**HERRAMIENTAS PARA PROBAR UNA API**

| **Información general** | |
| --- | --- |
| **Duración estimada en minutos:** | 120 |
| **Docente:** | Carlos Andrés Florez Villarraga |
| **Guía no.** | 04.1 |

| **Información de la Guía** |
| --- |

**OBJETIVO**

Aprender a probar una API REST hecha con Spring Boot.

**CONCEPTOS BÁSICOS**

Bases de datos NoSQL MongoDB, Java, archivos JSON.

**CONTEXTUALIZACIÓN TEÓRICA**

Cada funcionalidad de un proyecto debe ser testeada para verificar su correcto funcionamiento y garantizar que cumpla con los requisitos establecidos. En el caso de una API REST, existen múltiples formas de realizar pruebas, y algunas herramientas facilitan significativamente este proceso.

Una de las herramientas más utilizadas es Postman (<https://www.postman.com/>), que permite enviar peticiones HTTP a una API sin necesidad de desarrollar un cliente, lo que agiliza la validación de los endpoints. Postman admite distintos métodos HTTP (GET, POST, PUT, DELETE, entre otros) y permite configurar encabezados, parámetros y cuerpos de solicitud, facilitando así pruebas exhaustivas bajo diferentes escenarios.

Otra opción muy útil es Swagger (<https://swagger.io/>), que no solo permite probar la API de manera interactiva a través de una interfaz gráfica, sino que también facilita la documentación de los servicios. Con Swagger UI, los desarrolladores pueden visualizar y ejecutar peticiones directamente desde un navegador, lo que simplifica el proceso de prueba y comprensión de los endpoints disponibles.

Además, para realizar pruebas automatizadas dentro del código, es posible utilizar JUnit, un framework de pruebas unitarias para Java. Con JUnit, se pueden escribir pruebas automatizadas para los diferentes componentes de la API, garantizando que las funcionalidades se mantengan intactas a lo largo del desarrollo y que los cambios no introduzcan errores inesperados. JUnit puede integrarse con otras herramientas como Mockito para realizar pruebas de componentes individuales de manera aislada.

**PRECAUCIONES Y RECOMENDACIONES**

Recuerde verificar que tiene instalado el JDK de Java (preferiblemente la versión 21). Asegúrese de que tiene el servidor de MongoDB (mongod) en ejecución.

**ARTEFACTOS**

Se requiere tener instalado IntelliJ IDEA en su versión Ultimate, así como MongoDB y el controlador de versiones GIT.

**EVALUACIÓN O RESULTADO**

Se espera que el estudiante aprenda a probar una API REST creada con Spring Boot.

| **Herramientas para probar la API (Swagger)** |
| --- |

1. Swagger es una herramienta de código abierto que se utiliza para documentar y probar APIs RESTful. Es un conjunto de especificaciones que se pueden usar para describir la estructura y el comportamiento de una API. Con Swagger, se puede generar documentación de API interactiva y fácil de leer.

Para hacer pruebas con Swagger, agregue la siguiente dependencia en el archivo build.gradle.kts del proyecto y recuerde actualizar gradle para que se descargue la dependencia.

| *implementation*("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.8.0") |
| --- |

1. Si sabemos que hay endpoints (o rutas) de un contraldor cuyo acceso está protegido por algún mecanismo de autenticación, es necesario crear una clase con el nombre SwaggerConfig en el paquete co.edu.uniquindio.proyecto.config. Y agregamos la siguiente configuración:

| package co.edu.uniquindio.proyecto.config;  import io.swagger.v3.oas.annotations.enums.SecuritySchemeIn;  import io.swagger.v3.oas.annotations.enums.SecuritySchemeType;  import io.swagger.v3.oas.annotations.security.SecurityScheme;  import org.springframework.context.annotation.Configuration;  @Configuration  @SecurityScheme(  name = "bearerAuth",  scheme = "bearer",  bearerFormat = "JWT",  type = SecuritySchemeType.*HTTP*,  in = SecuritySchemeIn.*HEADER*  )  public class SwaggerConfig {  } |
| --- |

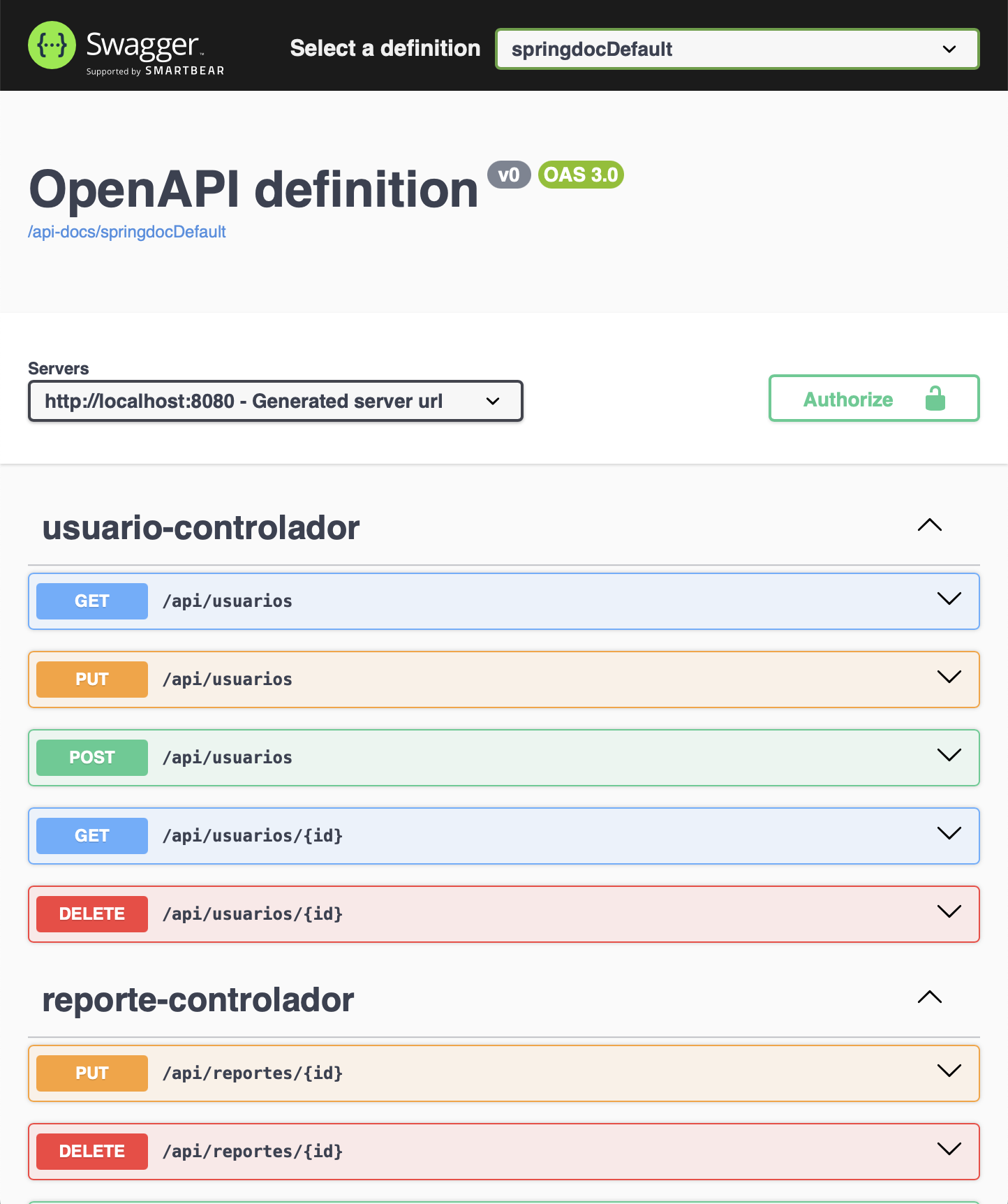
1. Finalmente, si todos los endpoints de un RestController están protegidos, entonces se debe agregar la siguiente anotación a nivel de la clase, debajo de la anotación @RequestMapping:

| @SecurityRequirement(name = "bearerAuth") |
| --- |

Observe que el nombre que tiene la anotación SecurityRequirement es el mismo que tiene SecurityScheme. Tenga en cuenta que si no todos los endpoints del controller están protegidos, sino algunos, entonces debe agregar la anotación anterior a cada método y no a la clase.

1. Una vez hecho esto, ponga a correr el proyecto de forma normal y vaya a la url: <http://localhost:8080/swagger-ui/index.html> (recuerde que el puerto por defecto es el 8080).

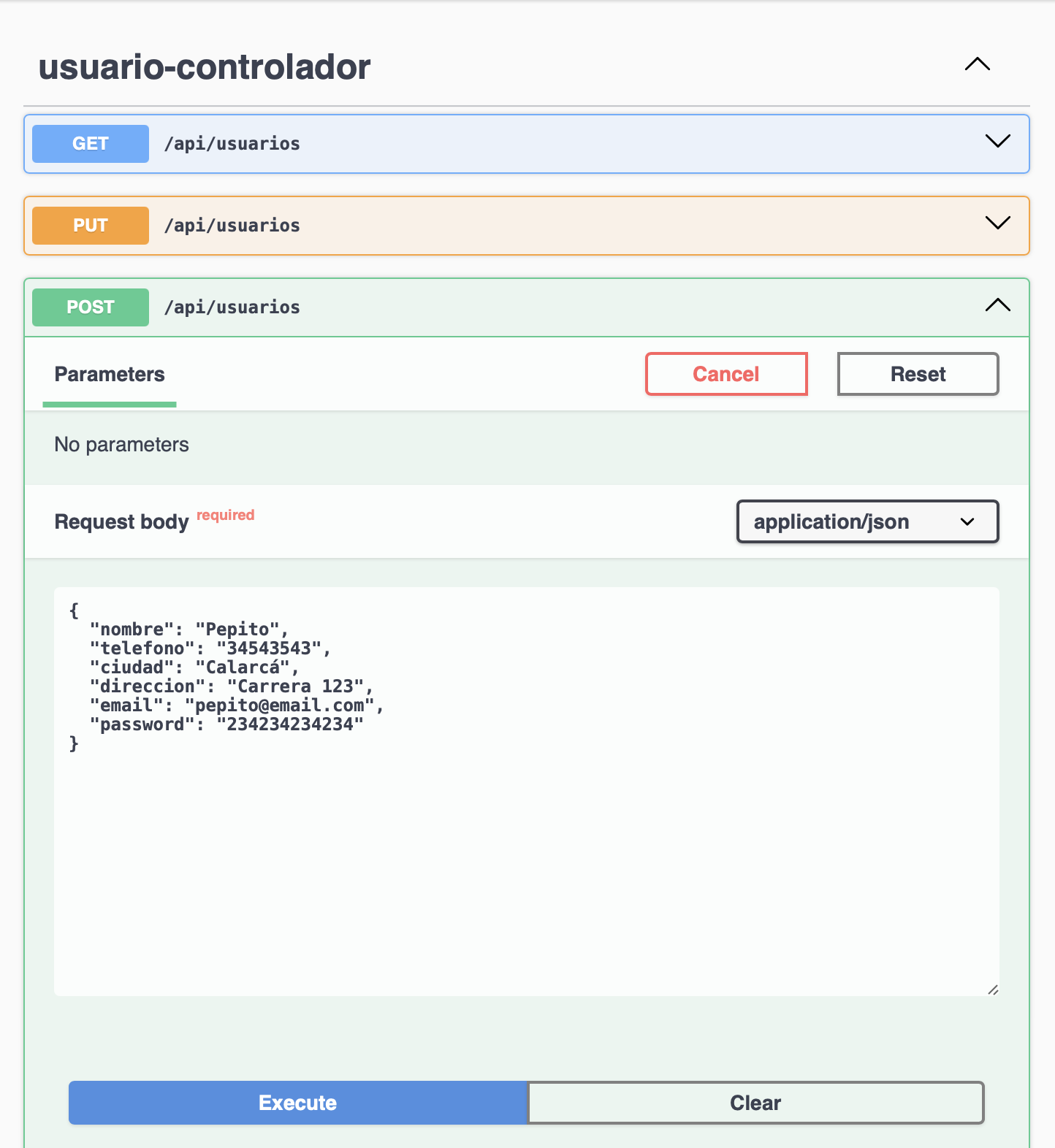
Al cargar la página, debe ver algo como se muestra en la siguiente imagen:



Lo interesante de Swagger es que genera la definición de la API y nos permite probarla directamente desde la página web. Incluso nos permite hacer la documentación de cada endpoint o controller usando la anotación @Operation. Por ejemplo, en la clase RestController de los usuarios se tiene un método que permite obtener los datos de un usuario dado su id, para agregar la documentación simplemente añadimos lo siguiente (encima del @GetMapping o @PostMapping, etc):

| @Operation(summary = "Detalle del usuario", description = "Permite acceder a la información de un usuario en específico") |
| --- |

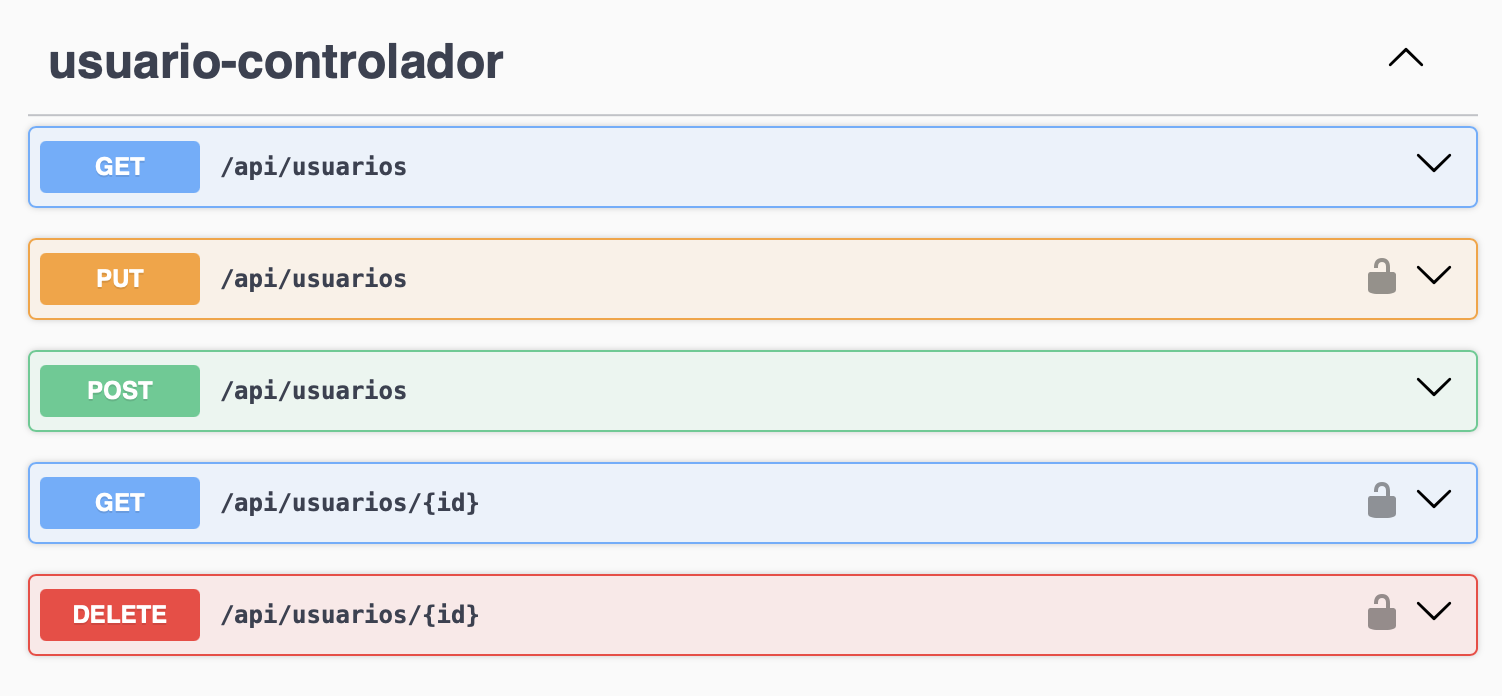
1. Si vamos a probar el servicio de registrar una nueva cuenta de usuario, vamos al endpoint creado, damos click, se despliega un formulario con un botón que dice “Try it out”, al dar click en ese botón se muestra un cajón de texto para crear el JSON que se le enviará a la petición, y un botón Execute.



Al dar click en el botón Execute, se nos arroja una respuesta con la información devuelta por el servicio:

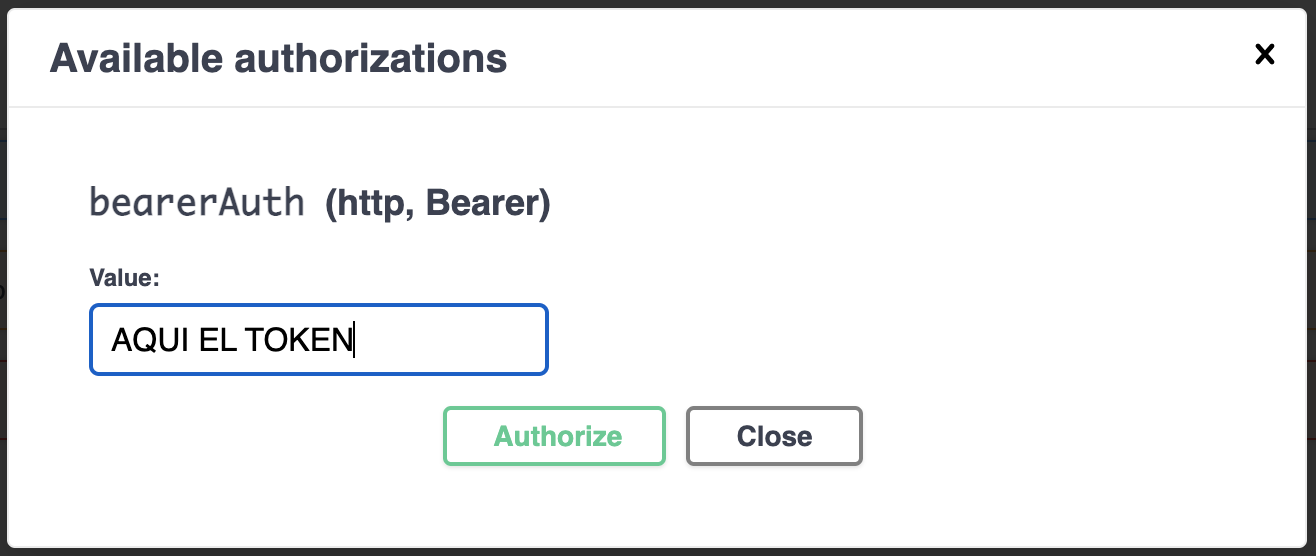


1. Para probar los endpoints que requieren un Token, se debe observar el candado a la derecha, así como se muestra en la siguiente imagen:



Este candado aparece como producto de la configuración que se hizo en el paso 3 con el @SecurityRequirement.

1. Al dar click en el candado aparecerá un modal para ingresar el token, como se muestra en la imagen:



Allí se ingresa el token y se da click en “Authorize”, y así puede probar el servicio que necesita. Por ahora puede poner cualquier cosa como Token ya que esto aún no lo hemos implementado.

Para más información de Swagger:

* <https://swagger.io/tools/swagger-ui/>
* <https://swagger.io/docs/specification/about/>
* <https://swagger.io/tools/swagger-editor/>

| **Información adicional (MockMVC)** |
| --- |

1. Otra manera de probar todos estos servicios es por medio de MockMvc. MockMvc es una herramienta que nos permite probar endpoints de Spring por medio de JUnit. Primero necesitamos crear una clase que se llame por ejemplo UsuarioControladorTest dentro del paquete co.edu.uniquindio.proyecto.test dentro de test/java y agregar lo siguiente:

| package co.edu.uniquindio.proyecto.test;  import com.fasterxml.jackson.databind.ObjectMapper;  import org.junit.jupiter.api.Test;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.boot.test.autoconfigure.web.servlet.AutoConfigureMockMvc;  import org.springframework.boot.test.context.SpringBootTest;  import org.springframework.test.web.servlet.MockMvc;  import org.springframework.test.web.servlet.result.MockMvcResultHandlers;  import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.\*;  import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.\*;  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.*RANDOM\_PORT*)  @AutoConfigureMockMvc  public class UsuarioControladorTest {  @Autowired  private MockMvc mockMvc;  @Autowired  private ObjectMapper objectMapper;  } |
| --- |

La anotación SpringBootTest tiene la propiedad webEnvironment que permite definir el modo en que se inicializará el servidor para las pruebas. Es importante que el nombre del paquete en test coincida con el de la clase ProyectoApplication (la clase que tiene el método main()).

1. Ahora creamos los métodos para probar el CRUD de las cuentas de los usuarios, así:

| @Test  public void crearTest() throws Exception {  //Se crea un objeto para realizar la creación de la cuenta  CrearUsuarioDTO cuentaDTO = new CrearUsuarioDTO(  "Carlos Andrés",  "3117812222",  "Armenia",  "Carrera 10 # 20-30",  "carlosandres@email.com",  "password123"  );  //Se realiza la petición POST al servidor usando el MockMvc y se valida que el estado de la respuesta sea 201  mockMvc.perform(*post*("/api/usuarios")  .contentType("application/json")  .content(objectMapper.writeValueAsString(cuentaDTO)))  .andDo(MockMvcResultHandlers.*print*())  .andExpect(*status*().isCreated());  }  @Test  public void actualizarTest() throws Exception {  //Se crea un token de autenticación para realizar la prueba  String token = "PEGAR AQUÍ TOKEN";  //Se crea un objeto para realizar la actualización de los datos de la cuenta  EditarUsuarioDTO cuentaDTO = new EditarUsuarioDTO(  "ID\_USUARIO",  "Carlos Andrés",  "Armenia",  "Nueva dirección",  "3117812222"  );  //Se realiza la petición PUT al servidor usando el MockMvc y se valida que el estado de la respuesta sea OK  mockMvc.perform(*put*("/api/usuarios")  .header("Authorization", "Bearer " + token)  .contentType("application/json")  .content(objectMapper.writeValueAsString(cuentaDTO)))  .andDo(MockMvcResultHandlers.*print*())  .andExpect(*status*().isOk());  }  @Test  public void eliminarTest() throws Exception {  //Se crea un token de autenticación para realizar la prueba  String token = "PEGAR AQUÍ TOKEN";  //Se realiza la petición DELETE al servidor usando el MockMvc y se valida que el estado de la respuesta sea OK  mockMvc.perform(*delete*("/api/usuarios/{id}", "ID\_USUARIO")  .header("Authorization", "Bearer " + token)  .contentType("application/json"))  .andDo(MockMvcResultHandlers.*print*())  .andExpect(*status*().isOk());  }  @Test  public void obtenerTest() throws Exception {  //Se crea un token de autenticación para realizar la prueba  String token = "PEGAR AQUÍ TOKEN";  //Se realiza la petición GET al servidor usando el MockMvc y se valida que el estado de la respuesta sea OK  mockMvc.perform(*get*("/api/usuarios/{id}", "ID\_USUARIO")  .header("Authorization", "Bearer " + token)  .contentType("application/json"))  .andDo(MockMvcResultHandlers.*print*())  .andExpect(*status*().isOk());  }  @Test  public void obtenerTodosTest() throws Exception {  //Se realiza la petición GET al servidor usando el MockMvc y se valida que el estado de la respuesta sea OK  mockMvc.perform(*get*("/api/usuarios")  .contentType("application/json"))  .andDo(MockMvcResultHandlers.*print*())  .andExpect(*status*().isOk());  } |
| --- |

MockMvc usa el RestController del cliente para ejecutar las peticiones que se realizan por medio de los endpoints y el método HTTP (GET, POST, DELETE, PUT).

Para más información:

* <https://blog.marcnuri.com/mockmvc-introduccion-a-spring-mvc-testing>
* <https://spring.io/guides/gs/testing-web/>

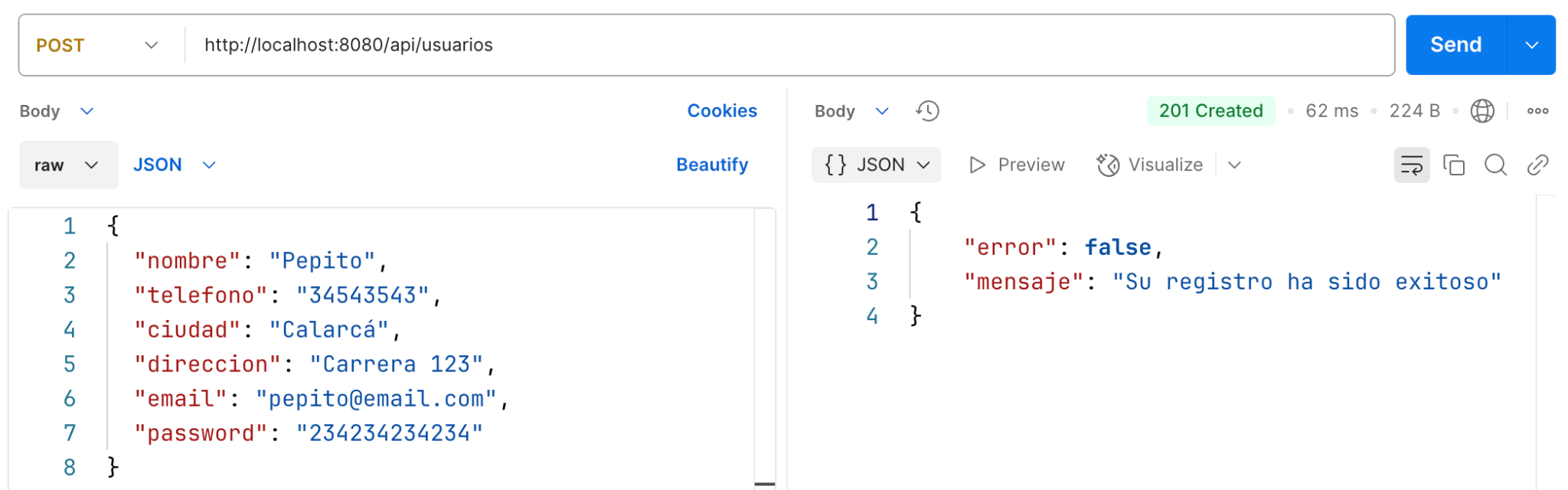
| **Herramientas para probar la API (Postman)** |
| --- |

1. Postman es una aplicación que nos permite enviar peticiones HTTP de forma rápida y sencilla. Para descargarlo podemos ir a <https://www.postman.com/downloads/>.
2. Pruebe todos los endpoints para el API del cliente haciendo uso de Postman.
3. Iniciamos probando los endpoints de autenticación ya que no requieren de un token para funcionar, por lo tanto probemos el de registro, usando la url: <http://localhost:8080/api/usuarios> este endpoint usa el método POST. Por defecto se usa el puerto 8080, pero puede ser diferente según su configuración.

Para testear los servicios que usan el método HTTP POST, seleccionamos el método POST, luego vamos al menú Body, allí seleccionamos raw y elegimos JSON, luego se escribe el objeto a guardar en notación JSON, así:

| {  "nombre": "Pepito",  "telefono": "34543543",  "ciudad": "Calarcá",  "direccion": "Carrera 123",  "email": "pepito@email.com",  "password": "234234234234"  } |
| --- |

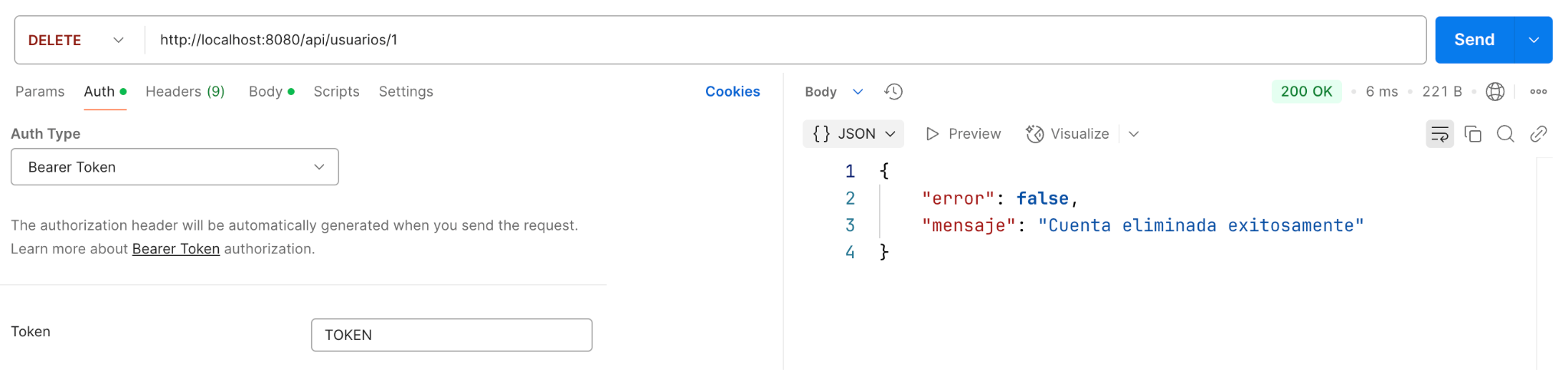
Así se ve la petición y respuesta en Postman:



Al enviar la petición debemos obtener la respuesta junto con el estatus definido en el controlador (como se observa en la imagen anterior). Observe que la estructura del JSON enviado al servicio de registro coincide con la clase CrearUsuarioDTO (los atributos se deben llamar igual y deben ser del mismo tipo de dato).

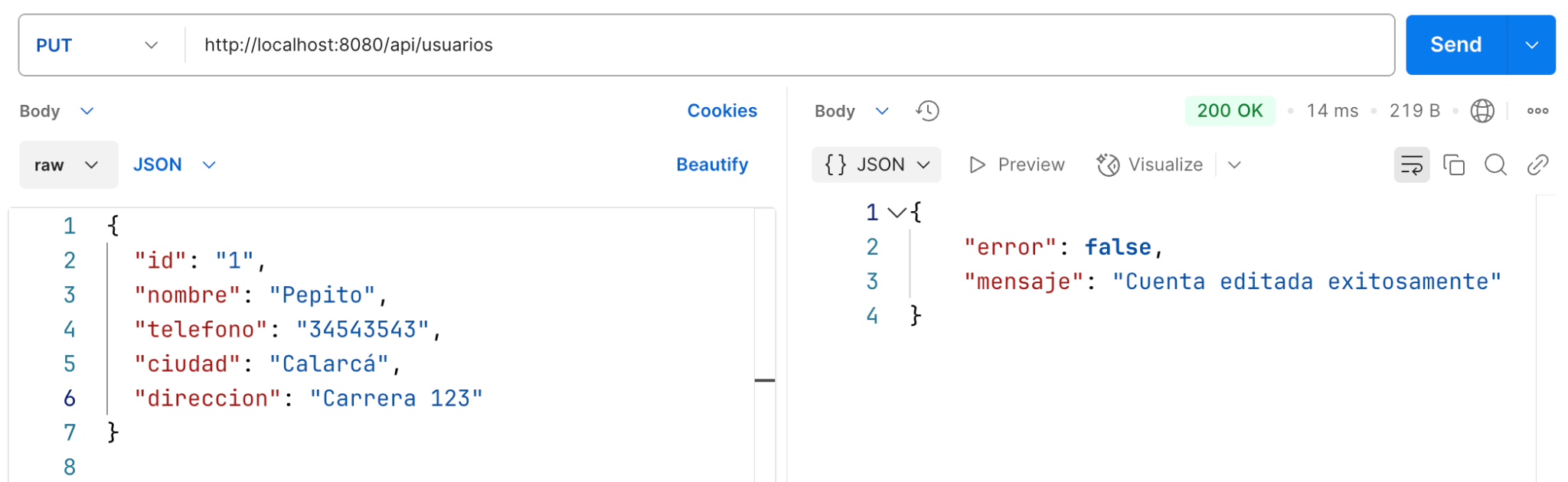
1. Para testear los servicios que usan el método HTTP GET, primero se elige el método GET y se escribe la url <http://localhost:8080/api/usuarios/ID> (la url depende del nombre que haya puesto en los @RequestMapping de cada controller). El ID es el (\_id) asignado a la cuenta del usuario.
2. Para testear los servicios que usan el método HTTP DELETE, seleccionamos el método DELETE, escribimos la ruta del endpoint, a la ruta le ponemos el código del cliente que vamos a borrar, en Body seleccionamos la opción none, y en Authorization asignamos el token de autenticación.

Por ejemplo, para eliminar un cliente usamos: <http://localhost:8080/api/usuarios/ID> elegimos el método HTTP DELETE y definimos el Token (cualquier texto por ahora). La url depende del nombre que haya puesto en los @RequestMapping de cada controller. Así se ve la petición y respuesta en Postman:



1. Para testear los servicios que usan el método HTTP PUT, seleccionamos el método PUT, escribimos la ruta del endpoint, en Body seleccionamos la opción raw JSON, y en Authorization asignamos el token de autenticación.

Por ejemplo, para actualizar un cliente usamos: <http://localhost:8080/api/usuarios> elegimos el método HTTP PUT y definimos el Token. La url depende del nombre que haya puesto en los @RequestMapping de cada controlador. Así se ve la petición y respuesta en Postman:



Observe que la estructura del JSON enviado al servicio de actualización coincide con la clase EditarUsuarioDTO (los atributos se deben llamar igual).

1. Pruebe el servicio de listar todos los usuarios. Este servicio usa HTTP GET. En este caso el body debe estar vacío.
2. Una vez probados todos los servicios (endpoints) podemos hacer uso de ellos para comunicar el backend con cualquier frontend. REST es una arquitectura que permite transferir información entre software que se ejecuta en diferentes plataformas, lenguajes y dispositivos.

Para más información de Postman acceda al siguiente enlace:

* <https://qalified.com/es/blog/postman-para-api-testing/>