P. (2013). *Cómo programar en Java*. Pearson

IBM Knowledge Center, (2023). *Swagger*. https://www.ibm.com/support/knowledgecenter/SSMKHH_10.0.0/com.ibm.etools.mft.doc/bi12018_.htm .

IBM Knowledge Center, (2025). *Swagger*. https://www.ibm.com/docs/es/app-connect/11.0.0?topic=apis-swagger

Joyanes Aguilar, L. (2014). *Fundamentos de programación*. McGraw-Hill

Moreno, P. J. (2015). *Programación orientada a objetos*. RA-MA Editorial

Swagger UI, (s.f.). *Swagger-api. swagger-ui*. https://github.com/swagger-api/swagger-ui

CONTINUAR