

Sistema de Gestión de Tareas Basado en Microservicios con Spring Boot, OAuth2, Docker y Angular.

ABDESAMAD OUBRAHIM AKKOUH

Contenido

Resumen.....	3
1 Introducción	3
1.1 Contexto del problema a resolver.....	3
1.2 Motivación detrás del proyecto	4
2 Arquitectura del proyecto	4
3 Descripción de los Microservicios	5
3.1 User	5
3.2 Tarea y Notificación.....	6
3.3 ConfigServer	7
3.4 EurekaServer	7
3.5 SpringBootAdmin	8
3.6 KaycoakAdapter	8
3.7 ApiGateway	9
3.8 MAVEN	10
4 Tecnologías utilizadas.....	10
4.1 Lenguajes de programación	10
4.2 Frameworks y herramientas:	10
4.3 Bases de datos.....	11
4.4 Contenedores y despliegue.....	11
4.5 Gestión de configuración y descubrimiento:	11
4.6 Autenticación y autorización.....	11
4.7 Monitorización	12
5 Pruebas y validación.....	12
5.1 Pruebas unitarias.....	12
5.2 Pruebas de integración	15
5.3 Errores y excepciones:	16
6 Creación, Gestión y Despliegue de Imágenes Docker con Docker Compose.....	17
6.1 Creación de una imagen Docker	17
7 Angular	19
7.1 Descripción de las Vistas y su Funcionalidad en Angular	19
8 Conclusión	25

Resumen

El proyecto es un sistema de gestión de tareas hecho en una arquitectura de microservicios utilizando Spring Boot, Docker y otras herramientas que explicare más adelante. Está desarrollado para ayudar a los usuarios a organizar sus tareas de forma eficiente, donde también, pueden compartirlas con otros usuarios.

El objetivo es proporcionar una solución que sea segura y fácil de moldear para futuros cambios, y así, facilitar la creación, asignación y seguimiento de tareas maximizando la productividad del usuario.

Algunas de las herramientas que utilizo para garantizar una arquitectura actual y funcional son las siguientes: Config Server, es un microservicio que accede a un repositorio local Git como fuente de datos para una configuración centralizada y asignación de perfiles. También utilizo API Gateway para gestionar las rutas y la seguridad, y Netflix Eureka para asegurar la comunicación entre los microservicios, ya que, con esta herramienta, se registran y descubren servicios. También utilizo Spring Boot Admin para supervisar el sistema. Keycloak para la autenticación y autorización segura. Para la base de datos utilizo PostgreSQL, gestionada con pgAdmin para facilitar la administración y analizar datos.

Muchas de estas herramientas las consigo desplegar de forma eficiente mediante Docker Compose. También utilizo Docker para crear contenedores independientes para cada uno de los microservicios. Para la documentación y prueba de las APIs, se emplean herramientas como Swagger y Postman.

1 Introducción

1.1 Contexto del problema a resolver

Hoy en día, muchas personas y equipos se enfrentan a dificultades para gestionar y organizar sus tareas de forma efectiva, ya que faltan herramientas centralizadas y adaptables. Existen soluciones, pero suelen ser costosas o difíciles de implementar en entornos personalizados.

1.2 Motivación detrás del proyecto

La principal motivación por la que hago este proyecto es para demostrar mis conocimientos en la arquitectura de microservicios, gestión de contenedores con Docker, tecnologías actuales como Spring Boot, etc. Además, este proyecto está diseñado como una solución backend que puede ser utilizada por un cliente frontend para la gestión de tareas. Ofrece flexibilidad para su integración en proyectos más grandes y se adapta a las necesidades del usuario.

2 Arquitectura del proyecto



La arquitectura del proyecto se divide en dos dominios:

InfraestructuraDomain

Este dominio contiene los microservicios que se encargan de la configuración y gestión de la infraestructura del proyecto, que son **ConfigServer**, **EurekaServer**, **SpringBootAdmin**, **ApiGateway** y **KeycloakAdapter**.

BusinessDomain

Este dominio contiene los microservicios que se encargan de la lógica de negocio, y son **User**, **Notificación** Y **tarea**.

Todos estos microservicios los explicaré más adelante cada uno.

Para simplificar la gestión de dependencias de los microservicios he utilizado carpetas pom padre. Esto me permite centralizar las dependencias comunes, permitiendo así,

que los microservicios hereden automáticamente las configuraciones o dependencias necesarias sin tener que repetir líneas en cada fichero pom de cada microservicio.

Soy consciente de que esta decisión puede afectar la independencia de cada microservicio. Al compartir un mismo POM padre, cualquier pequeño error o cambio en las dependencias comunes puede perjudicar todos los microservicios. Esto puede dificultar tanto el despliegue como el mantenimiento independiente de cada uno.

3 Descripción de los Microservicios

3.1 User

Este microservicio está diseñado para gestionar usuarios dentro del proyecto. Además está documentada utilizando Swagger OpenAPI para que otros desarrolladores puedan comprender mejor los endpoints y su uso mediante una interfaz visual y bien estructurada.

A continuación, iré explicando los distintos paquetes que se han organizado de forma que garanticen una programación basada en buenas prácticas, facilitando la claridad del código, el mantenimiento y la escalabilidad del proyecto en futuras implementaciones. Son los siguientes:

- `com.gestionatustareas.user`: Aquí se encuentra la clase principal `UsersApplication.java` que inicia el microservicio, y donde se configura y arranca la aplicación. Tiene una anotación `@SpringBootApplication`.
En esta clase también se encuentra un método que configura y personaliza la documentación de la API utilizando Swagger OpenAPI. Lo utilizo para generar una interfaz visual de la API con título y versión con Swagger, facilitando así, su exploración y las pruebas que se vayan a realizar.
- `com.gestionatustareas.user.business.transaction`: En este paquete se encuentra el archivo `BusinessTransaction.java` el cual contiene parte de la lógica de negocio. Esto se hace para disminuir código, y así, hacer que el `RestController` del microservicio sea más claro.
- `com.gestionatustareas.user.common`: Donde encontramos una clase llamada `StandardizedApiExceptionResponse.java`. Ésta se utiliza para definir un modelo estándar, REC 7807, de respuesta para errores o excepciones. Sirve para enviar

mensajes claros en caso de fallos en la API como errores de validación o problemas del servidor.

- `com.gestionatustareas.user.controller`: Aquí encontramos la clase `UserRestController.java` donde se definen los endpoints REST del microservicio para interactuar con los recursos de usuarios. Contiene métodos como para gestionar operaciones CRUD o cualquier otra funcionalidad del microservicio.
- `com.gestionatustareas.user.entities`: La clase que contiene el paquete es `Usuario.java` y representa la entidad Usuario que define la estructura de la tabla en la base de datos. Contiene sus respectivas anotaciones JPA y atributos como `id`, `nombre`, etc.
- `com.gestionatustareas.user.exception`: Donde encontramos dos clases. `ApiExceptionHandler.java` donde se maneja de forma global las excepciones, y capturar así, errores lanzados en el microservicio y personalizar las respuestas de errores que se envían al cliente. Está vinculado con `StandardizedApiExceptionResponse` para enviar respuestas estructuradas. `BusinessRuleException.java` sin embargo sirve para manejar excepciones específicas relacionadas con reglas de negocio en el proyecto. Proporciona información adicional como un `id` un código de error, etc. Lo cual facilita el control y la personalización de errores del microservicio.
- `com.gestionatustareas.user.repository`: Contiene la interfaz `UserRepository` que actúa como puente entre la aplicación y la base de datos. Extiende de `JpaRepository` para proporcionar métodos predefinidos para realizar operaciones CRUD sobre la entidad Usuario en este caso.

Ahora mismo, me falta una capa que sirva como puente entre el controlador y la entidad. Esto es importante porque hace que el controlador no tenga un acceso directo a la entidad, lo que lo hace más seguro y organizado. Para lograr esto se puede usar un mapeo con DTOs. Esto mejora la seguridad y hace que el proyecto sea más fácil de mantener y entender.

3.2 Tarea y Notificación

Estos dos microservicios son muy parecidos y más simples que el anterior. También se estructuran en diferentes paquetes para mantener una organización clara. Son los siguientes:

- `com.gestionatustareas.notificacion/tarea`: en este paquete, ambos microservicios contienen una clase principal que los inicializa.
- `com.gestionatustareas.notificacion/tarea.controller`: En éste tienen un controlador REST con endpoints relacionados con las notificaciones y tareas. Define las operaciones CRUD y en este caso no está documentado en Swagger OpenAPI por ahorro de tiempo.
- `com.gestionatustareas.notificacion/tarea.entities`: Contienen sus respectivas entidades Notificacion/Tarea que define la estructura de la tabla en la base de datos.
- `com.gestionatustareas.notificacion/tarea.enums`: Contiene los Enum de cada microservicio. En caso de la Notificación, se definen los tipos de ésta que puede haber, creación, actualización o un recordatorio de la tarea. Sin embargo, en la tarea indica el estado de ésta. Puede estar pendiente en progreso, completada o cancelada.
- `com.gestionatustareas.notificacion/tarea.repository`: Ambos contienen una interfaz que extiende `JpaRepository`, proporcionando métodos para interactuar con la base de datos, como guardar, buscar o eliminar.

3.3 ConfigServer

Este microservicio se encarga de centralizar el uso de configuraciones para los demás microservicios. Accede a un repositorio Git local donde se encuentran los archivos de propiedades yaml o properties, permitiendo así, administrar configuraciones por perfil, como local, preproducción y producción. Utiliza la anotación `@EnableConfigServer` en su clase principal para habilitar las funcionalidades de servidor de configuración. La conexión al repositorio Git local se desarrolla en su archivo de propiedades con los parámetros necesarios, para que los microservicios puedan acceder a las configuraciones, de forma centralizada, pasando por éste microservicio.

3.4 EurekaServer

Este actúa como un servidor de registro de servicios. Facilita la comunicación entre microservicios permitiendo que se registren y descubran entre ellos. Utiliza la anotación `@EnableEurekaServer` en su clase principal para obtener las funcionalidades de un

servidor Eureka. Desde su archivo de propiedades, se configuran parámetros como el puerto, las zonas de disponibilidad y las opciones de registro. Gracias a esto los microservicios cliente pueden registrarse en el servidor y consultar otros ya registrados. Es importante y que garantiza que el proyecto sea más flexible y escalable.

Con este accedemos a una interfaz gráfica que nos permite visualizar los microservicios registrados y obtener información sobre ellos.

3.5 SpringBootAdmin

Este microservicio monitoriza, administra y supervisa los microservicios del proyecto de forma centralizada. Utiliza las anotaciones `@Configuration`, `@EnableAutoConfiguration`, `@EnableDiscoveryClient`, `@EnableScheduling` y `@EnableAdminServer` para habilitar sus funcionalidades principales. Gracias al servidor de Netflix Eureka descubre automáticamente los microservicios registrados, permitiendo gestionar su estado y operaciones. Nos permite acceder a una interfaz gráfica donde se puede consultar información detallada como el estado de cada microservicio, logs, configuraciones activas, etc, facilitando su administración y asegurando un buen funcionamiento de todo el proyecto.

3.6 KaycoackAdapter

En este microservicio utilizo Keycloak Adapter para gestionar la seguridad de la aplicación implementando autenticación y autorización basadas en el protocolo OAuth2. Keycloak actúa como un servidor de identidad, permitiendo a los usuarios iniciar sesión de forma segura y obtener tokens de acceso para autenticar sus solicitudes. Con esto el microservicio puede validar las credenciales de los usuarios y gestionar sus permisos sin desarrollar directamente la lógica de seguridad, alojando estas responsabilidades en Keycloak para que el proyecto sea más simple y escalable.

Ahora paso a explicar los distintos paquetes y archivos que contiene el microservicio:

- **Com.gestionatustareas:** Aquí se encuentra la clase principal `KaycoackApplication.java` que inicia el microservicio, y donde se configura y arranca la aplicación. Tiene una anotación `@SpringBootApplication`.

- `com.gestionatustareas.controller`: En este paquete se encuentra la clase `IndexController` que es un controlador REST que gestiona la autenticación y validación de usuarios mediante tokens JWT con Keycloak.
- `com.gestionatustareas.exception`: Al igual que el microservicio User, en este paquete encontramos las clases `ApiExceptionHandler.java` y `BusinessRuleException.java` que se utilizan exactamente para lo mismo. Y como en este microservicio decido tener todo lo relacionado con las excepciones en un mismo paquete, también vemos que está la clase estándar `StandardizedApiExceptionResponse` ya que no creo otro paquete common, mostrando así, otra forma de gestionar u ordenar paquetes y archivos.
- `com.gestionatustareas.service`: Contiene tres archivos, el primero es la clase `JwtService` que se encarga de gestionar las claves públicas para validar tokens JWT enviados por Keycloak. `KeycloakRestService` sin embargo interactúa con Keycloak también para gestionar la autenticación, validación de tokens, roles de usuario, cierre de sesión y renovación de tokens. Y por último `RestTemplateConfig` que configura un cliente HTTP seguro para realizar solicitudes a Keycloak u otros servicios.

3.7 ApiGateway

Este microservicio dirige todas las solicitudes a los microservicios correspondientes. También se encarga de implementar seguridad, autenticación, autorización, y logging para las solicitudes que llegan al a todo el proyecto.

- `Com.gestionatustareas`: Como en todos los microservicios, aquí también se encuentra la clase principal `ApiGatewayApplication.java` que inicia el microservicio, y donde se configura y arranca la aplicación. Tiene una anotación `@SpringBootApplication`.
- `com.gestionatustareas.setups`: Primero nos encontramos con la clase `AuthenticationFiltering.java` que redefine los filtros de autenticación proporcionados por Keycloak y se conecta microservicio de `KeycloakAdapter` para gestionar la validación de las peticiones. Su objetivo es verificar si las solicitudes que llegan están autenticadas validando los tokens de éstas a través de Keycloak. Solo las peticiones autenticadas y con los permisos necesarios son redirigidas a los microservicios correspondientes.

GlobalPreFiltering sin embargo se encarga de realizar operaciones antes de enviar la solicitud al microservicio destino. Al contrario de GlobalPostFiltering que realiza operaciones después de recibir la respuesta de los microservicios.

3.8 MAVEN

Por último, en los archivos pom.xml utilizo dependencias para incluir bibliotecas y marcos de trabajo que son necesarios para el buen funcionamiento del proyecto. Uso dependencias tanto de servidor como de cliente. Algunas dependencias importantes son spring-boot-starter, que ofrece las configuraciones básicas para ejecutar la aplicación, spring-cloud-starter-config, que permite gestionar configuraciones externas y spring-cloud-starter-netflix-eureka-client, que permite la integración con Eureka para el descubrimiento de microservicios. También incluye spring-boot-devtools, que facilita el desarrollo con recarga automática y spring-boot-starter-test que ofrece herramientas para realizar pruebas en el proyecto.

4 Tecnologías utilizadas

4.1 Lenguajes de programación

En este proyecto sólo utilizo Java como lenguaje principal para el desarrollo del backend. Quiero añadir que en mi Git Hub podrás encontrar más proyectos que he realizado con distintos lenguajes de programación como javascript, PHP, etc. También encontrarás otros en los que utilizo Bootstrap, html, css, etc.

4.2 Frameworks y herramientas:

El framework que utilizo es Spring Boot y es el principal para la creación de microservicios, incluyendo componentes como Spring Data JPA para la gestión de datos y Spring Security para la seguridad.

Para documentar las APIs, y facilitar así, la comprensión para otros desarrolladores que vayan a utilizar el proyecto, he usado Swagger OpenAPI. Y para probar y validar los endpoints de cada controlador rest he utilizado Postman.

4.3 Bases de datos

He utilizado PostgreSQL como sistema de gestión de bases de datos relacional por su buen rendimiento. La base de datos es administrada mediante pgAdmin, que permite gestionar las tablas y ejecutar consultas de forma eficiente.

4.4 Contenedores y despliegue

He implementado Docker para contenerizar los microservicios mediante un archivo dockerfile en cada microservicio, lo que garantiza un entorno de ejecución consistente. Además, he utilizado Docker Compose para orquestar múltiples contenedores y facilitar la integración entre los diferentes servicios, como la base de datos PostgreSQL, Keycloak y otros microservicios.

4.5 Gestión de configuración y descubrimiento:

Para centralizar la configuración de los microservicios uso Spring Cloud Config Server. Utilizo un repositorio Git local como fuente de datos para los perfiles.

Y Netflix Eureka para el registro y descubrimiento de los microservicios, facilitando la comunicación entre ellos.

4.6 Autenticación y autorización

Para esto utilizo Keycloak como servidor de autenticación, proporcionando una solución robusta para gestionar usuarios, roles y permisos.

4.7 Monitorización

He usado Spring Boot Admin para monitorear el estado y el rendimiento de los microservicios, asegurando así, su correcto funcionamiento.

5 Pruebas y validación

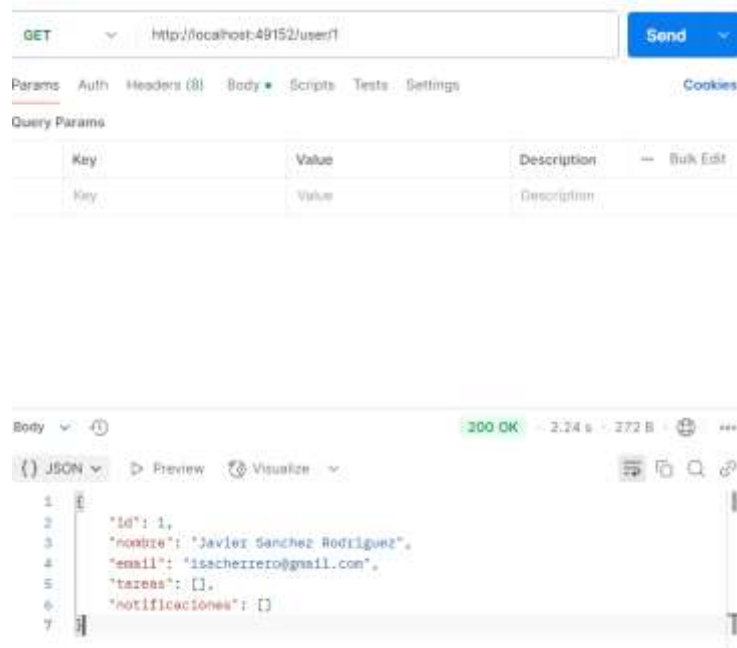
5.1 Pruebas unitarias

He probado cada método de la API de los microservicios individualmente para garantizar que las operaciones CRUD y la lógica de negocio funcionan correctamente.

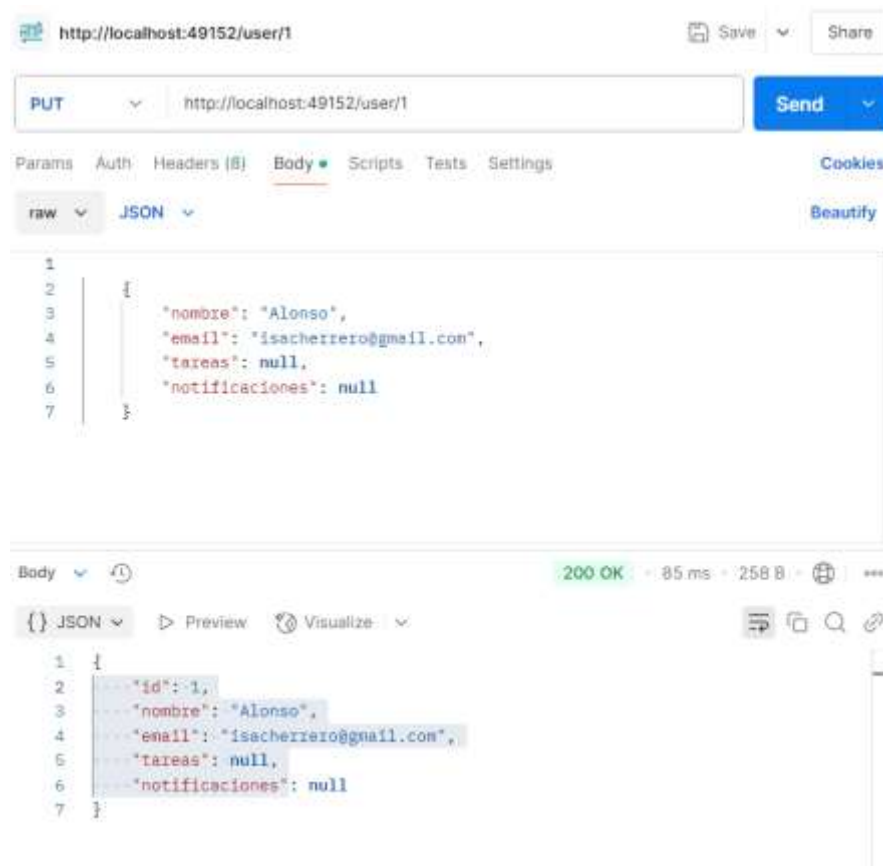
He usado Postman y Swagger para ejecutar las pruebas y verificar las respuestas.

Code	Details
200	<div>Response body</div> <pre>[{ "id": 1, "nombre": "Javier Sanchez Rodriguez", "email": "isacherrero@gmail.com", "tareas": null, "notificaciones": null }]</pre>

En primer lugar, ejecuto el método POST de la API del microservicio usuario. Después de esto pruebo el método GET donde se muestra la información del usuario guardado, y verificando así, el buen funcionamiento de ambos métodos como se muestra en la imagen.



Después compruebo el método GET `user/{id}`, y como vemos en la imagen, retorna el valor esperado.



Con este método PUT, mi objetivo es cambiar el nombre del usuario con id 1, y como comprobamos en la imagen, el valor retornado es el esperado.

DELETE Send

Params Auth Headers (8) Body Scripts Tests Settings Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body 200 OK • 48 ms • 123 B • ...

Raw Preview Visualize

1

Ahora compruebo el método DELETE. Al eliminar el usuario con Id 1 vemos que el retorno es el esperado ya que nos retorna un código 200.

Estas pruebas, se han llevado a cabo con todos los microservicios de la misma forma que éste.

También quiero añadir que, en este caso, el microservicio User utiliza puerto 49152, garantizando así, que accede al archivo de configuración config-client-prod de Config Server que usa este puerto.

Code Details

200

Response body

```
Hello your proerty value is: develop
```

Por último, compruebo el perfil que se está utilizando en ese momento con este método llamado check. Con esta imagen se compruebo que la respuesta retornada es la indicada.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
APIGATEWAY	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.apigateway.R061
BUSINESSDOMAIN-NOTIFICACION	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.businessdomain-notificacion.R061
BUSINESSDOMAIN-TAREA	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.businessdomain-tarea.R061
CONFIG-SERVER	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.config-server.R061
KEYCLOCK	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.keyclock.R061
SPRINGBOOTADMIN	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.springbootadmin.R061
USER	n/a (1)	(1)	UP (1) - DESKTOP-V5QHF1Q.user.R061

Gracias a esta imagen, comprobamos también que todos los microservicios se registran correctamente en el servidor de Eureka.



Aquí también comprobamos que se registran en el de Spring Boot Admin correctamente.

```
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.g.setups.globalpostfiltering : Global Post Filter ejecutado
INFO 17412 --- [apigateway] [ctor-http-nio-3] r.n.http.client.HttpClientOperations : [03d2c3e2-1, 1:/192.168.5.2:56708]
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.a.c.g.h.RoutePredicateHandlerMapping : Route matched: user_service
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.a.c.g.h.RoutePredicateHandlerMapping : Mapping [exchange: PORT, http://192.168.5.2:56708] Mapped to org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.a.c.g.handler.FilteringWebHandler : Sorted gatewayFilterFactories: []
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.g.setups.globalprefiltering : Global prefilter ejecutado
INFO 17412 --- [apigateway] [ctor-http-nio-3] f.f.h.o.ObservedHttpRequestHttpHeadersFilter : Will instrument the HTTP request
INFO 17412 --- [apigateway] [ctor-http-nio-3] f.f.h.o.ObservedHttpRequestHttpHeadersFilter : Client observation [name=http.request, type=HTTP, uri=http://192.168.5.2:56708]
INFO 17412 --- [apigateway] [ctor-http-nio-3] r.netty.http.client.HttpClientConnect : [03d2c3e2-2, 1:/192.168.5.2:56708]
INFO 17412 --- [apigateway] [ctor-http-nio-3] r.n.http.client.HttpClientOperations : [03d2c3e2-2, 1:/192.168.5.2:56708]
INFO 17412 --- [apigateway] [ctor-http-nio-3] f.f.h.o.ObservedResponseHttpHeadersFilter : Will instrument the response
INFO 17412 --- [apigateway] [ctor-http-nio-3] f.f.h.o.ObservedResponseHttpHeadersFilter : The response was handled for observation
INFO 17412 --- [apigateway] [ctor-http-nio-3] o.g.setups.globalpostfiltering : Global Post Filter ejecutado
```

Gracias a esta imagen comprobamos que las solicitudes http pasan por el ApiGateway ya que se activa el GlobalPostFiltering y GlobalPretFiltering.

5.2 Pruebas de integración

Se ha realizado una prueba específica desde el microservicio User, donde se valida la interacción con los microservicios Tarea y Notificación gracias al método full. Este método accede a los demás microservicios para visualizar a los usuarios con sus tareas y notificaciones correspondientes.

Code	Details
200	<p>Response body</p> <pre>{ "id": 1, "nombre": "Javier Sanchez Rodriguez", "email": "isacherrero@gmail.com", "tareas": [{ "id": 1, "titulo": "string", "descripcion": "string", "estado": "PENDIENTE", "fechaCreacion": "2025-01-14T23:06:08.306", "fechaVencimiento": "2025-01-14T23:06:08.306", "propietariosIds": [1] }], "notificaciones": [{ "id": 1, "usuario_id": 1, "tipo": "CREACION", "mensaje": "hola soy una notificación", "fechaCreacion": "2025-01-14T23:06:32.031", "fechaVencimiento": "2025-01-14T23:06:32.031", "leido": true }] }</pre>

Aquí comprobamos que la conexión es correcta ya que la respuesta retornada es la esperada.

5.3 Errores y excepciones:

Las respuestas de los errores y el manejo de excepciones han sido comprobadas para asegurar que los mensajes sean claros y estructurados, siguiendo el estándar REC 7807.

```

{
  "type": "TECNICO",
  "title": "Input Output error",
  "code": "1024",
  "detail": "Method parameter 'id': Failed to convert value of type 'java.lang.String' to required type 'long'; For input string: \"0,1\"",
  "instance": null
}

```

Así, por ejemplo.

6 Creación, Gestión y Despliegue de Imágenes Docker con Docker Compose

6.1 Creación de una imagen Docker

Gracias al Dockerfile genérico que tiene cada microservicio puedo crear las imágenes.

```
C:\Users\usuario>cd C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas\infraestructuradomain
\springBootAdmin

C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas\infraestructuradomain\springBootAdmin>
```

En primer lugar, accedo a la ubicación del microservicio, que, en este caso, es SpringBootAdmin. Aquí que es donde se encuentra el archivo Dockerfile.

```
C:\Users\usuario\OneDrive\Escritorio\Proyectos\HackingServices\GestionFacusAreas\Infraestructura\Domain\springbootAdmin>docker build -t springbootadmin_image --no-cache --build-arg JAR_FILE=target/springbootadmin-0.0.1-SNAPSHOT.jar .
```

```
[+] Building 26.8s (9/9) FINISHED
```

	docker:desktop-linux
>> [internal] load build definition from Dockerfile	0.3s
>> <=> transferring dockerfile: 409B	0.0s
>> [internal] load metadata for docker.io/library/ubuntu-tremur-in-alpine	2.5s
>> [internal] load dockerignore	0.2s
>> transferring context: 2B	0.0s
>> [2/4] FROM docker.io/library/ubuntu-tremur-in-alpine@sha256:b3d1cd2ed7fddadccda33bd921c36cfdb71c077fb6bf75e66978ba20709febe	18.4s
>> resolve docker.io/library/ubuntu-tremur-in-alpine@sha256:b3d1cd2ed7fddadccda33bd921c36cfdb71c077fb6bf75e66978ba20709febe	0.1s
>> sha256:b3d1cd2ed7fddadccda33bd921c36cfdb71c077fb6bf75e66978ba20709febe 1.37kB / 1.37kB	0.3s
>> sha256:b3d1cd2ed7fddadccda33bd921c36cfdb71c077fb6bf75e66978ba20709febe 1.94kB / 1.94kB	0.9s
>> sha256:c6ef8401dc39307cb3ebaf93cbb9026034924fa2ca19d31.94kB / 1.94kB	0.6s
>> sha256:66a3d08831fa52324b843e6867619c78dab45949216aa5ce696ae684b79b 3.47kB / 3.47kB	0.3s
>> sha256:dfeb1617e78d9484b7682d2176ed2ab4bc94126e1aabcc8085705194f83d3b 26.87MB / 26.87MB	2.1s
>> sha256:e984dcdf75c32d4ac3d87fc3b7e7f3c8a080fcb6b4b389ace3d4cfa1abb 101.69MB / 143.69MB	16.1s
>> sha256:3ec71adfd42892143649f7087ff1886d33c1306444510fac4ee052621ee7de 128B / 128B	0.9s
>> extracting sha256:66a3d08831fa52324b843e6867619c78dab45949216aa5ce696ae684b79b 3.47kB / 3.47kB	0.4s
>> sha256:b9aba4b45d6e708472f4b170dde179ade8007f4c2236d4b45d9438717a5718352 7.24kB / 7.24kB	1.1s
>> extracting sha256:dfeb1617e78d9484b7682d2176ed2ab4bc94126e1aabcc8085705194f83d3b 26.87MB / 26.87MB	1.7s
>> extracting sha256:e984dcdf75c32d4ac3d87fc3b7e7f3c8a080fcb6b4b389ace3d4cfa1abb 101.69MB / 143.69MB	3.6s
>> extracting sha256:3ec71adfd42892143649f7087ff1886d33c1306444510fac4ee052621ee7de 128B / 128B	0.6s
>> extracting sha256:b9aba4b45d6e708472f4b170dde179ade8007f4c2236d4b45d9438717a5718352 7.24kB / 7.24kB	0.8s
>> [internal] load build context	0.3s
>> transferring context: 45.13KB	0.2s
>> [2/4] RUN addgroup -s 182M devspoc && adduser -o -U devspoc admin	1.1s
>> [3/4] COPY target/springbootadmin-0.0.1-SNAPSHOT.jar /tmp/app.jar	0.4s
>> [4/4] RUN chmod -R admin/devspoc /tmp	0.5s
>> exporting to image	0.6s
>> exporting layers	0.7s
>> writing image sha256:cfd3dc79cedc41e7094fa883b943581fad507a0f1c4e47700d8ae9f54a	0.6s
>> naming to docker.io/library/springbootadmin_image	0.3s

```
View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/jatq8zwanjkw9k3lytbx2
```

What's next:

View a summary of image vulnerabilities and recommendations + docker scout quickview

Posteriormente pasamos a crear la imagen con el comando docker build pasando un argumento a la variable JAR_FILE. Esta variable se crea como argumento en el Dockerfile y se hace para para que éste sea flexible reutilizable.

Como comprobamos en la imagen, se crea correctamente. Esto se hace con todas las imágenes Docker que he creado.

```
C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas\businessdomain\tarea>docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tarea_image	latest	8388b433acf1	6 seconds ago	517MB
user_image	latest	e27f73a737eb	5 minutes ago	517MB
eureka-server_image	latest	116e448c9734	5 minutes ago	455MB
apigateway_image	latest	0aa02d803693	6 minutes ago	452MB
notificacion_image	latest	b7de19a25e86	6 minutes ago	517MB
springbootadmin_image	latest	45d2579cd2ce	7 minutes ago	461MB
keycloakadapter_image	latest	809e3d3c1e0e	8 minutes ago	439MB
configserver_image	latest	c864572c5dcf	11 minutes ago	454MB

Aquí se muestran todas las imágenes de los microservicios.

```
C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas\infrastructuredomain\springbootadmin>cd C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas
```

Ahora accedo a la ubicación que se muestra en la imagen ya que es donde se encuentra el archivo de Docker Compose.

```
C:\Users\usuario\OneDrive\Escritorio\ProyectosMicroservicios\GestionaTusTareas>docker compose up -d
```

```
[+] Running 12/12
```

- ✓Network gestionatustareas_default Created
- ✓Container postgres Started
- ✓Container id-eureka Started
- ✓Container id-keycloakServer Started
- ✓Container pgadmin Started
- ✓Container bd-tarea Started
- ✓Container bd-user Started
- ✓Container id-notificación Started
- ✓Container id-configserver Started
- ✓Container id-springbootadmin Started
- ✓Container id-apigateway Started
- ✓Container id-keycloakadapter Started

Por último, se levanta en segundo plano los servicios en el archivo de docker-compose, que acceden a sus respectivas imágenes, gracias al comando docker compose up -d.

7 Angular

Para el frontend del backend he decidido usado Angular y he trabajado con Standalone Components. Esto elimina la necesidad de declarar los componentes dentro de módulos. Cada componente es independiente, lo que hace más fácil la reutilización de código. Gracias a esta arquitectura, también mejora el rendimiento de la aplicación, ya que, al no depender de los módulos, la carga de los componentes es más rápida.

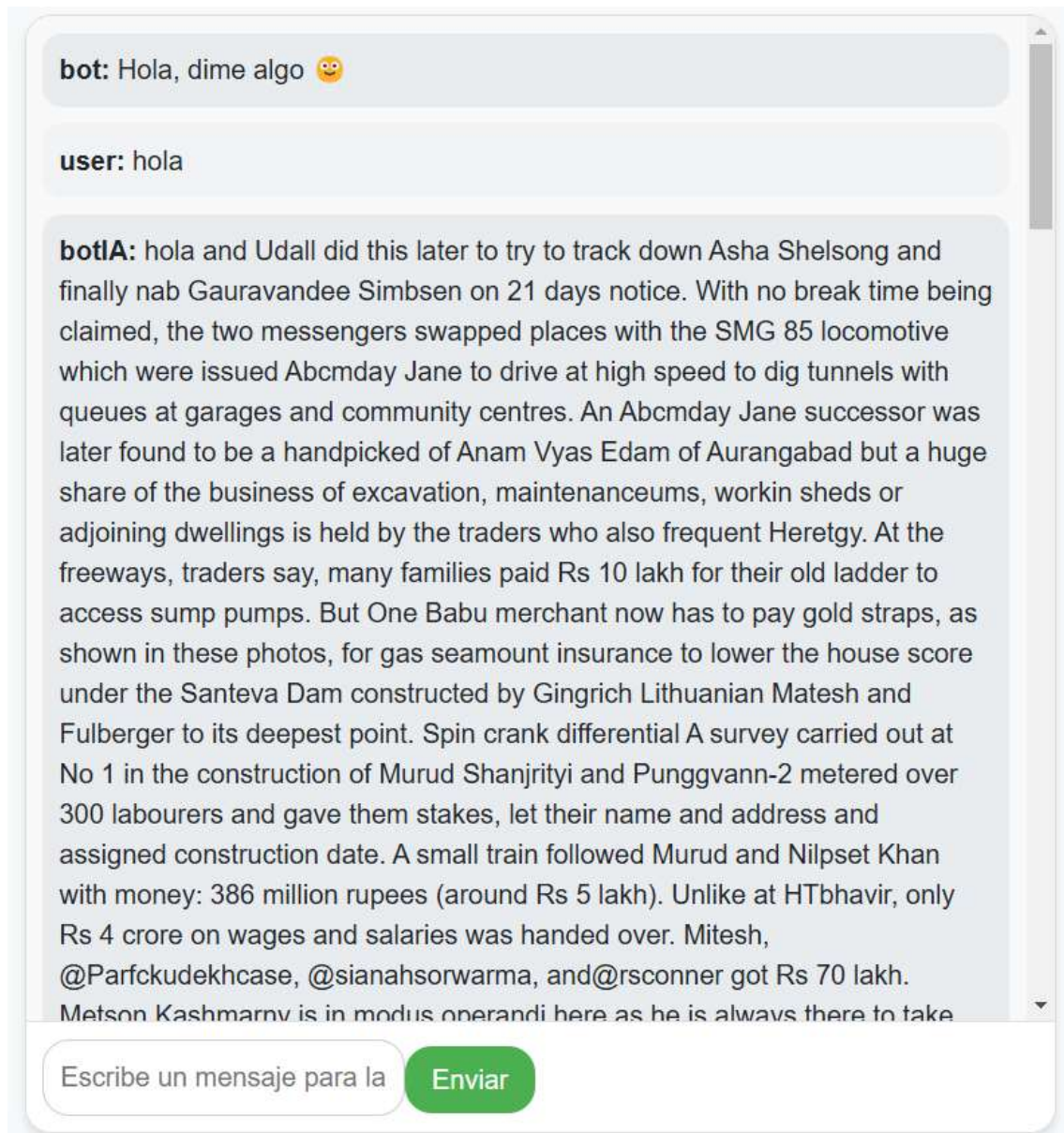
Mi objetivo con este front es demostrar mis habilidades utilizando Angular para interactuar con una API, en este caso, la de gestión de usuarios. La aplicación está diseñada para mostrar cómo puedo consumir y utilizar los datos de una API utilizando Angular.

Quiero añadir que en mi Git Hub podrás encontrar proyectos donde demuestro mis habilidades con frameworks como Laravel y cakePHP, en lenguajes como Javascript y Php. También en HTML, CSS y Bootstrap.

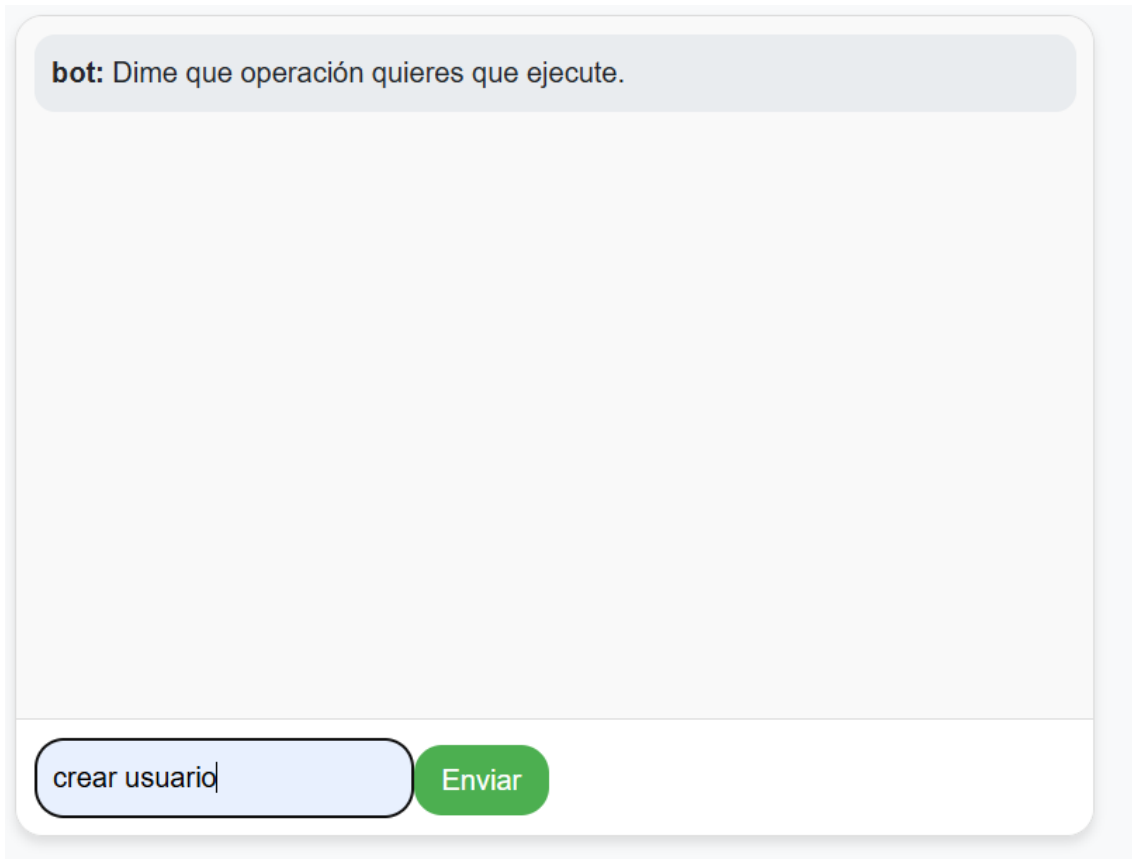
7.1 Descripción de las Vistas y su Funcionalidad en Angular



Esta es la vista principal del proyecto. Son dos Chatbots con diferentes funcionalidades. Uno se encarga de realizar operaciones CRUD, y el otro accede a una API externa de Hugging Face, usando un token personal con el objetivo de interactuar con una IA. Pasaré a explicar el segundo con la siguiente imagen:



Cuando se le envía un mensaje “hola” por ejemplo, el Chatbot envía esta entrada al modelo de Hugging Face, el cual, devuelve una respuesta aleatoria. Esto es porque estoy utilizando un modelo gratuito y no es eficiente.



bot: Dime que operación quieres que ejecute.

crear usuario

Enviar

En el primer Chatbot, al enviar un mensaje donde ponga “crear usuario” se abre un modal donde nos muestra un formulario para registrar un usuario.



Crear Usuario ×

Nombre:

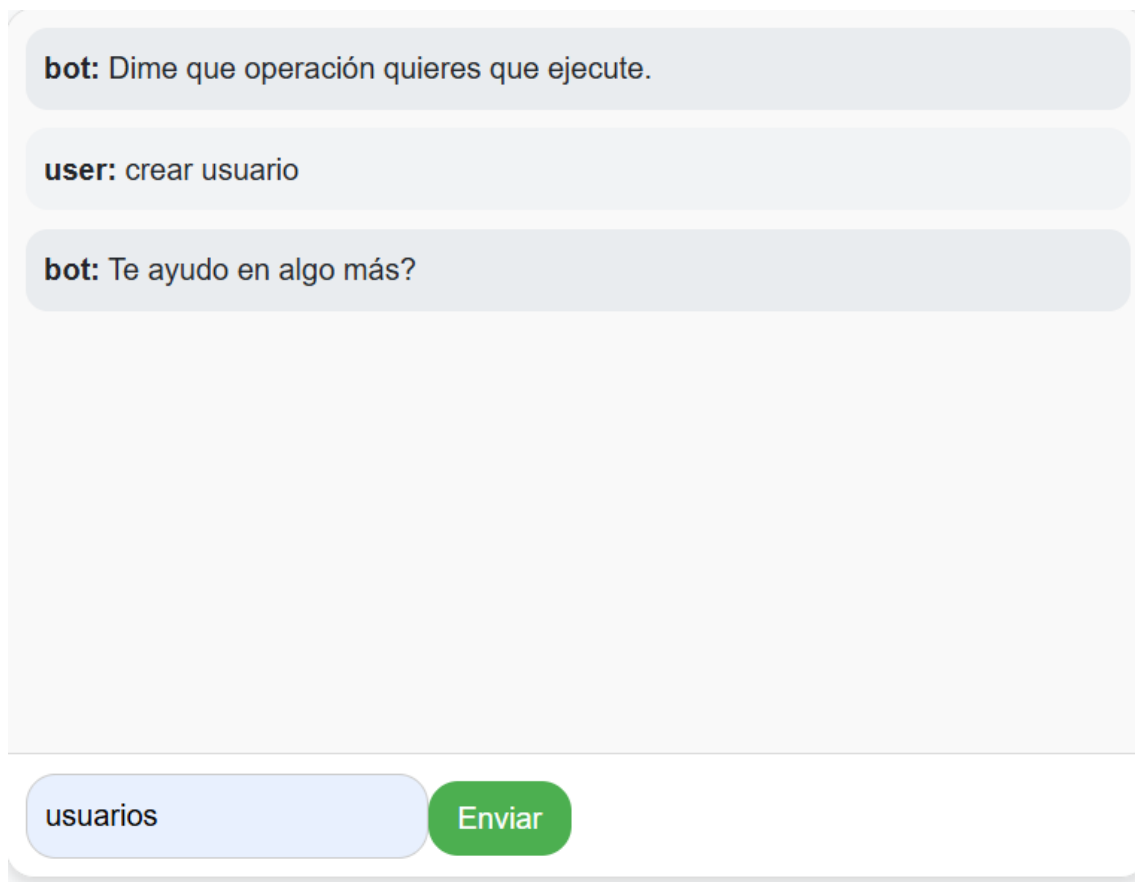
Abdel

Correo:

abdel.oub9@gmail.com

Crear Usuario

Registramos el usuario con los siguientes datos y al darle a la X el modal se cierra.



The screenshot shows a chat window with a light gray background. It contains three messages in rounded rectangular bubbles. The first bubble is light blue and contains the text "bot: Dime que operación quieres que ejecute.". The second bubble is light gray and contains the text "user: crear usuario". The third bubble is light blue and contains the text "bot: Te ayudo en algo más?". At the bottom of the chat window, there is a light blue input field containing the text "usuarios" and a green button labeled "Enviar".

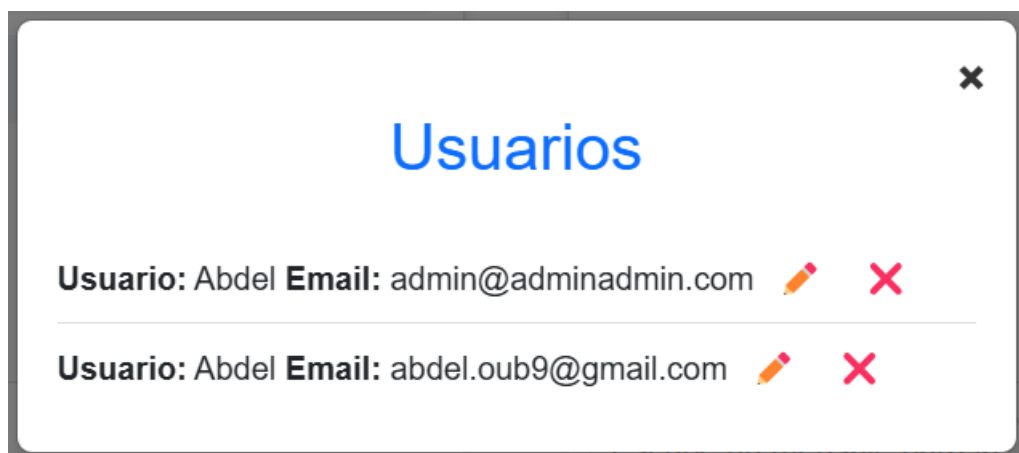
bot: Dime que operación quieres que ejecute.

user: crear usuario

bot: Te ayudo en algo más?

usuarios Enviar

El Bot muestra un mensaje que dice “Te ayudo en algo más?”. Para visualizar todos los usuarios, se le debe enviar un mensaje donde ponga “usuarios”.



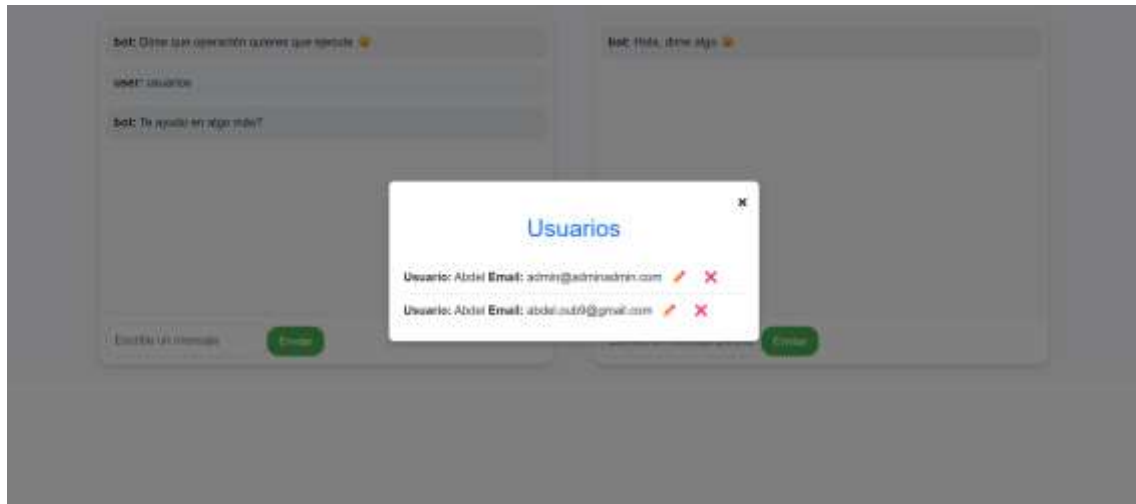
The screenshot shows a modal window with a white background and a gray border. It has a close button (X) in the top right corner. The title "Usuarios" is centered at the top in blue. Below the title, there is a list of users. Each user entry consists of the text "Usuario: Abdel" followed by the email address, a pencil icon for editing, and a red X icon for deletion. The first entry is "Email: admin@adminadmin.com" and the second is "Email: abdel.oub9@gmail.com".

Usuarios

Usuario: Abdel Email: admin@adminadmin.com ✎ ✕

Usuario: Abdel Email: abdel.oub9@gmail.com ✎ ✕

Y muestra los usuarios existentes. Como podemos ver, el usuario antes creado se muestra correctamente.

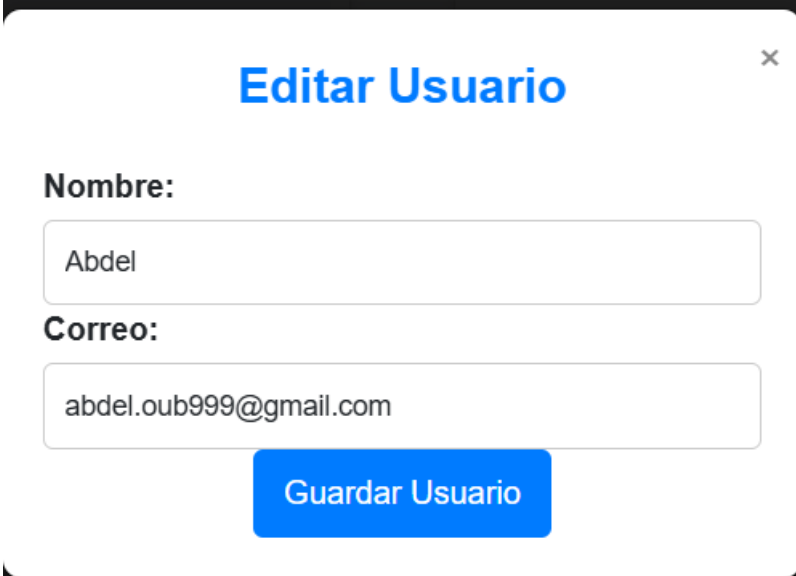


Aprovecho para mostrar el tamaño de los modales que se abren en la página.

Al hacer click en el lápiz se abrirá otro modal donde se podrán editar los datos de dicho usuario.

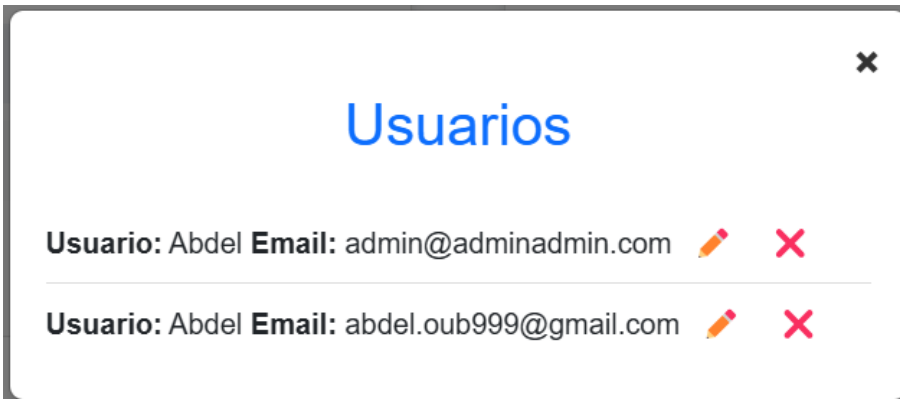
A screenshot of the "Editar Usuario" modal. It features a title "Editar Usuario" and a close button. Below the title, there are two form fields: "Nombre:" with the value "Abdel" and "Correo:" with the value "abdel.oub9@gmail.com". At the bottom of the modal is a blue button labeled "Guardar Usuario".

Verificamos que los datos del usuario aparecen correctamente en el formulario.



A modal window titled "Editar Usuario" with a close button (X) in the top right corner. It contains two input fields: "Nombre:" with the value "Abdel" and "Correo:" with the value "abdel.oub999@gmail.com". Below the fields is a blue button labeled "Guardar Usuario".

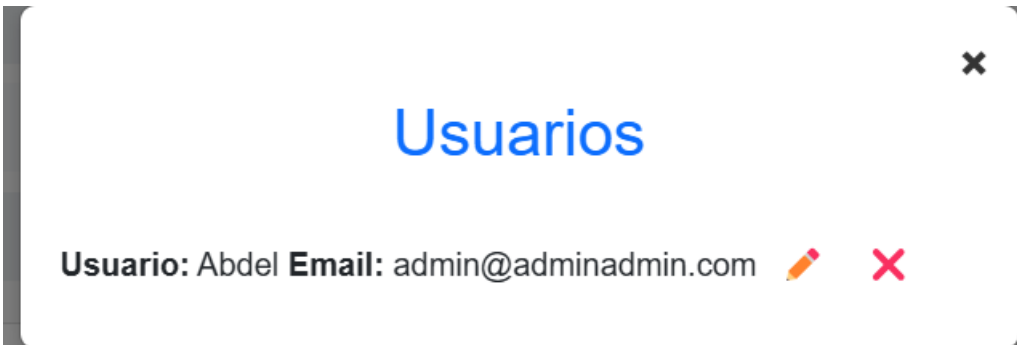
En este caso modifiko el email para comprobar que el formulario funciona correctamente.



A modal window titled "Usuarios" with a close button (X) in the top right corner. It displays a list of two users. Each user entry shows the username "Usuario: Abdel", the email "Email: admin@adminadmin.com" (for the first) and "Email: abdel.oub999@gmail.com" (for the second), followed by a pencil icon for editing and a red X icon for deletion.

Al cerrar el modal después de editar el usuario se refresca la lista de usuarios. Como podemos observar el usuario ha sido modificado correctamente.

Para eliminar el usuario se pulsa la X



A modal window titled "Usuarios" with a close button (X) in the top right corner. It displays a list of one user. The entry shows the username "Usuario: Abdel", the email "Email: admin@adminadmin.com", followed by a pencil icon for editing and a red X icon for deletion.

Y se verifica que el usuario es eliminado con éxito.

8 Conclusión

Este proyecto demuestra la implementación de una arquitectura moderna de microservicios utilizando Spring Boot, Docker, Keycloak y PostgreSQL, etc. Cada microservicio está diseñado para cumplir una función específica asegurando modularidad y mantenimiento eficiente. Además, se añaden pruebas unitarias, integración con herramientas de monitoreo y contenedores para asegurar un entorno eficiente.

El uso de Angular para el front, aunque sea sencillo, refleja mi compromiso con el aprendizaje continuo de frameworks modernos y la capacidad de integrarlos con sistemas backend.

Este proyecto refuerza mis habilidades técnicas y mi capacidad para desarrollar soluciones completas y bien documentadas, están listas para su implementación en entornos reales.