Universidad Nacional Autónoma de México

Facultad de Ciencias

Documentación del Proyecto (2a iteración)

Equipo:
Skops & Company

Integrantes:
Flores Gutierrez José Luis
García López Francisco Daniel
Gómez López Erik Eduardo
Luna Campos Emiliano
Vázquez Reyes Jesús Elías

Asignatura:
Ingeniería de Software

Documentación del Proyecto Skops-Incidencias

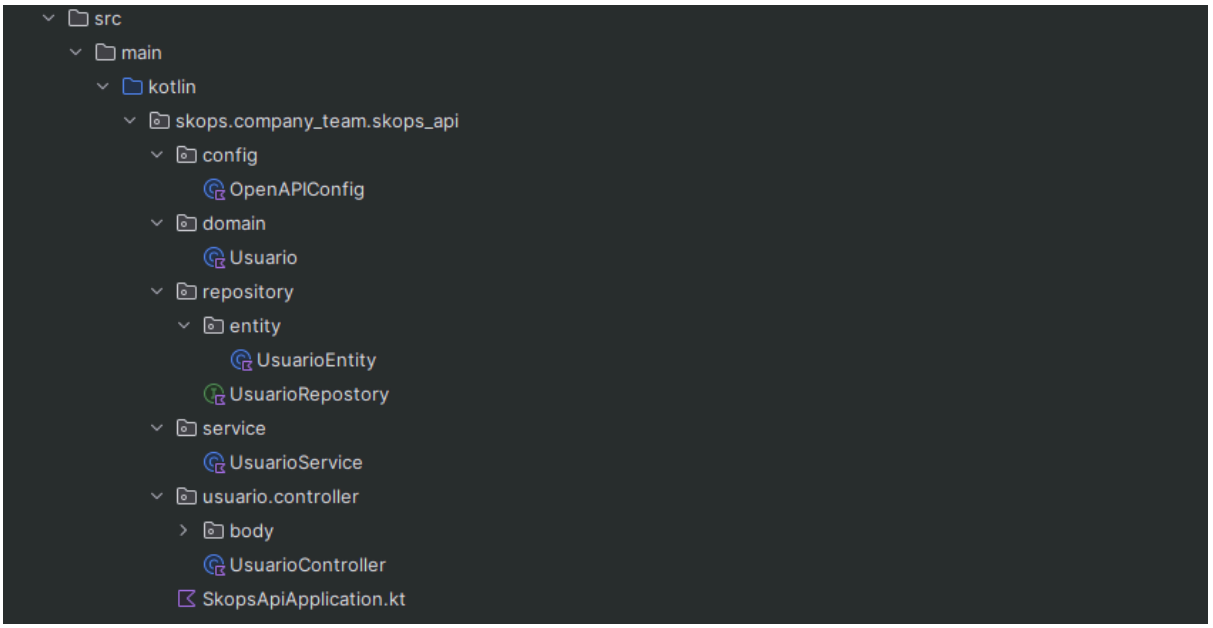
El proyecto Skops-Incidencias consta principalmente de dos módulos que sobresalen: el Backend y el Frontend, ambos estrechamente relacionados con las funciones que los usuarios pueden realizar en la aplicación de incidencias.

Backend

El *Backend* del proyecto está desarrollado en el lenguaje de programación **Kotlin**, esto debido fuertemente al framework que se utiliza para desarrollarlo, qué es *Spring*, que nos da varias herramientas para implementar un *Backend* completo en dicho lenguaje.

Este funciona a través de *Maven,* cuyo principal fin es la creación de proyectos completos, simplificando el compilar los distintos módulos que forman el componente en general.

Hacemos uso de las convenciones de la ingeniería de software para el *backend*. Primeramente, al organizar el componente en distintos paquetes que se relacionan entre sí. Nuestra estructura **actual** es la siguiente:



Se usa el paquete *entity* para definir los atributos que tendrá el usuario en la aplicación. El paquete *repository* funciona como conexión con la base de datos, ésta realiza las inserciones y consultas para actualizar la base de datos cada vez que el usuario hace uso de un recurso de la aplicación.

El paquete *service* recibe entradas en **JSON** para copiarlas en la base de datos, así como devolver salidas en formato **JSON** para que las funcionalidades devuelvan datos que serán guardados en la BD.
Finalmente, tenemos al paquete *controller* el cual define los *endpoints* que se usan en el *frontend* y a los cuales acude el usuario para realizar sus operaciones particulares. Este hace uso del paquete *service* y atrapa las posibles excepciones, además de realizar la conexión directa con la dirección del *frontend*.

Los *endpoints* definidos en *controller* son los siguientes:

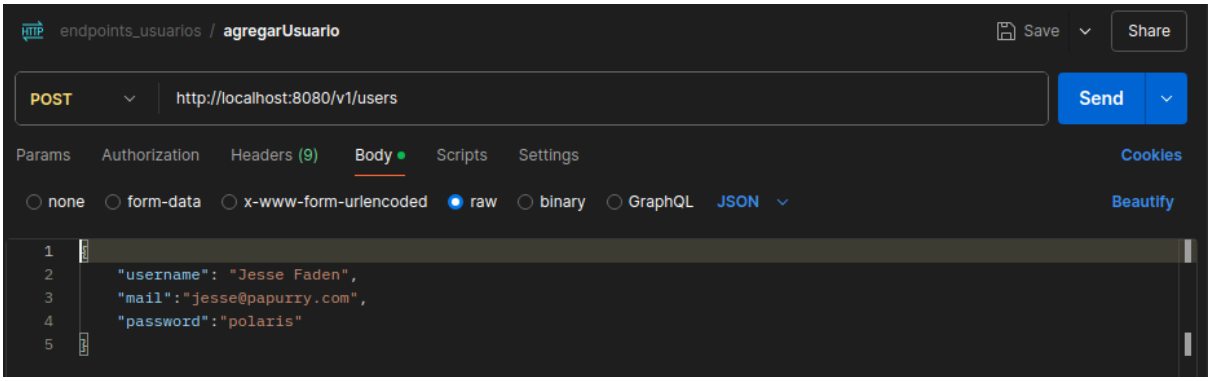
1.- POST: /v1/users/

Este *endpoint* sirve para registrar nuevos usuarios, a través de la función '*agregarUsuario*'.

```
import org.springframework.web.bind.annotation.CrossOrigin;

@CrossOrigin(origins = ["http://localhost:3000"])  Francisco Daniel García López +2
@RestController
@RequestMapping("/v1/users")

class UsuarioController (var usuarioService: UsuarioService){

    @PostMapping  Francisco Daniel García López +1
    fun agregarUsuario(@RequestBody usuarioBody: UsuarioBody): ResponseEntity<Any> {
        val miUsuario = Usuario(username = usuarioBody.username,
                                mail = usuarioBody.mail,
                                password = usuarioBody.password)
        val response = usuarioService.addUser(miUsuario)
        return ResponseEntity.ok(response)
    }
```

Recibe un cuerpo **JSON** con los parámetros: *username, mail, password.* en ese mismo orden para ser mapeados correctamente.

Ejemplo:

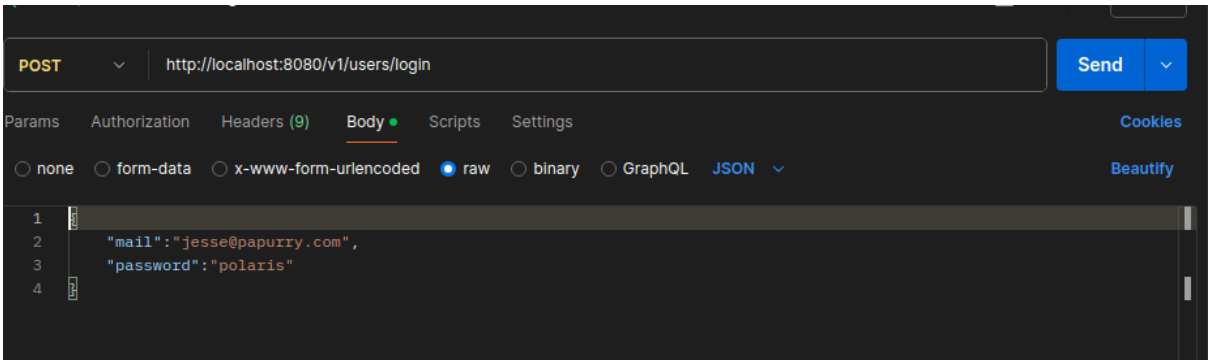


2.- POST: /v1/users/login

Este *endpoint* se encarga de ingresar a la aplicación una vez que se tiene una cuenta registrada. Recibe un **JSON** con un correo y contraseña que hayan sido registrados anteriormente, a través de la función *createLogin*.

```
@PostMapping("/login")  Francisco Daniel García López
fun createLogIn(@RequestBody loginUserBody: LoginUserBody): ResponseEntity<Usuario> {
    val result = usuarioService.login(loginUserBody.mail,loginUserBody.password)
    return if (result == null){
        ResponseEntity.notFound().build()
    } else {
        ResponseEntity.ok(result)
    }
}
```

Ejemplo:



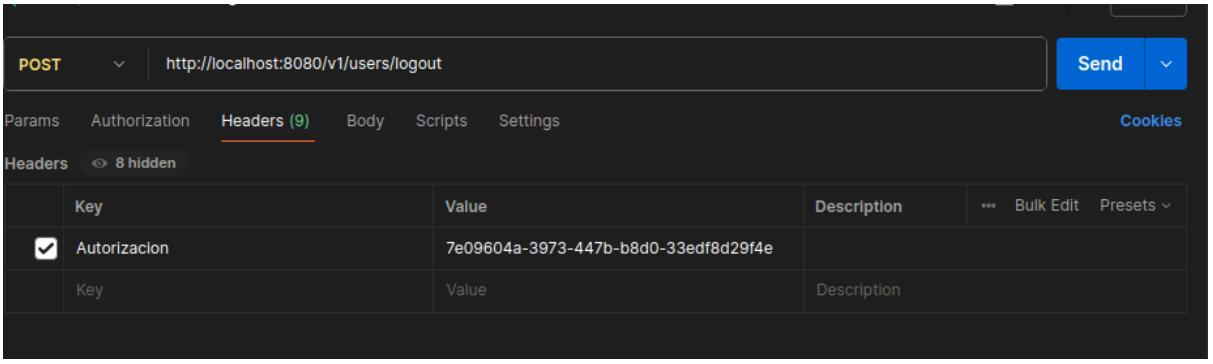
3.- POST: /v1/users/logout

Este *endpoint* sirve para cerrar sesión y que el usuario salga con éxito de la aplicación, a través de la función *createLogOut*.

```
@PostMapping("/logout")  Francisco Daniel García López
fun createLogOut(@RequestHeader("Autorizacion") token:String): ResponseEntity<String>{
    val logout = usuarioService.logout(token)
    return if (!logout){
        ResponseEntity.badRequest().build()
    } else {
        ResponseEntity.ok( body: "Sesión Cerrada")
    }
}
```

Cada vez que se ingresa en la aplicación, se genera un token único para dicho usuario, el cual se pide como autorización para poder cerrar sesión.

Ejemplo:

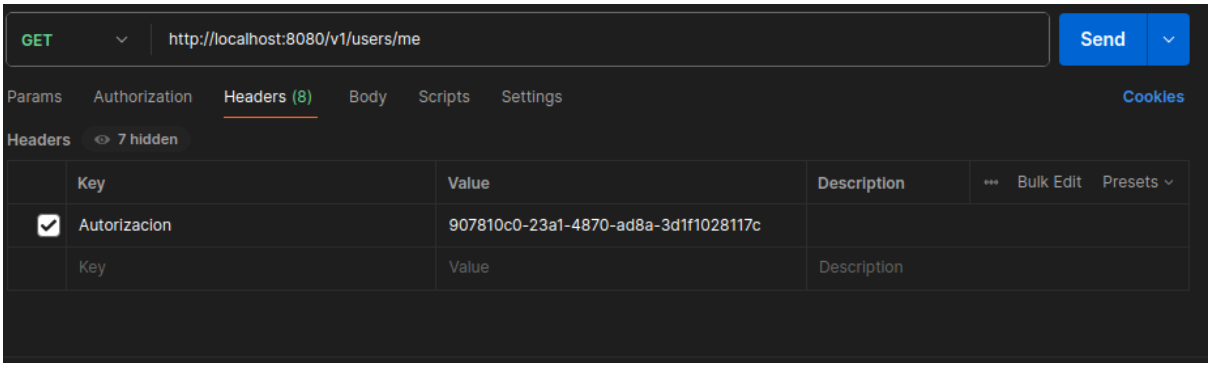


4.- GET: v1/users/me

Este *endpoint* sirve para poder ver la información de un usuario, a través de la función *meUsuario*. Al igual que *logout*, el *endpoint* funciona con el token único para pedirlo como autorización.

```
@GetMapping("/me")  Francisco Daniel García López
fun meUsuario(@RequestHeader("Autorizacion") token:String): ResponseEntity<Usuario>{
    val response = usuarioService.getInfoAboutMe(token)
    return if (response != null){
        ResponseEntity.ok(response)
    } else {
        ResponseEntity.status(401).build()
    }
}
```

Ejemplo:



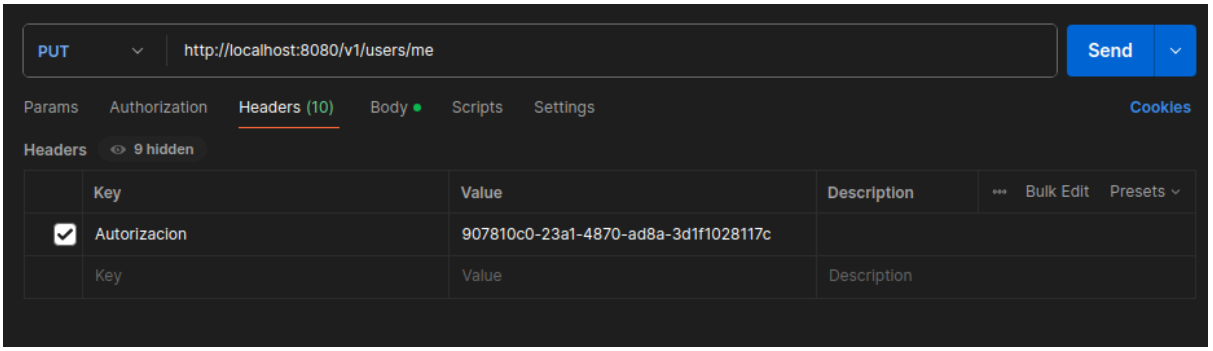
5.- PUT: GET: v1/users/me

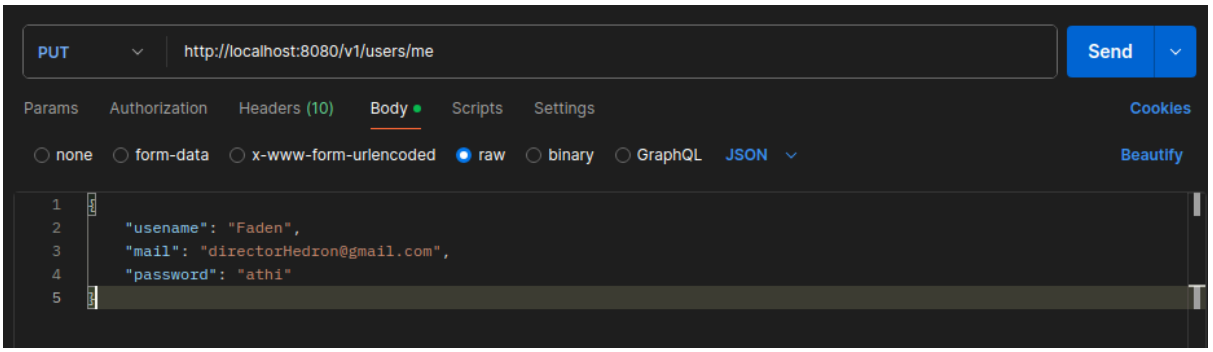
Este *endpoint* también sirve para actualizar la información del usuario.

```
@PutMapping("/me")  Francisco Daniel García López
fun actualizarUsuario(@RequestHeader("Autorizacion") token: String, @RequestBody usuarioUpdateBody
    if (token == "") {
        return ResponseEntity.status(401).build()
    }
    val usuarioActualizado = usuarioService.updateUser(token, usuarioUpdateBody)
    return if (usuarioActualizado != null) {
        ResponseEntity.ok(usuarioActualizado)
    } else {
        ResponseEntity.status(401).build()
    }
}

}
```

Recibe un ***JSON*** con los nuevos nombre de usuario, correo y contraseña. También se pedirá el token único como autorización.

Ejemplo:





Swagger

En la estructura del backend, contamos con una carpeta llamada *config* que posee dentro una clase de nombre ***OpenAPIConfig***, la cual se encarga de personalizar y configurar la documentación del *backend* utilizando la herramienta *Swagger*.

Asignamos un título propio, descripción, versión y agrupamiento a la API, mientras que, en la clase *UsuarioController*, colocamos ciertos comentarios para poder configurar esto de manera correcta.

El uso de Swagger nos permite, además de visualizar cada endpoint junto con una breve descripción y ejemplos, hacer uso de ellos y realizar pruebas sin necesidad de usar *Postman*.

En la sección de '**Ejecución**' se detalla de mejor manera cómo hacer uso de esto último.

Frontend

El *frontend* está desarrollado en el lenguaje React, ya que resultó ser el más sencillo de usar para nosotros, y así lograr facilitarnos la implementación de la interfaz que utilizará el usuario para poder reportar sus incidencias.

Nuestra interfaz tiene cinco funcionalidades (hasta el momento), las cuáles son **Registrarse, Iniciar Sesión**, **Mostrar Datos**, **Actualizar Datos** y **Cerrar Sesión**.

Comenzando con **Registrar**, en esta función el usuario debe ingresar un nombre de usuario, su email y una contraseña de manera obligatoria, para poder darlos de alta en nuestra base de datos. Esta acción es realizada por ***handleRegistro*** dentro de nuestros componentes, el cual hace un *Post* con los datos necesarios para registrar al usuario.

Similar a *Registrar*, en **Iniciar Sesión** el usuario ingresa su correo y contraseña con la que se registró anteriormente, y ahora ***handleLogin*** se encarga de buscarlo en la base de datos y asignarle un token temporal.

Luego tenemos dos funciones más que sirven para que el usuario pueda interactuar con su cuenta, las cuales son **Mostrar Datos** y **Actualizar Datos**.

Mostrar Datos funciona una vez que el usuario inició sesión. Se da click en la pestaña de 'Mi Cuenta', y se despliega la información del usuario. La clase *Cuenta* maneja y busca la información del usuario mediante el token y le regresa dos datos de éste. Cabe aclarar que, **por ahora**, solo se le muestra al usuario su usuario y

correo electrónico, dado que el token y contraseña son datos con un mayor grado de privacidad. Así mismo, no se le muestra su rol porque, en esta iteración, el rol de todos los usuarios es '*user*'.

Actualizar Datos se realiza mediante el componente homónimo, asegurándose que en la barra de navegación se pueda acceder a la actualización de los datos del usuario. El campo de la contraseña no es obligatorio, por lo que puede dejarse en blanco.

Por último tenemos **Cerrar Sesión**, que en la interfaz consiste en dos botones que, en el momento cuando se presione alguno de los dos, la clase Cuenta con ayuda de su método ***handleLogout*** hace un *Post* para buscar al usuario por token y después procede a eliminar dicho token de la base de datos. Finalmente, regresa al usuario a la página para iniciar sesión o registrarse.

Casos de Uso

Se realizaron diagramas de flujo para representar los casos de uso del sistema, así como las diferentes acciones que el usuario puede realizar por el momento. Decidimos por subir el documento de los diagramas por separado, dentro del mismo directorio *documentos/iteracion-2* del repositorio **skops-api**, esto porque consideramos que así es posible apreciarlos y analizarlos de mejor manera, a si los colocamos dentro de este mismo documento como capturas o imágenes.

Ejecución

A continuación se detalla cómo ejecutar ambos módulos: backend y frontend.

> Backend

1. Clonar el repositorio **skops-api**.
2. Correr un contenedor para la base de datos.
3. Ingresar en el directorio

   skops-api/src/main/resources/application.properties

   y cambiar los campos correspondientes:

   spring.datasource.username=nombreDeTuContenedor
   spring.datasource.password=contraseñaDeTuContenedor

4. Abrir un sistema manejador de bases de datos, y correr los scripts del directorio

   skops-api/documentos/base-de-datos

   Primero se debe ejecutar **BDD.sql**, para luego abrir y ejecutar el script **DDLv1.sql** en la base de datos recién creada por el script anterior: "Incidencias-DataBase".
5. Realizar la configuración del repositorio en IntelliJ (JDK, maven, Sync Project, Build Project, etc.)
6. Ejecutar la clase del directorio donde se encuentra la clase ejecutable, es decir:

   src/main/kotlin/skops/company_team/skops_api/SkopsApiApplication.kt

7. Para acceder a la documentación de Swagger y probar los endpoints sin necesidad de Postman, se ejecuta el backend y luego se ingresa a la dirección:

   http://localhost:8080/swagger-ui.html

> *Frontend*

1. Clonar el repositorio **skops-frontend.**
2. Para *Fedora*, se ejecutan los siguientes comandos en terminal:

   curl -fsSL https://rpm.nodesource.com/setup_20.x | sudo bash -
   sudo dnf install -y nodejs

3. Una vez terminado el proceso anterior, se ejecutan los siguientes comandos en la carpeta **skops-frontend**:

   rm -rf node_modules package-lock.json
   npm install
   npm start

4. Para *Ubuntu*, se ejecutan los siguientes comandos:

   sudo apt update
   sudo apt install -y nodejs npm

5. Comprobamos la instalación:

   node -v
   npm -v

6. Una vez terminado el proceso anterior, se ejecutan los siguientes comandos en la carpeta **skops-frontend**:

   rm -rf node_modules package-lock.json
   npm install
   npm start

7. La página se abre por sí sola en el navegador.