

|  |
| --- |
| DOCUMENTACIÓN fASE DE DISEÑO  PROYECTO FINAL CICLO |
| ALBERTO MOLEIRO SÁNCHEZ  I.E.S LÁZARO CARDENAS  CURSO 2022 / 2023 – 2º D.A.W |

## Índice

[Índice 1](#_Toc135941820)

[Objetivo 3](#_Toc135941821)

[Propósito del documento 3](#_Toc135941822)

[Alcance del proyecto 4](#_Toc135941823)

[Definiciones, acrónimos y abreviaturas 4](#_Toc135941824)

[Referencias 4](#_Toc135941825)

[Descripción General de la Arquitectura 5](#_Toc135941826)

[Contexto del sistema 5](#_Toc135941827)

[Objetivos y restricciones arquitectónicas 5](#_Toc135941828)

[Tecnologías seleccionadas 6](#_Toc135941829)

[Diseño de Microservicios 7](#_Toc135941830)

[Microservicio de gestión de usuarios 7](#_Toc135941831)

[Requisitos funcionales 7](#_Toc135941832)

[Diseño de la API 7](#_Toc135941833)

[Modelo de datos 8](#_Toc135941834)

[Microservicio de optimización de rutas 8](#_Toc135941835)

[Requisitos funcionales 8](#_Toc135941836)

[Diseño de la API 9](#_Toc135941837)

[Integración con Google Maps API 9](#_Toc135941838)

[Modelo de datos 9](#_Toc135941839)

[Microservicio de Gestión de Rutas 10](#_Toc135941840)

[Requisitos Funcionales 10](#_Toc135941841)

[Diseño de la API 10](#_Toc135941842)

[Modelo de Datos 11](#_Toc135941843)

[Microservicio de Autenticación 12](#_Toc135941844)

[Requisitos Funcionales 12](#_Toc135941845)

[Diseño de la API 12](#_Toc135941846)

[Modelo de Datos 13](#_Toc135941847)

[Implementación de JWT 13](#_Toc135941848)

[Microservicio de Registro de Empresas 14](#_Toc135941849)

[Requisitos Funcionales 14](#_Toc135941850)

[Diseño de la API 14](#_Toc135941851)

[Modelo de Datos 15](#_Toc135941852)

[Diseño de la Capa de Presentación (Frontend) 16](#_Toc135941853)

[Estructura de la aplicación Angular 16](#_Toc135941854)

[App 16](#_Toc135941855)

[Componentes y módulos principales 17](#_Toc135941856)

[Scafolding Aplicación 18](#_Toc135941857)

[Navegación y enrutamiento 19](#_Toc135941858)

[Integración con microservicios 19](#_Toc135941859)

[Diseño de la infraestructura 21](#_Toc135941860)

[API Gateway 21](#_Toc135941861)

[Funcionalidades 21](#_Toc135941862)

[Configuración 21](#_Toc135941863)

[Registro y descubrimiento de servicios 22](#_Toc135941864)

[Funcionalidades 22](#_Toc135941865)

[Configuración 22](#_Toc135941866)

[Seguridad y protección de datos 23](#_Toc135941867)

## Objetivo

Propósito del documento

El propósito de este documento es describir el diseño de la arquitectura de la aplicación **Lazy Postman**.

Esta documentación tiene como objetivo proporcionar a los desarrolladores, arquitectos y otros interesados una comprensión clara y detallada de la arquitectura propuesta, incluyendo los componentes clave, las tecnologías utilizadas y las decisiones de diseño que se han tomado.

#### Alcance del proyecto

El alcance de este proyecto incluye el diseño y desarrollo de una aplicación de gestión de usuarios y optimización de rutas, que se basa en una arquitectura de microservicios. La aplicación se compone de una capa de presentación implementada con Angular, microservicios desarrollados con Spring Boot y una base de datos PostgreSQL.

Las principales funcionalidades de la aplicación incluyen:

* Registro, autenticación y gestión de usuarios.
* Creación y optimización de rutas utilizando servicios de Google Maps.
* Visualización de rutas y puntos de interés en un mapa interactivo.

###### Definiciones, acrónimos y abreviaturas

* **API:** Application Programming Interface
* **CI/CD:** Integración Continua y Entrega Continua
* **REST:** Representational State Transfer
* **UI:** Interfaz de Usuario
* **UX:** Experiencia de Usuario

###### Referencias

* **Angular:** <https://angular.io/>
* **Spring Boot:** <https://spring.io/projects/spring-boot>
* **PostgreSQL:** <https://www.postgresql.org/>
* **Google Maps API:** <https://developers.google.com/maps>

## Descripción General de la Arquitectura

#### Contexto del sistema

Se trata de un sistema diseñado para facilitar la **administración de usuarios** y la **generación de rutas eficientes** utilizando los **servicios de Google Maps**.

La aplicación se basa en una **arquitectura de microservicios** y utiliza **Angular** para la capa de presentación, **Spring Boot** para el backend y **PostgreSQL** para la base de datos.

#### Objetivos y restricciones arquitectónicas

Los objetivos de la arquitectura propuesta para la aplicación son los siguientes:

* Facilitar la **separación de responsabilidades** y la **creación de módulos funcionales**.
* Permitir la **escalabilidad horizontal** de los diferentes componentes del sistema.
* Facilitar el desarrollo y despliegue independientes de cada componente.
* Integrarse con las tecnologías seleccionadas (Angular, Spring Boot y PostgreSQL).

Las **restricciones arquitectónicas** del proyecto incluyen:

* La capa de **presentación** debe ser desarrollada utilizando **Angular**.
* Los servicios **backend** deben ser implementados con **Spring Boot**.
* El sistema gestor de base de datos será **PostgreSQL**.
* La aplicación debe utilizar los **servicios de Google Maps** para la generación y optimización de rutas.

Tecnologías seleccionadas

Las tecnologías seleccionadas para la implementación de la aplicación son:

* **Capa de presentación:** Angular, un framework de desarrollo web que permite la creación de aplicaciones de una sola página (SPA) y ofrece una amplia gama de herramientas y funcionalidades para el desarrollo de interfaces de usuario interactivas y escalables.
* **Lógica de Negocio:** **Spring Boot**, un framework basado en Java que facilita la creación de **aplicaciones basadas en microservicios**, simplificando la configuración y la implementación de servicios RESTful. Spring Boot integra de manera nativa con Spring Security y Spring Data, lo que permite un manejo eficiente de la autenticación, autorización y acceso a datos.
* **Base de datos:** PostgreSQL, un sistema de gestión de bases de datos relacionales de código abierto que ofrece alta compatibilidad, rendimiento y escalabilidad. PostgreSQL es compatible con diversas características avanzadas, como transacciones y consultas complejas, lo que facilita el almacenamiento y la recuperación de información de los usuarios y las rutas.
* **Integración de servicios externos:** Google Maps API, que proporciona acceso a diversos servicios de mapas, como la generación de rutas, la geocodificación y la visualización de mapas interactivos. La aplicación utilizará estas API para optimizar las rutas y mostrar los resultados a los usuarios.

Diseño de Microservicios

La arquitectura de microservicios de la aplicación se compone de **dos microservicios principales**: el microservicio de **gestión de usuarios** y el microservicio de **optimización de rutas**, y varios **microservicios auxiliares** como: **autenticación** con JWT, **gestión de rutas** y **registro de empresas**.

Microservicio de gestión de usuarios

El microservicio de gestión de usuarios es responsable de las operaciones relacionadas con la autenticación, registro y administración de usuarios.

###### Requisitos funcionales

* **Administración de usuarios**: permitir a los administradores gestionar las cuentas de usuario (CRUD usuarios, cambiar roles, etc.).
* **Obtención de usuarios**: permite a los usuarios administradores obtener la lista de los usuarios de la empresa y a los usuarios con rol de responsable obtener la lista de usuarios a su cargo.
* **Asignación de rutas:** permite a los usuarios administradores o responsables asignar rutas a otros usuarios.
* **Actualización de contraseñas**: permitir a los usuarios actualizar su contraseña.

###### Diseño de la API

La API del microservicio de gestión de usuarios será RESTful e incluirá los siguientes puntos finales:

* **POST /users:** permite crear un usuario nuevo.
* **PUT /users/update:** permite a los usuarios actualizar su información personal.
* **PUT /users/update/password:** permite a los usuarios actualizar su contraseña.
* **GET / users/admin/{adminId}**: permite a los administradores obtener una lista de usuarios de su empresa.
* **GET / users/manager/{managerId}**: permite a los responsables obtener una lista de usuarios a su cargo.
* **DELETE /{userId}**: permite eliminar un usuario de la base de datos.

###### Modelo de datos

El modelado de la información se deja a cargo del desarrollo, pero tanto las peticiones como las respuestas sí deberán seguir estos modelos:

* **POST /users y PUT /users/update**

public class User {

private int idUser;

private String name;

private String lastname1;

private String lastname2;

private String phoneNumber;

private String login;

private Integer idRole;

}

* **PUT /users/update/password**

public class User {

private int idUser;

private String password;

private String newPassword;

* **GET / users/admin/{adminId} y GET / users/manager/{managerId}**:

[{

idUser: 1,

name: 'Juan',

lastName1: 'Perez',

lastName2: 'Gomez',

username: 'juanpg',

rol: '1'

},{…}]

#### Microservicio de optimización de rutas

El microservicio de optimización de rutas es responsable de generar y optimizar rutas basadas en los servicios de Google Maps y las preferencias del usuario.

###### Requisitos funcionales

* **Creación de rutas:** permitir a los usuarios crear rutas a partir de una serie de vías o tramos de vías.
* **Optimización de rutas**: calcular la ruta óptima entre los puntos especificados, utilizando los servicios de Google Maps y las preferencias del usuario (modo de transporte, restricciones de tiempo, etc.).
* **Devolver información sobre las localidades y calles**.

Diseño de la API

La API del microservicio de optimización de rutas será RESTful e incluirá los siguientes puntos finales:

* **POST /route/optimize**: permite a los usuarios solicitar la optimización de una ruta existente, utilizando sus preferencias y los servicios de Google Maps.
* **GET /route/towns**: permite a los usuarios obtener información detallada sobre las localidades de la base de datos.
* **GET /roads/{cdmuni}**: permite a los usuarios obtener información de un municipio pasado como parámetro.

###### Integración con Google Maps API

El microservicio de optimización de rutas se integrará con las API de Google Maps, incluyendo la API de Directions y la API de Geocoding. Estas API permitirán calcular rutas óptimas y obtener información geográfica.

* **API de Directions:** se utilizará para calcular rutas óptimas entre los puntos especificados, teniendo en cuenta las preferencias del usuario (modo de transporte, restricciones de tiempo, etc.).
* **API de Geocoding:** se utilizará para obtener información geográfica detallada sobre las direcciones y coordenadas proporcionadas por el usuario.

###### Modelo de datos

El modelado de la información se deja a cargo del desarrollo, pero tanto las peticiones como las respuestas sí deberán seguir estos modelos:

* **POST /route/optimize:**

[{

"province":"Madrid",

"town":{

"cdmuni":68,

"dsmuni":"Guadarrama"

},

"postCode":28440,

"roadType":"Calle",

"roadName":"ABEDUL",

"minOdd":1,

"maxOdd":11,

"minEven":2,

"maxEven":12

},{…}

]

* **GET /route/towns**:

public class Town {

private Integer cdmuni;

private String dsmuni;

}

* **GET /roads/{cdmuni}**:

public class Road {

private String name;

private int minOdd;

private int maxOdd;

private int minEven;

private int maxEven;

}

#### Microservicio de Gestión de Rutas

El microservicio de gestión de rutas es responsable de las operaciones relacionadas con la creación, asignación y administración de rutas.

###### Requisitos Funcionales.

* **Obtención de rutas**: permite a los usuarios obtener su lista de rutas disponibles y a los usuarios administradores o responsables obtener la lista de rutas asignadas a los usuarios.
* **Asignación de rutas**: permite a los usuarios administradores o responsables asignar rutas específicas a los empleados.
* **Administración de rutas:** permite a los administradores gestionar las rutas (CRUD rutas).
* **Devuelve información sobre las centrales de reparto.**

###### Diseño de la API

La API del microservicio de gestión de rutas será RESTful e incluirá los siguientes puntos finales:

* **GET /route-management/{routeId}:** permite obtener una ruta específica.
* **POST /route-management/create-route**: persiste la información de una nueva ruta.
* **PUT /route-management/update/{routeId}:** Actualiza la ruta.
* **GET /route-management /users-routes/{userId}**: permite a los usuarios obtener su lista de rutas disponibles.
* **POST /route-management /{routeId}/assign/{userId}:** permite a los administradores o responsables asignar una ruta específica a un usuario.
* **GET /route-management /manager/routes /{userId}**: Obtiene la lista de rutas de disponible para el usuario responsable, estas son las rutas propias y las de los empleados a su cargo.
* **GET /route-management/company/{companyId}:** Obtener la dirección de la central de reparto.

###### Modelo de Datos

El modelado de la información se deja a cargo del desarrollo, pero tanto las peticiones como las respuestas sí deberán seguir estos modelos:

* **GET /route-management/{routeId}:**

[

{

"lat":40.56784,

"lng":-3.13344

},

{...}

]

* **POST /route-management/create-route**:

{

"name":"Nombre Ruta",

"route":[

{

"lat":40.56784,

"lng":-3.13344

}],

"itinerary":[

{

"province":"Madrid",

"town":"Guadrrama",

"postCode":28440,

"roadType":"Calle",

"roadName":"Rio Arena",

"roadNumber":4

}

]

}

* **GET /route-management /users-routes/{userId}**:

[23,57,90,1,2] (Lista de identificadores de ruta)

* **GET /route-management /manager/routes /{userId}**:

[23,57,90,1,2] (Lista de identificadores de ruta)

* **GET /route-management/company/{companyId}:**

“Guadarrama 28440 Madrid Calle RIO ARENA 4”

#### Microservicio de Autenticación

El servicio encargado de la autenticación usando JWT.

###### Requisitos Funcionales

El microservicio de autenticación será responsable de **validar las credenciales** de los usuarios y **emitir tokens** de autenticación JWT. Además, este servicio proporcionará información sobre el **rol del usuario** para que se pueda llevar a cabo la autorización en otros servicios. Este servicio debe:

* Proveer un punto de acceso (endpoint) para la autenticación de usuarios, donde reciba las credenciales del usuario y, si son válidas, emita un token JWT.
* El token JWT debe contener información sobre la identidad del usuario y su rol.
* Proveer un endpoint para refrescar el token JWT, permitiendo a los usuarios obtener un nuevo token sin necesidad de autenticarse nuevamente.

###### Diseño de la API

El servicio de autenticación contendrá los siguientes endpoints:

* **POST /auth/login**: Este endpoint aceptará las credenciales del usuario (username y password) en el cuerpo de la petición, y si son válidas, retornará un token JWT.
  + **Body Request:** { "username": "user", "password": "pass" }
  + **Response:** { "token": "<JWT token>", "expires\_in": "<expiration time>" }
* **POST /auth/refresh**: Este endpoint aceptará un token JWT válido y retornará un nuevo token.
  + **Header Request:** Authorization: Bearer <token>
  + **Response:** { "token": "<JWT token>", "expires\_in": "<expiration time>" }

###### Modelo de Datos

Este servicio manejará principalmente la entidad **User** para autenticar usuarios. Sin embargo, la información específica del usuario estará en el servicio de gestión de usuarios. La entidad User puede tener la siguiente estructura:

public class User {

private String id; // Unique identifier for the user

private String username; // The user's username

private String password; // The user's hashed password

private Smallint role; // The user's role

// ... additional fields, constructors, getters and setters

}

El campo **role** es un Smallint que representa el rol del usuario. Este rol se incluirá en el token JWT para su uso en la autorización en otros servicios. La contraseña debe ser almacenada como un hash, no como texto plano, para la seguridad del sistema.

###### Implementación de JWT

El JWT generado contendrá una claim "sub" (subject) que almacenará el identificador del usuario, una claim "rol" que indicará el rol del usuario y una claim de expiración para definir cuándo caduca el token. Se usará un algoritmo de firma para garantizar la autenticidad del token (como HS256 o RS256).

Finalmente, es importante notar que este servicio debe manejar los errores adecuadamente, como las credenciales inválidas y los tokens caducados, y responder con los códigos de estado HTTP apropiados.

#### Microservicio de Registro de Empresas

El microservicio de registro de empresas es responsable de permitir el registro de nuevas empresas por parte de usuarios anónimos, seguido de una confirmación vía correo electrónico.

###### Requisitos Funcionales

* + **Registro de empresas**: permite a los usuarios anónimos registrar nuevas empresas. El proceso de registro implica proporcionar detalles de la empresa y un correo electrónico para confirmar la inscripción.
  + **Confirmación de registro:** permite a los usuarios confirmar el registro de su empresa a través de un enlace enviado a su correo electrónico.

###### Diseño de la API

La API del microservicio de registro de empresas será RESTful e incluirá los siguientes puntos finales:

* + **POST /company**: permite a los usuarios anónimos registrar una nueva empresa. Los datos necesarios incluirán los detalles de la empresa y un correo electrónico para confirmar el registro.
  + **GET /company/confirm/{confirmationCode}:** permite a los usuarios confirmar el registro de su empresa a través de un código de confirmación enviado a su correo electrónico.

###### Modelo de Datos

El modelo de datos del microservicio de registro de empresas incluirá la siguiente entidad:

* **Empresa**: contiene información sobre las empresas en proceso de registro, incluyendo un identificador único, nombre de la empresa, dirección, detalles de contacto, y estado de confirmación.

La entidad Empresa podría tener la siguiente estructura:

public class Company {

private Integer id;

private String cif;

private String businessName;

private String phoneNumber;

private String email;

private Integer idTown;

private String address;

}

Al igual que en otros microservicios, es importante manejar adecuadamente los errores y responder con los códigos de estado HTTP apropiados.

Diseño de la Capa de Presentación (Frontend)

La capa de presentación de la aplicación se desarrollará utilizando Angular, un popular framework para la construcción de aplicaciones web de una sola página (SPA) y escalables.

#### Estructura de la aplicación Angular

La estructura de la aplicación Angular se organizará en módulos y componentes de acuerdo con las mejores prácticas y directrices de Angular. La estructura de directorios incluirá:

###### App

Este es el módulo principal de la aplicación y es el punto de entrada de la misma. Se compone de varios submódulos y componentes, tales como:

* **Registered:** Este es un submódulo de la aplicación que contiene varios componentes y módulos relacionados con los usuarios registrados. Incluye elementos como componente **Home**, módulo de **Optimized-routes** y módulo de **UsersManagement**. Cada componente tiene su archivo CSS, HTML y TS. También cuenta con su propio módulo de enrutamiento para gestionar la navegación entre estos componentes.
* **Core:** Este módulo contiene la funcionalidad principal de la aplicación que se necesita en todo el proyecto. Incluye modelos, servicios compartidos, guards, interceptores y pipes. También tiene un módulo compartido que contiene componentes, directivas y pipes que se utilizan en varias partes de la aplicación.
* **Public:** Este módulo contiene páginas públicas como Login y Register. Estas páginas son accesibles sin autenticación. Cada componente tiene su propio archivo CSS, HTML y TS

#### Componentes y módulos principales

La aplicación Angular se dividirá en varios componentes y módulos principales que representan las diferentes áreas y funcionalidades de la aplicación:

* **AppModule:** módulo principal de la aplicación que define la estructura y las dependencias básicas.
* **PublicModule:** módulo que agrupa las vistas públicas de la aplicación.
* **LoginComponent:** componente para el inicio de sesión y autenticación de usuarios.
* **RegisterComponent:** componente para el registro de nuevos usuarios.
* **RegisteredModule:** módulo que agrupa las vistas y componentes accesibles solo para usuarios registrados.
* **UsersModule:** módulo que agrupa los componentes y servicios relacionados con la gestión de usuarios.
* **UserManagementComponent:** Componente para la gestión de usuarios, solo accesible para usuarios con roles de responsabilidad.
* **UserConfigurationComponent:** componente para mostrar información del usuario y proporcionar una vía para el cambio de contraseña.
* **OptimizedRoutesModule:** módulo que agrupa los componentes y servicios relacionados con la optimización de rutas.
* **RouteCreatorComponent:** componente para la creación de rutas y la especificación de puntos de inicio, destino y puntos intermedios.
* **RouteViewComponent:** componente para la visualización de rutas utilizando los servicios de Google Maps.
* **ItineraryComponent:** componente para la visualización de rutas y puntos de interés en un mapa interactivo.

#### Scafolding Aplicación

* **App**
  + **Registered**
    - **Home**
      * **Components**
      * **Containers**
        + Home.component.css
        + Home.component.html
        + Home.component.ts
    - **Optimized-routes**
      * **Itinerary**
      * **Route-creator**
      * **Route-view**
      * Optimimized-routes-routing.module.ts
      * Optimimized-routes.module.ts
    - **Users**
      * **User-settings**
      * **Users-management**
      * Users-routing.module.ts
      * Users.module.ts
    - Registered-routing.module.ts
    - Registered.component.css
    - Registered.component.html
    - Registered.component.ts
    - Registered.module.ts
  + **Core**
    - **Models**
    - **Services**
    - **Shared**
      * **Components**
      * **Guards**
      * **Interceptors**
      * **Pipes**
      * Shared.module.ts
    - Core.module.ts
  + **Public**
    - **Login**
    - **Register**
    - Public-routing.module.ts
    - Public.component.css
    - Public.component.html
    - Public.component.ts
  + App-routing.module.ts
  + App.component.css
  + App.component.html
  + App.component.ts
  + App.module.ts

#### Navegación y enrutamiento

La navegación y el enrutamiento de la aplicación se manejarán utilizando el **RouterModule** de Angular. Se definirán rutas para cada uno de los componentes principales y se protegerán las rutas que requieren autenticación utilizando guardianes (**AuthGuard**).

Algunas de las rutas principales incluirán:

* **/:** ruta para el *LoginComponent*
* **/login:** ruta para el *LoginComponent*.
* **/register:** ruta para el *RegisterComponent*.
* **/registered:** ruta protegida por AuthGuard, para la parte de usuarios registrados.
* **/registered/home:** ruta para el *HomeComponent*.
* **/registered/route/creator:** ruta para el *RouteCreatorComponent*.
* **/registered/route/view:** ruta para el *RouteViewComponent*.
* **/registered/routes/itinerary:** ruta para el *ItineraryComponent*.
* **/registered/users/management:** ruta para el *UsersManagementComponent*.
* **/registered/users/settings:** ruta para el *UserSettingsComponent*.

#### Integración con microservicios

La capa de presentación se **comunicará con los microservicios** a través de **servicios de Angular** que encapsulan las llamadas a la **API RESTful**. Estos servicios utilizarán el **HttpClientModule** de Angular para realizar solicitudes HTTP a los puntos finales de la API y gestionarán las respuestas y errores de la API de manera adecuada.

Se crearán servicios específicos para cada microservicio, como UserService para la gestión de usuarios y RouteService para la optimización de rutas. Algunos de los métodos principales de estos servicios incluirán:

* **UserService**:
  + **createUser(userData)**: Crea un usuario nuevo (solo administradores).
  + **updateProfile(userData):** actualiza la información del perfil del usuario.
  + **deleteAccount():** elimina la cuenta del usuario.
  + **getUsers():** obtiene una lista de usuarios para administradores.
* **RegisterService:**
  + **register(userData):** registra un nuevo usuario en el sistema.
* **AuthService**
  + **login(credentials):** autentica al usuario y obtiene un token de acceso.
  + **logout():** Borra la sesión de un usuario
  + **checkLogin():** Comprueba si un usuario tiene una sesión activa.
* **RouteService**:
  + **createRoute(routeData):** crea una nueva ruta optimizada con puntos de inicio, destino y puntos intermedios opcionales.
  + **getRoute(routeId):** obtiene información detallada sobre una ruta específica.
  + **getItinerary(routeId):** obtiene el itinerario de una ruta.

## Diseño de la infraestructura

La infraestructura de la aplicación se diseñará para garantizar la escalabilidad, la disponibilidad y la seguridad de los microservicios y la base de datos.

#### API Gateway

El API Gateway actuará como un punto de entrada único para las solicitudes de los clientes a los microservicios, facilitando la gestión de solicitudes, la seguridad y el monitoreo.

###### Funcionalidades

* **Enrutamiento de solicitudes:** dirigir las solicitudes de los clientes al microservicio adecuado en función de la ruta y el método HTTP.
* **Autenticación y autorización:** validar los tokens de acceso (JWT) y asegurar que las solicitudes estén autorizadas antes de dirigirlas a los microservicios.
* **Limitación de la tasa de solicitudes (rate limiting):** proteger los microservicios de la sobrecarga y los ataques de fuerza bruta limitando la cantidad de solicitudes por unidad de tiempo.
* **Monitoreo y registro:** registrar las solicitudes y las respuestas para facilitar la supervisión y el análisis del rendimiento de la API.

###### Configuración

El API Gateway se configurará utilizando **Spring Cloud Gateway** en un ambiente local. La configuración incluirá la definición de rutas, reglas de autenticación y autorización, y políticas de limitación de la tasa de solicitudes.

#### Registro y descubrimiento de servicios

El registro y descubrimiento de servicios permitirán a los microservicios descubrir y comunicarse entre sí de manera dinámica y resiliente en un ambiente local.

###### Funcionalidades

* **Registro de microservicios:** permitir que los microservicios se registren en un servidor de descubrimiento centralizado al iniciarse.
* **Descubrimiento de microservicios:** permitir que los microservicios localicen y se comuniquen con otros microservicios a través del servidor de descubrimiento.
* **Resiliencia:** garantizar que los microservicios puedan descubrir y comunicarse entre sí incluso en caso de fallos en el servidor de descubrimiento o cambios en la infraestructura.

###### Configuración

El registro y descubrimiento de servicios se configurarán utilizando Spring Cloud Discovery con Eureka para el ambiente local. La configuración incluirá la definición de los servidores de descubrimiento y las propiedades de registro y descubrimiento de los microservicios.

Para ello, en cada microservicio, deberá configurarse la dependencia de Eureka Client en el archivo pom.xml y configurar las propiedades de Eureka en el archivo de propiedades de Spring. Adicionalmente, deberá configurarse una instancia de Eureka Server que actuará como el servidor de descubrimiento centralizado.

#### Seguridad y protección de datos

La seguridad y la protección de los datos de los usuarios y las rutas serán una prioridad en el diseño de la infraestructura. Algunas de las medidas de seguridad que se implementarán incluyen:

* **Comunicación cifrada:** utilizar HTTPS y TLS para cifrar la comunicación entre el cliente, el API Gateway y los microservicios.
* **Almacenamiento seguro de contraseñas:** almacenar las contraseñas de los usuarios utilizando algoritmos de hash seguros.
* **Autenticación y autorización basadas en tokens:** utilizar tokens JWT para autenticar a los usuarios y garantizar que solo tengan acceso a los recursos autorizados.
* **Protección contra ataques comunes:** proteger los microservicios y la base de datos contra ataques como inyección SQL, CSRF y XSS mediante la validación y saneamiento de datos, el uso de políticas de seguridad adecuadas y la aplicación de las mejores prácticas de desarrollo seguro.