ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

Grado en Ingeniería Informática

Diseño y programación de una aplicación web para la gestión personal

Autor: Cristian Emilov Georgiev
Tutor: Borja Bordel Sánchez
Curso Académico: 2024/2025

Madrid, 30 de junio de 2025

Resumen

Este Trabajo de Fin de Grado tiene como objetivo el diseño y desarrollo de una aplicación web orientada a la gestión y seguimiento de metas personales. La aplicación está pensada para que los usuarios puedan introducir, organizar y monitorizar sus propios objetivos, ya sean de tipo profesional, académico, personal o de salud. Cada meta introducida por el usuario debe estar asociada a un valor cuantitativo que permita realizar un seguimiento objetivo de su evolución en el tiempo.

El sistema permite registrar actualizaciones sobre el estado de cada meta, lo que facilita al usuario evaluar su progreso. Para ello, se ha implementado un panel de control visual que ofrece una representacion gráfica clara y dinámica del cumplimiento de las metas definidas. Esta interfaz mejora la experiencia del usuario y promueve la constancia en el logro de sus objetivos.

Desde el punto de vista técnico, la aplicación se ha desarrollado utilizando una arquitectura basada en microservicios, lo cual favorece la escalabilidad, mantenibilidad y despliegue independiente de los distintos módulos del sistema. Además, el entorno de despliegue se basa en tecnologías cloud, lo que garantiza accesibilidad, disponibilidad y flexibilidad en el uso de los recursos.

El proyecto combina así aspectos de desarrollo frontend y backend, arquitectura distribuida, servicios en la nube y experiencia de usuario, con el propósito de ofrecer una herramienta práctica y eficaz para la mejora de la productividad personal.

Abstract

This Final Degree Project aims to design and develop a web application focused on the management and monitoring of personal goals. The application is intended to allow users to input, organize, and track their own objectives, whether they are professional, academic, personal, or health-related. Each goal entered by the user must be associated with a quantitative value that enables objective tracking of its progress over time.

The system allows users to record updates on the status of each goal, facilitating the evaluation of their progress. To support this, a visual dashboard has been implemented, offering clear and dynamic graphical representation of the goals fulfillment. This interface enhances the user experience and encourages consistency in achieving objectives.

From a technical perspective, the application has been developed using a microservices-based architecture, which promotes scalability, maintainability, and independent deployment of the different system modules. In addition, the deployment environment is based on cloud technologies, ensuring accessibility, availability, and flexibility in resource usage.

The project thus combines aspects of frontend and backend development, distributed architecture, cloud services, and user experience, with the goal of providing a practical and effective tool for improving personal productivity.

Índice

1. Introducción

1.1. Problemática

En la sociedad actual, muchas personas encuentran dificultades para mantener la constancia y el control en el cumplimiento de sus metas personales. Ya sea en el ámbito profesional, académico, de salud o desarrollo personal, establecer objetivos es habitual, no obstante, el verdadero reto reside en darles seguimiento, medir el progreso de forma objetiva y mantener la motivación a lo largo del tiempo. La ausencia de herramientas simples y efectivas suele derivar en desorganización, pérdida de enfoque y abandono de los objetivos.

Este proyecto aborda esta problemática mediante el desarrollo de una aplicación web básica pero funcional, centrada en la gestión de metas personales. La solución permite a los usuarios registrar objetivos personalizados, asociarles métricas cuantitativas y realizar actualizaciones periódicas. Además, incorpora un panel visual que facilita la comprensión del progreso y refuerza el compromiso del usuario.

A nivel técnico, la aplicación se estructura en torno a una arquitectura de microservicios y se despliega en un entorno cloud utilizando Kubernetes, lo que asegura escalabilidad y accesibilidad. Así, el proyecto propone una solución sencilla pero bien organizada, alineada con buenas prácticas de desarrollo y centrada en facilitar la mejora de la productividad personal.

1.2. Objetivos

Este proyecto tiene de objetivo desarrollar una aplicación web que permita a los usuarios gestionar y monitorizar sus metas personales de forma cuantificable, visual y accesible, utilizando una arquitectura moderna basada en microservicios desplegada en un entorno cloud.

Para lograr este propósito general, se establecen los siguientes objetivos específicos:

- Diseñar una interfaz web sencilla e intuitiva que permita al usuario registrar y organizar sus metas personales.
- Establecer un modelo de datos que permita representar objetivos con métricas cuantificables y actualizaciones periódicas.
- Implementar funcionalidades CRUD (crear, leer, actualizar y eliminar) para gestionar las metas de cada usuario.
- Desarrollar un panel de control visual que muestre el progreso de cada objetivo mediante representaciones gráficas.
- Aplicar una arquitectura basada en microservicios para separar los distintos módulos funcionales de la aplicación.
- Realizar el despliegue de la aplicación en un entorno cloud, utilizando herramientas como contenedores y orquestación con Kubernetes.

2. Estado del arte

2.1. Herramientas de Desarrollo

Durante el desarrollo del proyecto se han utilizado diversas herramientas que han facilitado tanto la programación como la gestión, pruebas y despliegue de los distintos componentes del sistema. A continuación, se describen brevemente las principales herramientas utilizadas:

- **IntelliJ IDEA:** Entorno de desarrollo integrado (IDE) utilizado principalmente para la programación de los microservicios desarrollados en Java. Ofrece soporte avanzado para proyectos Maven, Spring Boot, depuración y gestión de dependencias.
- **PyCharm:** IDE especializado en desarrollo en Python. Se utilizó para la creación y mantenimiento de componentes o scripts auxiliares escritos en dicho lenguaje. PyCharm proporciona herramientas de análisis de código, gestión de entornos virtuales y ejecución controlada de scripts.
- **GitHub:** Plataforma de control de versiones basada en Git, empleada para gestionar el código fuente del proyecto. La principal ventaja de su uso es la sencillez que otorga a procesos automácticos en el entorno de despliegue/testing en kubernetes.
- **Postman:** Herramienta de pruebas para APIs REST. Se utilizó para realizar pruebas manuales de los endpoints expuestos por los microservicios, verificar respuestas, gestionar colecciones de peticiones y simular distintos escenarios de uso.
- **WSL (Windows Subsystem for Linux):** Entorno de ejecución de GNU/Linux dentro de Windows. Fue empleado para trabajar con herramientas de línea de comandos y realizar configuraciones relacionadas con Docker, Kubernetes y scripts de automatización en un entorno Unix-like.

2.2. Tecnologías Utilizadas en el Backend

El desarrollo del backend del sistema se ha llevado a cabo utilizando un conjunto de tecnologías modernas centradas en el ecosistema de Spring, orientadas a facilitar la creación de APIs REST seguras y escalables. A continuación se describen las principales tecnologías empleadas:

- **Spring Boot[1]:** Framework que simplifica la creación de aplicaciones basadas en Spring mediante configuración automática, integración con herramientas externas y un modelo de desarrollo centrado en la creación de microservicios. Permite exponer de manera rápida y eficiente endpoints RESTful.
- **Spring Security[2]:** Módulo del ecosistema Spring utilizado para implementar mecanismos de autenticación y autorización en la aplicación. Permite restringir el acceso a los recursos según el rol del usuario y proteger los endpoints de manera estructurada y configurable.
- **MySQL Connector:** Librería que permite la comunicación entre la aplicación desarrollada en Java y una base de datos MySQL. Se utiliza para establecer la conexión, ejecutar consultas y gestionar las operaciones CRUD de forma eficiente.
- **JWT (JSON Web Token[3]):** Tecnología empleada para la gestión de sesiones y autenticación basada en tokens. En conjunto con Spring Security, permite emitir y validar tokens firmados que autentican al usuario de forma segura en cada petición.

2.3. Tecnologías Utilizadas en el Frontend

El desarrollo del frontend se ha realizado utilizando tecnologías ligeras y flexibles, adecuadas para una aplicación web centrada en la visualización y actualización de datos. A continuación se detallan las principales herramientas empleadas:

- **Flask[4]:** Microframework web escrito en Python que se ha utilizado como base para el desarrollo del frontend. Flask incluye el motor de plantillas Jinja2, lo cual permite generar páginas HTML dinámicas a partir de datos proporcionados por el backend. Su diseño minimalista lo hace más adecuado para aplicaciones ligeras, en contraste con frameworks más complejos como Django.
- **Plotly[5]:** Librería de visualización de datos que permite generar gráficos interactivos y dinámicos directamente en la web. En este proyecto, se ha utilizado para representar de forma visual el progreso de las metas personales del usuario a lo largo del tiempo.
- **Gunicorn[6]:** Servidor WSGI utilizado para desplegar la aplicación Flask en producción. Gunicorn permite manejar múltiples peticiones concurrentes y mejora el rendimiento del servidor en entornos reales, actuando como intermediario entre Flask y el servidor web.
- **Bootstrap[7]:** Framework de diseño web basado en HTML, CSS y JavaScript, utilizado para construir una interfaz limpia, responsiva y fácil de usar. Su integración con las plantillas Jinja ha permitido mantener una experiencia de usuario coherente sin necesidad de desarrollar estilos desde cero.

2.4. Tecnologías de Infraestructura y Persistencia de Datos

Además del desarrollo de la lógica de negocio y la interfaz de usuario, el proyecto ha requerido el uso de tecnologías orientadas a la persistencia de datos, la gestión de contenedores y el despliegue de los servicios en un entorno distribuido. A continuación se describen los principales componentes utilizados en este ámbito:

- **MySQL:** Sistema de gestión de bases de datos relacional (RDBMS) utilizado para almacenar de forma estructurada la información relacionada con usuarios, metas y actualizaciones de progreso. MySQL ofrece un equilibrio entre rendimiento, fiabilidad y facilidad de integración con aplicaciones basadas en Java.
- **Docker:** Plataforma de contenedorización utilizada para empaquetar los distintos componentes del sistema (backend, frontend, base de datos) junto con sus dependencias. Esto permite garantizar entornos de ejecución consistentes y facilita tanto el desarrollo como el despliegue.
- **Kubernetes:** Sistema de orquestación de contenedores que permite gestionar, escalar y desplegar automáticamente los distintos servicios del proyecto. Gracias a Kubernetes, es posible coordinar el funcionamiento de los microservicios, asegurar su disponibilidad y simplificar la gestión del entorno cloud.

3. Arquitectura y Diseño

3.1. Diagrama de casos de uso

La aplicación `Goal Tracker` contempla dos tipos principales de usuarios: el **usuario normal** y el **usuario administrador**, cada uno con diferentes niveles de acceso y funcionalidades disponibles. A continuación se presenta el diagrama de casos de uso, el cual resume las principales interacciones que estos actores pueden tener con el sistema.

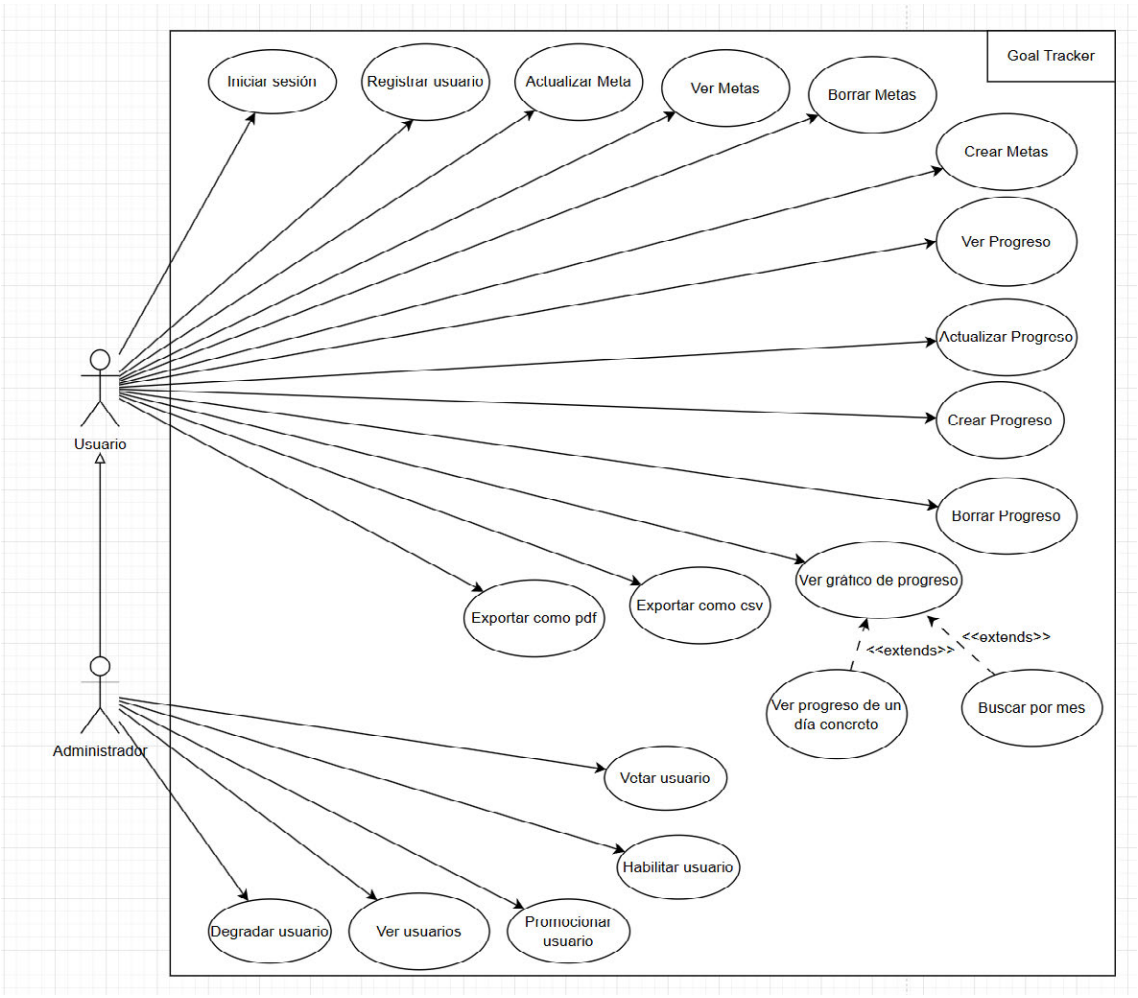


Figura 1: Diagrama de casos de uso de la aplicación `Goal Tracker`.

3.2. Arquitectura

La aplicación *Goal Tracker* está diseñada utilizando una arquitectura basada en microservicios, desplegada sobre un clúster de Kubernetes. Esta estructura permite una alta escalabilidad, aislamiento de responsabilidades y facilidad de mantenimiento.

Cada microservicio cumple un rol bien definido dentro del sistema, y la comunicación entre ellos se gestiona a través de rutas HTTP, coordinadas por un **Ingress Controller** (implementado con NGINX). Este componente actúa como punto de entrada único, redirigiendo las solicitudes entrantes según la ruta a los servicios correspondientes.

A continuación se describen los componentes principales del sistema:

- **Frontend Service:** desarrollado con Flask y Jinja, se encarga de renderizar las vistas HTML y manejar la interacción con el usuario. No contiene lógica de negocio, sino que actúa como puente entre el navegador y el backend.
- **Backend API:** implementado en Spring Boot (Java), centraliza toda la lógica de negocio y el acceso a la base de datos. Expone una API RESTful que permite la gestión de usuarios, metas y entradas de progreso. Este servicio también atiende solicitudes provenientes de otros microservicios, como los de gráficos y exportación, debido a que es el que gestiona la autenticación.
- **Graphing Service:** microservicio en Flask que utiliza Plotly para generar visualizaciones del progreso del usuario. Obtiene los datos desde el backend y los transforma en gráficos mensuales accesibles desde la interfaz web.
- **Exporting Service:** también basado en Flask, permite la exportación de datos del usuario en formatos CSV y PDF. Al igual que el servicio de gráficos, accede a los datos a través del Backend API.
- **Base de Datos (MySQL):** almacena de forma persistente toda la información del sistema, incluyendo usuarios, metas y entradas de progreso. Únicamente el Backend API tiene acceso directo a esta base de datos.
- **Ingress Controller:** mediante NGINX, enruta las peticiones entrantes a los microservicios correctos, según patrones de ruta definidos. Esto permite un control centralizado del tráfico y un desacoplamiento entre cliente y servicios internos.

Todo el sistema se ejecuta dentro de un entorno orquestado por Kubernetes, lo que permite escalar cada microservicio de forma independiente, realizar despliegues sin interrupciones, realizar actualizaciones de cada servicio de manera independiente y asegurar una alta disponibilidad. La arquitectura propuesta refleja buenas prácticas modernas en el desarrollo de sistemas distribuidos, priorizando la modularidad, la claridad en las responsabilidades y la preparación para producción.

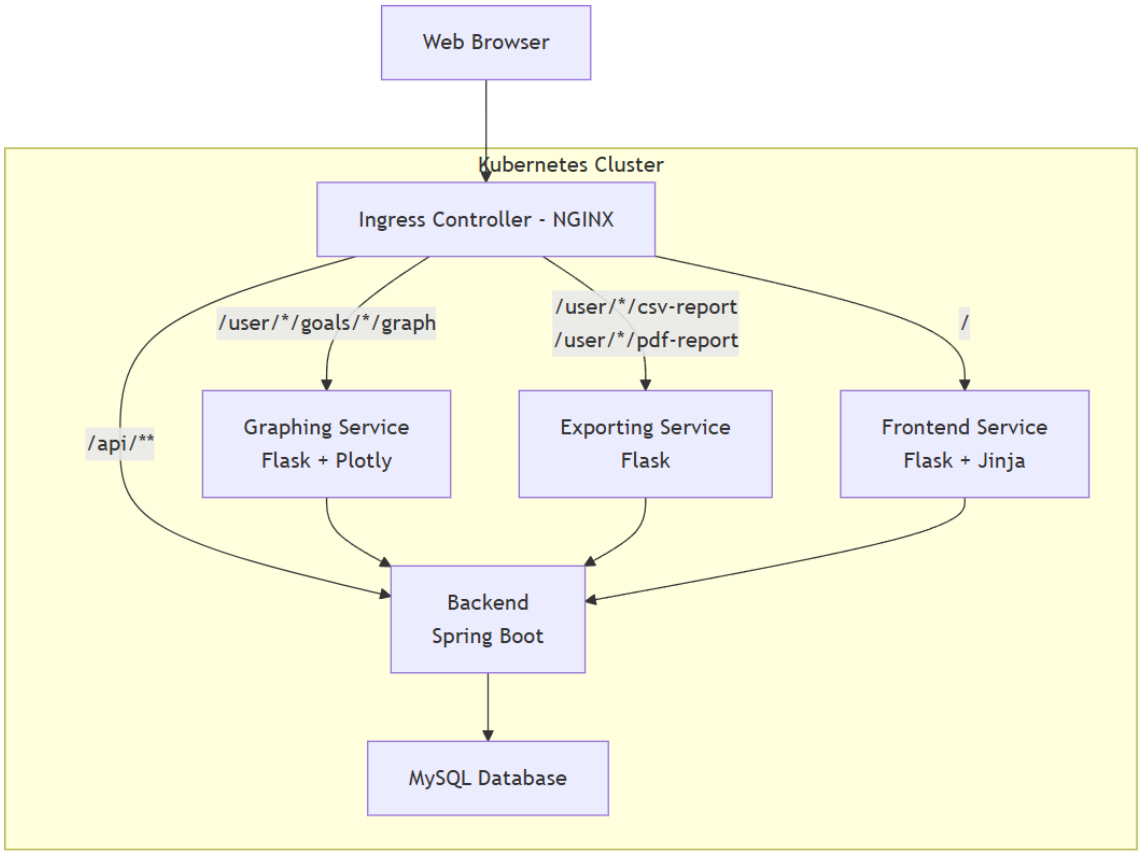


Figura 2: Diagrama de la arquitectura de la aplicación Goal Tracker, mostrando los microservicios desplegados en Kubernetes y el flujo de comunicación entre ellos.

3.3. Autenticación

La aplicación implementa un sistema de autenticación basado en JSON Web Tokens (JWT), utilizando dos tipos de tokens para gestionar la seguridad y el acceso de forma eficiente y escalable:

- **Access Token (JWT):** token que se utiliza para autorizar las solicitudes a la API, proporcionando acceso a los recursos protegidos mientras el token sea válido. Tiene un tiempo de vida de 1 hora.
- **Refresh Token:** token empleado para obtener nuevos access tokens sin que el usuario tenga que autenticarse nuevamente mediante credenciales. Esto permite mantener sesiones activas de manera segura y cómoda para el usuario. Este token tiene un tiempo de vida de 1 semana.

La principal ventaja de utilizar JWT como mecanismo de autenticación es que permite implementar un sistema "stateless" en el backend. Esto significa que el servidor no necesita almacenar información de sesión, ya que toda la información relevante viaja en el propio token. Como resultado, es posible desplegar múltiples instancias del servicio backend de manera paralela, facilitando la escalabilidad horizontal.

La única parte "stateful" del sistema de autenticación es la gestión del refresh token, que se almacena en la base de datos para validar su validez y evitar abusos. Esta estrategia ofrece un equilibrio razonable entre seguridad, simplicidad y escalabilidad.

Actualmente, la implementación no realiza rotación (refresh) del refresh token en sí mismo, lo cual es una práctica común para simplificar el diseño. Esto implica que los refresh tokens pueden tener una vida útil más larga de lo ideal, pero resulta adecuado para el alcance y los objetivos del proyecto, que buscan demostrar los fundamentos de seguridad sin la complejidad de un sistema completamente endurecido para producción.

3.4. Base de datos

La base de datos utilizada en *Goal Tracker* es **MySQL**, un sistema de gestión de bases de datos relacional ampliamente adoptado por su rendimiento, fiabilidad y facilidad de integración. MySQL permite estructurar los datos mediante tablas relacionales, lo cual es ideal para representar entidades como usuarios, metas y entradas de progreso con relaciones bien definidas entre ellas.

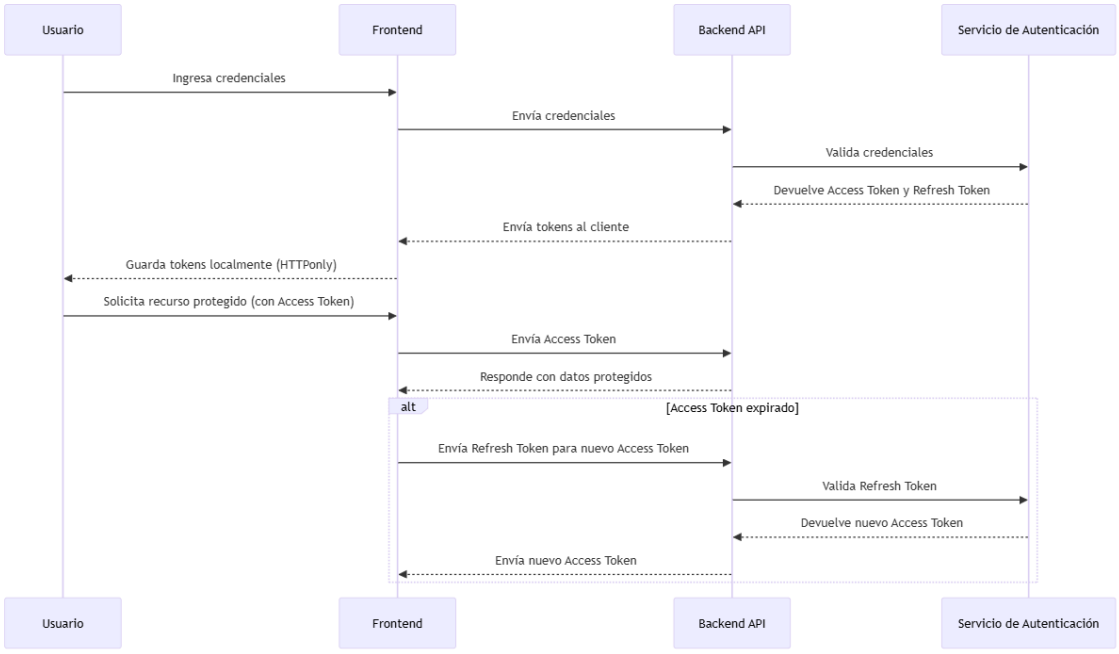


Figura 3: Diagrama de flujo de la autenticación en Goal Tracker.

Ventajas de utilizar MySQL:

- Es un sistema robusto y probado en entornos de producción a gran escala.
- Tiene soporte completo para el lenguaje SQL, lo que facilita consultas complejas y relaciones entre entidades.
- Ofrece buenas capacidades de escalabilidad y rendimiento para aplicaciones web modernas.
- Cuenta con soporte nativo en la mayoría de frameworks y lenguajes de programación, incluyendo Java (Spring Boot), con el que se realizará la conexión mediante el conector oficial (MySQL Connector).
- Aunque existen diferencias entre el paradigma orientado a objetos y el modelo relaciona, para aplicaciones sencillas resulta conveniente utilizar un ORM como Hibernate, lo que simplifica significativamente el diseño y desarrollo.

Propiedades ACID:

MySQL garantiza el cumplimiento de las propiedades **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad), que son fundamentales en aplicaciones donde se gestionan datos sensibles y consistentes como:

- **Atomicidad:** asegura que las operaciones dentro de una transacción se completen completamente o no se realicen en absoluto. Esto es crítico, por ejemplo, al registrar una entrada de progreso junto con una nota, ambas deben guardarse juntas.
- **Consistencia:** garantiza que la base de datos pase de un estado válido a otro igualmente válido. Se evita así que operaciones inválidas dejen datos corruptos.
- **Aislamiento:** previene que transacciones concurrentes interfieran entre sí, manteniendo coherencia incluso cuando varios usuarios actualizan sus metas simultáneamente.
- **Durabilidad:** asegura que los datos confirmados persisten incluso ante fallos del sistema, lo que es indispensable para retener información de usuario como su email, nombre o metas registradas.

En el contexto de *Goal Tracker*, estas propiedades son especialmente importantes porque se maneja información personal identificable, como correos electrónicos y nombres de usuario, además del progreso personalizado de cada meta. Usar MySQL con soporte transaccional permite mantener la integridad y confiabilidad de estos datos a lo largo del tiempo y frente a posibles errores o caídas del sistema.

A continuación se muestran las tres entidades junto con su explicación:

3.4.1. User

La entidad **User** representa a cada persona registrada dentro del sistema *Goal Tracker*. Es una de las entidades principales de la base de datos y actúa como el punto de entrada para todas las funcionalidades personalizadas de la aplicación, incluyendo la gestión de metas y el control de acceso.

Cada usuario tiene una serie de atributos fundamentales:

- `id`: identificador único del usuario, generado automáticamente.
- `name`: nombre del usuario, requerido para la personalización de la experiencia.
- `email`: dirección de correo electrónico, única y obligatoria. Se utiliza como identificador principal en el login.
- `password`: contraseña cifrada del usuario. Será encriptada usando bcrypt.

- `refreshToken` y `refreshTokenExpiry`: usados para gestionar la autenticación basada en JWT, permitiendo sesiones persistentes y seguras.
- `isAdmin` y `banned`: indican si un usuario tiene privilegios administrativos o si se encuentra bloqueado por un administrador.
- `goals`: la lista de metas asociadas a un usuario indivudual.

```java
@Entity
public class User {
    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    2 usages
    @Column(nullable = false)
    private String name;

    2 usages
    @Column(nullable = false)
    private String password;

    2 usages
    @Column(nullable = false, unique = true)
    private String email;

    2 usages
    @Column(length = 500, nullable = false, unique = true)
    private String refreshToken;

    2 usages
    @Column(nullable = false)
    private LocalDateTime refreshTokenExpiry;


    3 usages
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Goal> goals = new ArrayList<>();

    2 usages
    private boolean isAdmin = false;

    2 usages
    private boolean banned = false;
```

Figura 4: Modelo entidad-relación de la clase `User`.

3.4.2. Goals

La entidad **Goal** representa una meta personal definida por el usuario dentro del sistema. Es una parte central de la lógica del dominio de la aplicación *Goal Tracker*, ya que permite estructurar y monitorear los objetivos individuales de cada usuario.

Cada instancia de `Goal` contiene atributos fundamentales para su identificación, descripción y seguimiento:

- `id`: identificador único de la meta.
- `title`: título de la meta, obligatorio y visible en la interfaz del usuario.
- `description`: descripción opcional que permite contextualizar la meta.
- `metric`: unidad de medida personalizada definida por el usuario (por ejemplo, *horas*, *km*, *libros*).
- `totalDesiredAmount`: cantidad total objetivo que el usuario espera alcanzar.
- `creationDate`: fecha de creación de la meta.
- `isCompleted`: indicador booleano que refleja si la meta ha sido completada. Este campo se gestiona manualmente por el usuario, permitiendo mayor control y personalización.
- `progressList`: la lista de avances registrados para una meta particular.
- `user`: el usuario al cual la meta esta asociada.

```java
@Entity
public class Goal {

    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    2 usages
    @Column(nullable = false)
    private String title;

    2 usages
    @Column(nullable = true)
    private String description;

    2 usages
    @Column(nullable = false)
    private String metric;

    2 usages
    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;

    2 usages
    @OneToMany(mappedBy = "goal", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<Progress> progressList;

    2 usages
    @Column(nullable = true)
    private Double totalDesiredAmount;

    2 usages
    @Column(nullable = false)
    private LocalDate creationDate;


    2 usages
    private boolean isCompleted = false;
```

Figura 5: Modelo entidad-relación de la clase `Goal`.

3.4.3. Progress

La entidad **Progress** representa una entrada individual de avance registrada por el usuario dentro del contexto de una meta específica (`Goal`). Esta entidad permite cuantificar y almacenar los esfuerzos realizados por el usuario en un determinado día, lo cual resulta fundamental para calcular el progreso total y generar visualizaciones históricas.

Cada instancia de `Progress` contiene la siguiente información:

- `id`: identificador único de la entrada de progreso.
- `amount`: cantidad aportada hacia la meta, expresada en la unidad definida por el usuario (por ejemplo, *horas*, *km*, *libros*). Este campo es obligatorio y permite agregar contribuciones diarias al objetivo general.
- `date`: fecha en la que se realizó el progreso. Este dato es obligatorio y permite la agrupación mensual para la generación de gráficos.
- `updateNote`: nota opcional escrita por el usuario, utilizada para añadir contexto o reflexiones sobre ese avance en particular.
- `goal`: la meta a la que el avance registrado esta asociado.

```java
@Entity
public class Progress {

    2 usages
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;


    2 usages
    private String updateNote;

    2 usages
    @Column(nullable = false)
    private Double amount; // e.g., 5000 steps

    2 usages
    @Column(nullable = false)
    private LocalDate date; // progress for which day

    2 usages
    @ManyToOne
    @JoinColumn(name = "goal_id")
    private Goal goal;

```

Figura 6: Modelo entidad-relación de la clase `Progress`.

4. Implementación

4.1. Backend

El backend está implementado en Java utilizando el framework Spring Boot, y sigue una arquitectura modular que facilita la escalabilidad, mantenibilidad y claridad del código. A continuación, se describe la estructura de carpetas del directorio `goal-tracker/backend` y su propósito:

- `pom.xml`: archivo de configuración de Maven que define las dependencias del proyecto, plugins, y parámetros de compilación.
- `deployment_files/`: contiene los ficheros de despliegue necesarios para Kubernetes y Docker.
- `src/`: directorio principal que contiene el código fuente del backend.
- `target/`: carpeta generada automáticamente por Maven que contiene los archivos compilados (.class) y el archivo .jar resultante.

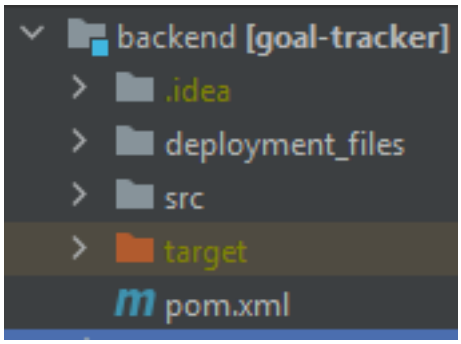


Figura 7: Los ficheros contenidos en `goal-tracker/backend`

Dentro de `src/main/java/com/project/goal_tracker`, la estructura del proyecto sigue una separación por responsabilidades conforme a buenas prácticas en aplicaciones Spring Boot:

- `GoalTrackerApplication.java`: clase principal que contiene el método `main()` y actúa como punto de entrada del backend. Se encarga de arrancar el contexto de Spring Boot.
- `config/`: contiene clases de configuración de la aplicación, como la configuración de seguridad.
- `controller/`: incluye los **controladores REST** (anotadas con `@RestController`), que exponen los endpoints de la API y gestionan las peticiones entrantes del frontend.
- `dto/` (*Data Transfer Objects*): agrupa las clases que se utilizan para transferir datos entre el frontend y el backend, sin exponer directamente las entidades de la base de datos.
- `misc/`: contiene código adicional que no encaja en otras categorías, en este caso la inicialización del administrador por defecto.
- `model/`: incluye las **entidades JPA**, que representan las tablas de la base de datos. Cada clase está anotada con `@Entity` y se encuentra vinculada mediante relaciones (como `@OneToMany`, `@ManyToOne`, etc.).
- `repository/`: contiene las interfaces que extienden `JpaRepository`, permitiendo el acceso a la base de datos mediante métodos de consulta automáticos o personalizados.
- `security/`: solamente contiene el filtro de JWT.
- `service/`: contiene la lógica de negocio de la aplicación, separando las reglas y procesos del controlador y permitiendo una arquitectura más limpia.
- `utils/`: clases de utilidad reutilizables.
- `src/main/resources/`: aunque se encuentra fuera del paquete de clases Java, esta carpeta contiene archivos de configuración (como `application.properties`) que se utilizan para inicializar variables de entorno y otras configuraciones necesarias de Spring Boot.

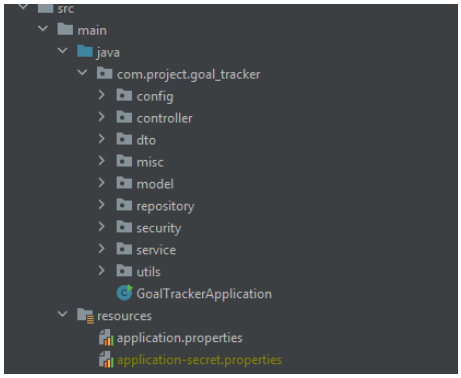


Figura 8: Los ficheros contenidos en `goal-tracker/backend/src/main`

4.1.1. config/

La carpeta `config` contiene una única clase: `SecurityConfig.java`. Esta clase es fundamental para definir el comportamiento de seguridad del backend en Spring Boot, configurando cómo se protegen los endpoints y cómo se integran los mecanismos de autenticación.

Una de las primeras acciones que se realiza en esta clase es la desactivación de CSRF (Cross-Site Request Forgery). Esto se debe a que la autenticación se basa en tokens JWT en lugar de cookies o sesiones de servidor, lo cual elimina la necesidad de protección CSRF en este contexto.

A través del método `securityFilterChain()`, se configuran las rutas públicas y protegidas. Las rutas relacionadas con la autenticación (`/auth/register`, `/auth/login`, `/auth/refresh`) se exponen libremente, mientras que otras como `/user/**` o `/goals/**` requieren autenticación. Además, se establece una política de sesión `STATELESS`, lo cual significa que el backend no guarda ninguna información de sesión en el servidor, confiando completamente en los JWT para validar cada solicitud de forma independiente. Esto habilita la escalabilidad horizontal del backend (múltiples instancias) sin necesidad de un almacenamiento compartido de sesión.

Otro aspecto clave es la integración del filtro personalizado `JwtFilter`, insertado antes del filtro por defecto `UsernamePasswordAuthenticationFilter`. Este filtro se encarga de extraer el token JWT de cada solicitud entrante, verificar su validez y, si es válido, proporcionar al contexto de seguridad las credenciales del usuario autenticado.

Por último, se define un `AuthenticationProvider` basado en `DaoAuthenticationProvider`,

configurado con una instancia de `UserDetailsService` y un encoder de contraseñas BCrypt con factor de trabajo 12. Esto asegura que las contraseñas estén protegidas mediante un algoritmo de hash robusto antes de almacenarse en la base de datos.

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig{

    1 usage
    @Autowired
    private JwtFilter jwtFilter;

    1 usage
    @Autowired
    private UserDetailsService userDetailsService;

    Cristian
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers( ...patterns: "/","/auth/register","/auth/login","/auth/refresh").permitAll()
                        .requestMatchers( ...patterns: "/home", "/user/**","/goals/**").authenticated()
                        .anyRequest().denyAll()
                )
                .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
                .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    Cristian
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    Cristian
    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
        provider.setPasswordEncoder(new BCryptPasswordEncoder( strength: 12));
        provider.setUserDetailsService(userDetailsService);
        return provider;

    }
```

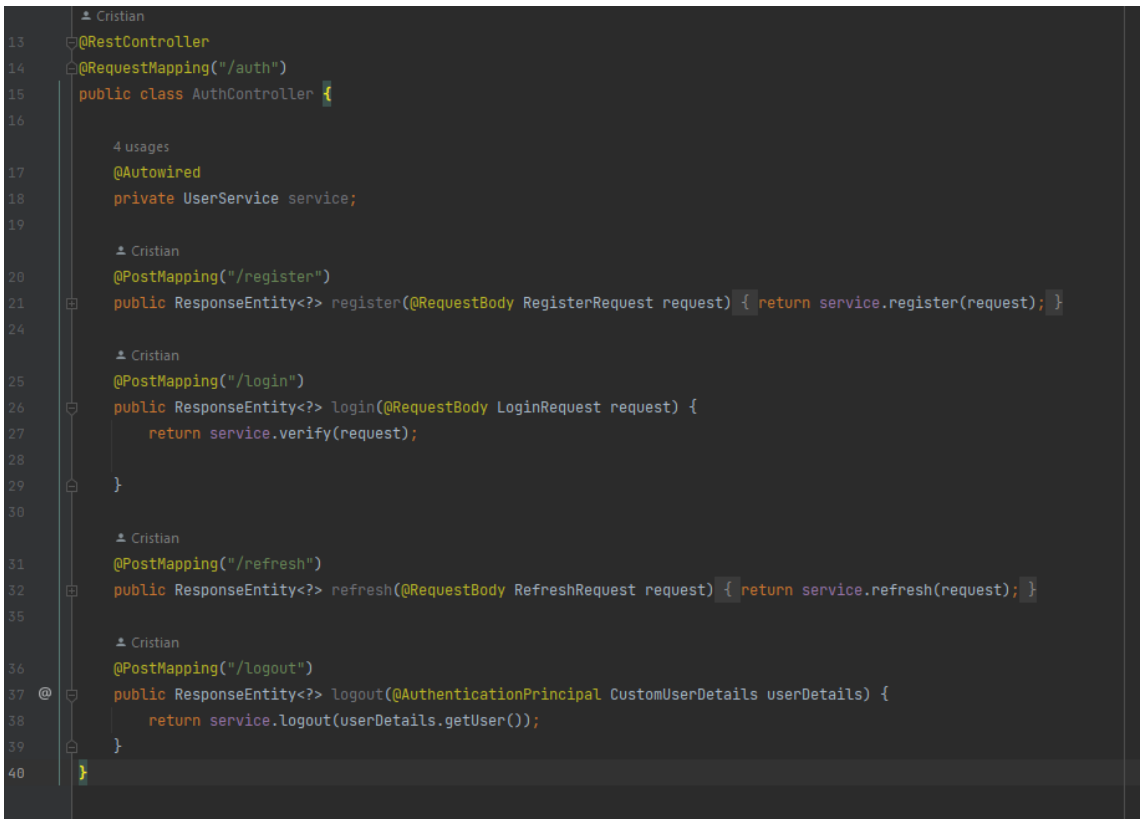
Figura 9: Contenido del fichero SecurityConfig.java

4.1.2. controller/

La carpeta `controller/` contiene las clases `AuthController.java`,`GoalController.java`, `ProgressController.java` y `UserController`.

AuthController.java Este controlador se encarga de gestionar todas las operaciones relacionadas con la autenticación y autorización de usuarios. Expone cuatro endpoints REST bajo el prefijo `/auth`, permitiendo a los usuarios registrarse, iniciar sesión, refrescar su token de acceso y cerrar sesión. Su propósito principal es facilitar un flujo de autenticación seguro mediante JWT, delegando la lógica al `UserService`.

- `POST /auth/register`: permite registrar un nuevo usuario en el sistema. Recibe un objeto `RegisterRequest` en el cuerpo de la petición, que contiene la información necesaria (nombre, email, contraseña y contraseña repetida). Retorna una respuesta HTTP con los dos tokens (JWT, RefreshToken) correspondiente si el registro es exitoso.
- `POST /auth/login`: permite a un usuario autenticarse. Se envía un `LoginRequest` con sus credenciales (email, contraseña). Si son válidas, el backend retorna un **access token** y un **refresh token** como respuesta.
- `POST /auth/refresh`: se utiliza para obtener un nuevo **access token** usando un **refresh token** válido. Esto permite extender la sesión del usuario sin que tenga que volver a introducir sus credenciales.
- `POST /auth/logout`: finaliza la sesión del usuario invalidando el refresh token almacenado. Para acceder a este endpoint, el usuario debe estar autenticado previamente. Se utiliza la anotación `@AuthenticationPrincipal` para obtener los detalles del usuario autenticado.



```
Cristian
@RestController
@RequestMapping("/auth")
public class AuthController {

    4 usages
    @Autowired
    private UserService service;

    Cristian
    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody RegisterRequest request) { return service.register(request); }

    Cristian
    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {
        return service.verify(request);

    }

    Cristian
    @PostMapping("/refresh")
    public ResponseEntity<?> refresh(@RequestBody RefreshRequest request) { return service.refresh(request); }

    Cristian
    @PostMapping("/logout")
    public ResponseEntity<?> logout(@AuthenticationPrincipal CustomUserDetails userDetails) {
        return service.logout(userDetails.getUser());
    }
}
```

Figura 10: Contenido del fichero AuthController.java

GoalController.java Este controlador gestiona todas las operaciones relacionadas con los objetivos (*goals*) de un usuario. Todos los endpoints están protegidos mediante autenticación, y siguen la convención de rutas anidadas bajo el prefijo `/user/{userId}/goals`, para asegurar que los recursos se consulten y manipulen únicamente por sus propietarios.

- `POST /user/{userId}/goals`: permite a un usuario autenticado crear un nuevo objetivo. Requiere un cuerpo con los datos de creación (`GoalCreate`) y realiza validaciones de entrada con `BindingResult`. Se valida también que el usuario autenticado coincida con el `userId` indicado en la ruta.

```java
@PostMapping("")
public ResponseEntity<?> createGoal(@AuthenticationPrincipal CustomUserDetails userDetails,
                                    @PathVariable Long userId,
                                    @Valid @RequestBody GoalCreate request,
                                    BindingResult bindingResult){
    AggregateOutput<String> out = new AggregateOutput<>();
    if(bindingResult.hasErrors()){
        bindingResult.getFieldErrors().forEach(error -> {
            out.error(error.getField(), error.getDefaultMessage());
        });
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }else{
        User user = userDetails.getUser();
        if (!userService.validAction(user, userId, out)){
            return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
        }
        User targetUser = userService.getUser(userId, out);
        if (targetUser == null){
            return out.toResponseEntity();
        }
        goalService.createGoal(targetUser,request, out);
        return out.toResponseEntity();
    }
}
```

Figura 11: Función `createGoal()`

- `GET /user/{userId}/goals`: lista todos los objetivos pertenecientes al usuario autenticado. Utiliza el servicio `GoalService` para obtener los objetivos y retornarlos en un formato estructurado con `GoalResponse`.

```
@Transactional
@GetMapping("")
public ResponseEntity<?> listGoals (@AuthenticationPrincipal CustomUserDetails userDetails,
                                    @PathVariable Long userId){
    User user = userDetails.getUser();
    AggregateOutput<GoalResponse> out = new AggregateOutput<>();

    if (!userService.validAction(user, userId, out)){
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }

    goalService.listGoals(userId,out);
    return out.toResponseEntity();
}
```

Figura 12: Función `listGoals()`

- `GET /user/{userId}/goals/{id}`: recupera un objetivo específico del usuario, verificando previamente que el objetivo pertenezca al usuario autenticado.

```
@GetMapping("/{id}")
public ResponseEntity<?> retrieveGoal (@AuthenticationPrincipal CustomUserDetails userDetails,
                                       @PathVariable Long userId,
                                       @PathVariable Long id){
    User user = userDetails.getUser();
    AggregateOutput<GoalResponse> out = new AggregateOutput<>();

    if (!userService.validAction(user, userId, out)){
        return out.toResponseEntity();
    }

    goalService.retrieveGoal(userId,id,out);

    return out.toResponseEntity();
}
```

Figura 13: Función `retrieveGoal()`

- `PATCH /user/{userId}/goals/{id}`: permite modificar parcialmente los datos de un objetivo. Recibe un objeto `GoalUpdate` con los campos a actualizar.

```java
@PatchMapping("/{id}")
public  ResponseEntity<?> updateGoal(@AuthenticationPrincipal CustomUserDetails userDetails,
                                     @PathVariable Long userId,
                                     @PathVariable Long id,
                                     @RequestBody GoalUpdate request){
    User user = userDetails.getUser();
    AggregateOutput<GoalResponse> out = new AggregateOutput<>();
    if (!userService.validAction(user, userId, out)){
        return out.toResponseEntity();
    }
    goalService.updateGoal(userId, id, request, out);

    return out.toResponseEntity();

}
```

Figura 14: Función `updateGoal()`

- `DELETE /user/{userId}/goals/{id}`: elimina el objetivo correspondiente al `id` proporcionado, siempre que pertenezca al usuario autenticado.

```java
@DeleteMapping("/{id}")
public ResponseEntity<?> deleteGoal(@AuthenticationPrincipal CustomUserDetails userDetails,
                                    @PathVariable Long userId,
                                    @PathVariable Long id){
    User user = userDetails.getUser();
    AggregateOutput<String> out = new AggregateOutput<>();
    if (!userService.validAction(user, userId, out)){
        return out.toResponseEntity();
    }
    goalService.deleteGoal(userId,id,out);
    return out.toResponseEntity();
}
```

Figura 15: Función `deleteGoal()`

ProgressController.java Este controlador gestiona el progreso de los objetivos definidos por el usuario. Todos los endpoints están anidados bajo la ruta `/user/{userId}/goals/{goalId}/progress` para asegurar que cada progreso esté asociado a un objetivo concreto y, a su vez, a un usuario específico. Cada operación verifica que el usuario autenticado tenga permisos sobre el recurso antes de ejecutarse.

- `GET /user/{userId}/goals/{goalId}/progress`: recupera todos los registros de progreso asociados a un objetivo. Valida tanto al usuario como la existencia del objetivo antes de listar.

```
31      @GetMapping("")
32  @   public ResponseEntity<?> listProgress(@AuthenticationPrincipal CustomUserDetails userDetails,
33                                            @PathVariable Long userId,
34                                            @PathVariable Long goalId) {
35          User user = userDetails.getUser();
36          AggregateOutput<ProgressResponse> out = new AggregateOutput<>();
37          if(!userService.validAction(user,userId,out)){
38              return out.toResponseEntity();
39          }
40          Goal goal = progService.retrieveGoal(userId,goalId,out);
41          if(out.haveErrors()){
42              return out.toResponseEntity();
43          }
44
45          progService.listProgress(goal,out);
46          return out.toResponseEntity();
47      }
48
```

Figura 16: Función `listProgress()`

- `GET /user/{userId}/goals/{goalId}/progress/{progressId}`: permite obtener un registro concreto de progreso. Se comprueba que el objetivo y el progreso pertenezcan al usuario autenticado.

```
49      @GetMapping("/{progressId}")
50  @   public ResponseEntity<?> retrieveProgress(@AuthenticationPrincipal CustomUserDetails userDetails,
51                                                @PathVariable Long userId,
52                                                @PathVariable Long goalId,
53                                                @PathVariable Long progressId){
54          User user = userDetails.getUser();
55          AggregateOutput<ProgressResponse> out = new AggregateOutput<>();
56          if(!userService.validAction(user,userId,out)){
57              return out.toResponseEntity();
58          }
59          Goal goal = progService.retrieveGoal(userId,goalId,out);
60          if(out.haveErrors()){
61              return out.toResponseEntity();
62          }
63
64          progService.retrieveProgress(goal,progressId,out);
65          return out.toResponseEntity();
66      }
```

Figura 17: Función `retrieveProgress()`

- `POST /user/{userId}/goals/{goalId}/progress`: permite registrar un nuevo progreso dentro de un objetivo. Se recibe un cuerpo JSON con los datos (`ProgressCreate`), se validan y luego se crea el progreso si todo es correcto.

```
68      @PostMapping("")
69  @   public ResponseEntity<?> createProgress(@AuthenticationPrincipal CustomUserDetails userDetails,
70                                              @PathVariable Long userId,
71                                              @PathVariable Long goalId,
72                                              @RequestBody ProgressCreate request,
73                                              BindingResult bindingResult) {
74          User user = userDetails.getUser();
75          AggregateOutput<ProgressResponse> out = new AggregateOutput<>();
76          if(bindingResult.hasErrors()){
77              bindingResult.getFieldErrors().forEach(error -> {
78                  out.error(error.getField(), error.getDefaultMessage());
79              });
80              return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
81          }
82          if(!userService.validAction(user,userId,out)){
83              return out.toResponseEntity();
84          }
85          Goal goal = progService.retrieveGoal(userId,goalId,out);
86          if(out.haveErrors()){
87              return out.toResponseEntity();
88          }
89
90          progService.createProgress(goal, request, out);
91          return out.toResponseEntity();
92      }
```

Figura 18: Función `createProgress()`

- `PATCH /user/{userId}/goals/{goalId}/progress/{progressId}`: permite modificar parcialmente un registro de progreso existente, siempre que esté vinculado a un objetivo del usuario autenticado.

```
94      @PatchMapping("/{progressId}")
95  @   ResponseEntity<?> updateProgress(@AuthenticationPrincipal CustomUserDetails userDetails,
96                                       @PathVariable Long userId,
97                                       @PathVariable Long goalId,
98                                       @PathVariable Long progressId,
99                                       @RequestBody ProgressUpdate request){
100         User user = userDetails.getUser();
101         AggregateOutput<ProgressResponse> out = new AggregateOutput<>();
102         if(!userService.validAction(user,userId,out)){
103             return out.toResponseEntity();
104         }
105         Goal goal = progService.retrieveGoal(userId,goalId,out);
106         if(out.haveErrors()){
107             return out.toResponseEntity();
108         }
109
110         progService.updateProgress(goal, progressId, request, out);
111         return out.toResponseEntity();
112     }
113
```

Figura 19: Función `updateProgress()`

- `DELETE /user/{userId}/goals/{goalId}/progress/{progressId}`: elimina un registro de progreso, tras validar que el usuario y el objetivo asociados sean correctos.

```
114     @DeleteMapping("/{progressId}")
115 @   ResponseEntity<?> deleteProgress(@AuthenticationPrincipal CustomUserDetails userDetails,
116                                      @PathVariable Long userId,
117                                      @PathVariable Long goalId,
118                                      @PathVariable Long progressId){
119         User user = userDetails.getUser();
120         AggregateOutput<ProgressResponse> out = new AggregateOutput<>();
121         if(!userService.validAction(user,userId,out)){
122             return out.toResponseEntity();
123         }
124         Goal goal = progService.retrieveGoal(userId,goalId,out);
125         if(out.haveErrors()){
126             return out.toResponseEntity();
127         }
128
129         progService.deleteProgress(goal, progressId, out);
130         return out.toResponseEntity();
131     }
132 }
```

Figura 20: Función `deleteProgress()`

UserController.java Este controlador gestiona operaciones relacionadas con la información de usuario, incluyendo visualización de perfiles y operaciones administrativas como la promoción o bloqueo de cuentas. Algunas rutas están disponibles para cualquier usuario autenticado, mientras que otras requieren privilegios de administrador mediante la anotación `@PreAuthorize("hasRole('ADMIN')")`.

- `GET /user/profile`: obtiene el perfil del usuario autenticado usando el token JWT.
- `GET /user/{userId}/profile`: obtiene el perfil público de cualquier usuario a partir de su `userId`, útil en contextos administrativos.

```
30
31      //this one will be used to get your JWT profile
        Cristian
32      @GetMapping("/profile")
33  @   public ResponseEntity<?> profile(@AuthenticationPrincipal CustomUserDetails userDetails){
34          AggregateOutput<ProfileResponse> out = new AggregateOutput<>();
35          return service.getProfile(userDetails.getUsername(),out);
36      }
37
        Cristian
38      @GetMapping("/{userId}/profile")
39      public ResponseEntity<?> profileFromUser(@AuthenticationPrincipal CustomUserDetails userDetails,
40                                               @PathVariable Long userId){
41          AggregateOutput<ProfileResponse> out = new AggregateOutput<>();
42          User user = service.getUser(userId, out);
43          return service.getProfile(user.getEmail(),out);
44      }
45
```

Figura 21: Funciones `profile()` y `profileFromUser()`

- `POST /user/{userId}/promote`: promueve a un usuario al rol de administrador. Solo accesible para usuarios con rol `ADMIN`.
- `POST /user/{userId}/demote`: revierte a un usuario al rol estándar. Solo accesible para administradores.

```
    //Takes in the email of the targetUser
    Cristian
    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping("/{userId}/promote")
    public ResponseEntity<?> promote(@AuthenticationPrincipal CustomUserDetails userDetails,
                                     @PathVariable Long userId){
        return service.manageRole(
                userDetails.getUsername(),
                userId,
                 makeAdmin: true);
    }

    Cristian
    @PreAuthorize("hasRole('ADMIN')")
    @PostMapping("/{userId}/demote")
    public ResponseEntity<?> demote(@AuthenticationPrincipal CustomUserDetails userDetails,
                                    @PathVariable Long userId){
        return service.manageRole(
                userDetails.getUsername(),
                userId,
                 makeAdmin: false);
    }
```

Figura 22: Funciones `promote()` y `demote()`

- `POST /user/{userId}/ban`: desactiva la cuenta de un usuario.
- `POST /user/{userId}/unban`: reactiva la cuenta previamente deshabilitada.

Figura 23: Funciones `banUser()` y `unbanUser()`

- `GET /user`: retorna la lista completa de usuarios del sistema, disponible únicamente para administradores.

Figura 24: Función `listUsers()`

4.1.3. dto/

La carpeta `dto/` contiene 13 clases, todas con un propósito común: servir de intermediarios entre los datos que se envían o reciben desde el frontend y la lógica del backend. Es decir, los DTOs nos permiten no depender directamente del modelo, evitando exponer entidades internas como `User`, `Goal` o `Progress`.

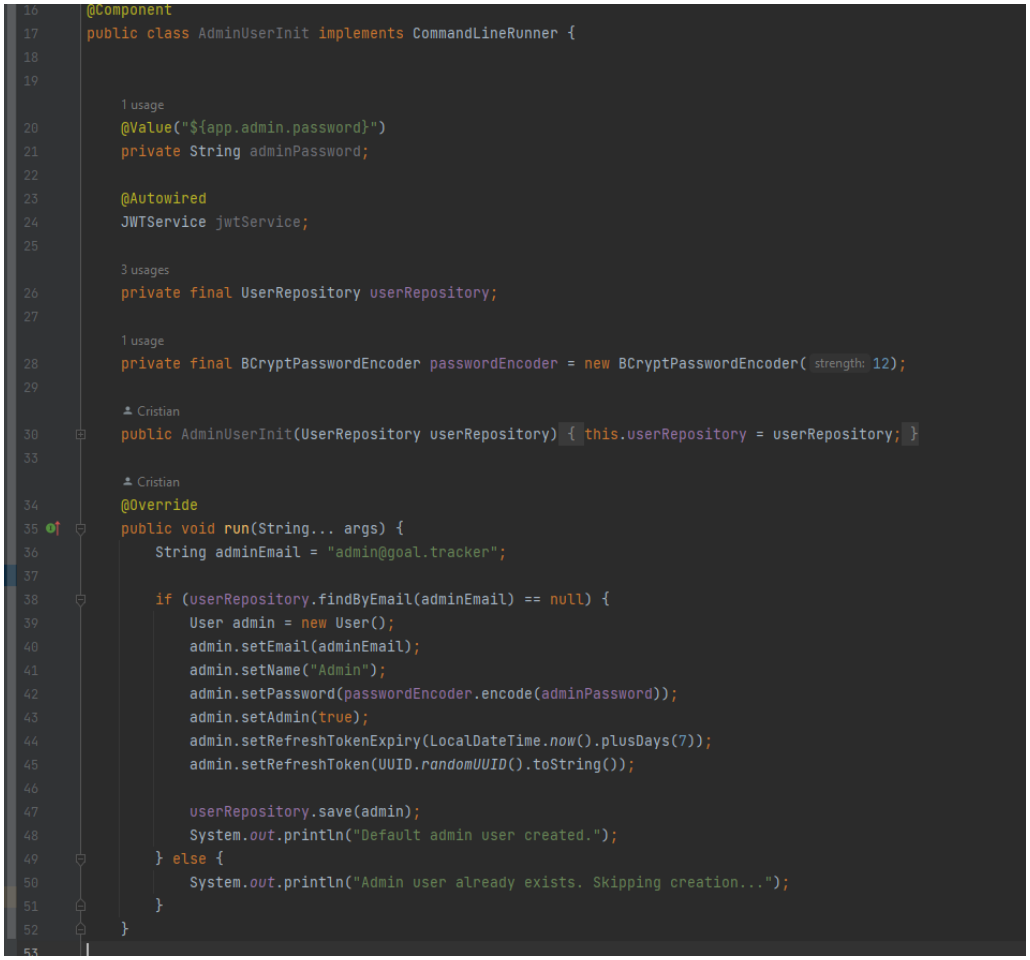
Cada clase en esta carpeta representa un contexto específico. Si es una petición, el nombre acaba en `Request`, y si es una respuesta, en `Response`. Esto hace que el código sea más limpio y predecible, ya que sabemos en todo momento qué esperamos recibir o devolver. Además las clases que son `Response`, tienen un método `.fromEntity()` que nos permite crear la respuesta a partir del una entidad del modelo.

4.1.4. misc/

La carpeta `misc/` contiene el fichero `AdminUserInit.java`, el cual se encarga de inicializar un usuario administrador por defecto en el arranque de la aplicación. Esta clase implementa la interfaz `CommandLineRunner`, por lo que su método `run()` se ejecuta automáticamente una vez que el contexto de Spring se ha cargado por completo(y además se ejecuta solo una vez).

El objetivo de esta clase es asegurar que siempre exista al menos un usuario con privilegios de administrador. Si ya existe un usuario con el correo `admin@goal.tracker`, el sistema no hace nada. En caso contrario, se crea uno nuevo con los datos por de-

fecto y se le asignan privilegios de administrador. La contraseña se toma desde una propiedad externa (`app.admin.password`), y se codifica con `BCrypt`.



```java
@Component
public class AdminUserInit implements CommandLineRunner {


    1 usage
    @Value("${app.admin.password}")
    private String adminPassword;

    @Autowired
    JWTService jwtService;

    3 usages
    private final UserRepository userRepository;

    1 usage
    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder( strength: 12);

    Cristian
    public AdminUserInit(UserRepository userRepository) { this.userRepository = userRepository; }

    Cristian
    @Override
    public void run(String... args) {
        String adminEmail = "admin@goal.tracker";

        if (userRepository.findByEmail(adminEmail) == null) {
            User admin = new User();
            admin.setEmail(adminEmail);
            admin.setName("Admin");
            admin.setPassword(passwordEncoder.encode(adminPassword));
            admin.setAdmin(true);
            admin.setRefreshTokenExpiry(LocalDateTime.now().plusDays(7));
            admin.setRefreshToken(UUID.randomUUID().toString());

            userRepository.save(admin);
            System.out.println("Default admin user created.");
        } else {
            System.out.println("Admin user already exists. Skipping creation...");
        }
    }
```

Figura 25: Clase `AdminUserInit.java`: creación de un usuario administrador por defecto

4.1.5. model/

La carpeta `model/` contiene todas las clases de modelo del proyecto, como `User`, `Goal` y `Progress`. Estas entidades representan directamente las tablas de la base de datos y ya han sido descritas en la sección correspondiente a persistencia.

No obstante, hay una clase adicional relevante: `CustomUserDetails`. Esta clase implementa la interfaz `UserDetails` de Spring Security, permitiendo integrar la entidad `User` con el sistema de autenticación de Spring. Su función principal es adaptar el modelo de usuario de la aplicación al modelo esperado por Spring Security.

Entre sus métodos más relevantes están:

- `getAuthorities()`: asigna los roles `ROLE_USER` por defecto, y `ROLE_ADMIN` si el usuario tiene privilegios administrativos.
- `isEnabled()`: determina si la cuenta está activa, devolviendo `false` si el usuario está baneado.
- `getUsername()` y `getPassword()`: proporcionan las credenciales necesarias para la autenticación.

Esta clase permite que Spring pueda autenticar usuarios directamente desde la entidad `User`, sin necesidad de adaptar la lógica de seguridad en otros componentes.

```java
public class CustomUserDetails implements UserDetails {
    6 usages
    private User user;

    1 usage  Cristian
    public CustomUserDetails(User user) { this.user = user; }

    1 usage  Cristian
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        List<GrantedAuthority> authorities = new ArrayList<>();
        authorities.add(new SimpleGrantedAuthority( role: "ROLE_USER"));

        if (user.isAdmin()) {
            authorities.add(new SimpleGrantedAuthority( role: "ROLE_ADMIN"));
        }

        return authorities;
    }

    Cristian
    @Override
    public String getPassword() { return user.getPassword(); }

    Cristian
    @Override
    public String getUsername() { return user.getEmail(); }

    Cristian
    @Override
    public boolean isAccountNonExpired() { return true; }

    Cristian
    @Override
    public boolean isAccountNonLocked() { return true; }

    Cristian
    @Override
    public boolean isCredentialsNonExpired() { return true; }

    Cristian
    @Override
    public boolean isEnabled() { return !user.isBanned(); }

    Cristian
    public User getUser() { return user; }
}
```

Figura 26: Clase `CustomUserDetails.java`: adaptación del modelo `User` para la autenticación

4.1.6. repository/

La carpeta `repository/` tiene las interfaces que extienden el JpaRepository con una clase del modelo, como `User`, `Goal` y `Progress`. Esta interfaz es que la nos permite manipular objetos en la base de datos de MySQL.

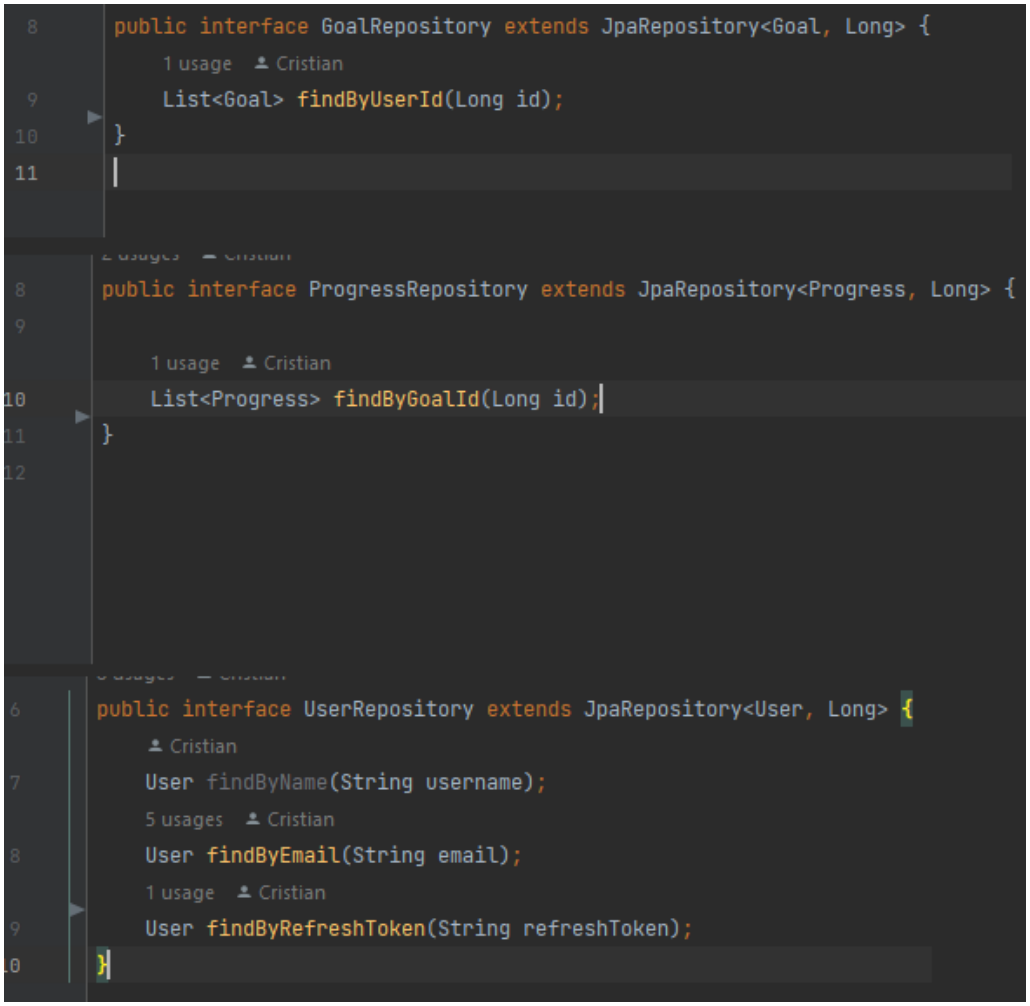


Figura 27: Interfaces del paquete `repository/`, extendiendo `JpaRepository`

4.1.7. security/

La carpeta `security/` contiene la clase `JwtFilter`, un filtro personalizado que intercepta todas las peticiones entrantes para validar tokens JWT y autenticar usuarios automáticamente. Esta clase define dos métodos principales:

- `doFilterInternal()`: se ejecuta por cada petición y realiza el proceso de autenticación JWT. Su flujo es:
  1. Extrae el token con `resolveToken()`.
  2. Intenta obtener el email desde el token.
  3. Si no hay usuario autenticado y el email es válido, se recuperan los datos del usuario.
  4. Si el token es válido para ese usuario, se configura la autenticación en el contexto de Spring Security.
  5. La ejecución sigue su curso mediante `chain.doFilter()`.

```java
protected void doFilterInternal(
        HttpServletRequest  request,
        HttpServletResponse response,
        FilterChain         chain
) throws ServletException, IOException {
    String token = resolveToken(request);
    String email = null;

    if (token != null) {
        try {
            email = jwtService.extractEmail(token);
        } catch (io.jsonwebtoken.JwtException e) {
            // Log the problem but DON'T abort the chain
            logger.debug( message: "Invalid or expired JWT: {}", e.fillInStackTrace());
        }
    }

    if (email != null
            && SecurityContextHolder.getContext().getAuthentication() == null
    ) {
        UserDetails userDetails = userDetailsService.loadUserByUsername(email);

        if (jwtService.validateToken(token, userDetails)) {
            UsernamePasswordAuthenticationToken authToken =
                    new UsernamePasswordAuthenticationToken(
                            userDetails,
                             credentials: null,
                            userDetails.getAuthorities()
                    );
            authToken.setDetails(
                    new WebAuthenticationDetailsSource()
                            .buildDetails(request)
            );
            SecurityContextHolder
                    .getContext()
                    .setAuthentication(authToken);
        }
    }

    // Always continue the filter chain
    chain.doFilter(request, response);
}
```

Figura 28: Método `doFilterInternal()`

```java
@    private String resolveToken(HttpServletRequest request) {
        // 1) Authorization header
        String bearer = request.getHeader(HttpHeaders.AUTHORIZATION);
        if (bearer != null && bearer.startsWith("Bearer ")) {
            return bearer.substring( beginIndex: 7);
        }
        // 2) JWT cookie
        if (request.getCookies() != null) {
            for (Cookie c : request.getCookies()) {
                if ("JWT".equals(c.getName())) {
                    return c.getValue();
                }
            }
        }
        return null;
    }
}
```

Figura 29: Método `resolveToken()`

- `resolveToken()`: método auxiliar que busca el token JWT en dos ubicaciones posibles:
  - En la cabecera `Authorization` si empieza con `Bearer` .
  - En una cookie llamada `JWT`, si existe.

  Si no se encuentra en ninguno de los dos lugares, devuelve `null`.

Rol de JwtFilter en la autenticación JWT La clase `JwtFilter` cumple una función clave dentro del sistema de seguridad de la aplicación: intercepta todas las peticiones HTTP entrantes antes de que lleguen a los controladores, y si detecta un token JWT válido, autentica al usuario automáticamente.

Este enfoque permite mantener sesiones sin necesidad de almacenar información en el servidor, ya que toda la verificación se hace mediante el token. Además, permite separar completamente la lógica de autenticación del resto de la aplicación, respetando el principio de responsabilidad única.

Este filtro se ejecuta una sola vez por petición, gracias a que extiende de `OncePerRequestFilter`, lo cual lo hace eficiente y predecible. Este filtro es el que se añade en `SecurityConfig.java`.

4.1.8. service/

La carpeta `service/` contiene la lógica de negocio y agrupa las siguientes clases: `CustomUserDetailsService.java`, `GoalService.java`, `JWTService.java`, `ProgressService.java` y `UserService.java`.

```java
2 usages  Cristian
@Service
public class CustomUserDetailsService implements UserDetailsService {

    2 usages
    private UserRepository userRepo;

    Cristian
    @Autowired
    public CustomUserDetailsService(UserRepository userRepo) { this.userRepo = userRepo; }


    1 usage  Cristian
    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
        User user = userRepo.findByEmail(email);
        if(user == null){
            throw new UsernameNotFoundException("User not found with email: " + email);

        }
        if (user.isBanned()) {
            throw new DisabledException("User is banned");
        }
        return new CustomUserDetails(user);
    }
}

```

Figura 30: Clase CustomUserDetailsService

Una de las piezas clave en la autenticación es la clase `CustomUserDetailsService`, que implementa la interfaz `UserDetailsService` de Spring Security. Esta clase se encarga de recuperar el usuario desde la base de datos y adaptarlo al sistema de seguridad de Spring mediante la clase `CustomUserDetails`.

El método `loadUserByUsername` recibe el email del usuario, busca el objeto `User` correspondiente en el repositorio, y si lo encuentra y no está baneado, lo encapsula en un `CustomUserDetails`. Si no se encuentra el usuario, lanza una excepción `UsernameNotFoundException`; y si el usuario está baneado, lanza una excepción `DisabledException`.

GoalService.java Esta clase se encarga de toda la lógica de negocio relacionada con los objetivos (`Goal`). Proporciona métodos para crear, leer, actualizar, listar y eliminar objetivos, así como validar los datos recibidos.

- **createGoal(User user, GoalCreate request, AggregateOutput<?>out)**

  Se encarga de crear un nuevo objetivo asociado al usuario recibido. Primero valida los datos con `validGoalObject`, luego instancia un nuevo `Goal`, rellena sus campos y lo guarda en la base de datos. Finalmente, añade un mensaje informativo en la salida.

```
29 @    public void createGoal(User user, GoalCreate request, AggregateOutput<?> out){
30          if(!validGoalObject(request.getTitle(),request.getMetric(), request.getTotalDesiredAmount(), out)){
31              return;
32          }
33          Goal goal = new Goal();
34          goal.setUser(user);
35          goal.setTitle(request.getTitle());
36          goal.setDescription(request.getDescription());
37          goal.setMetric(request.getMetric());
38          goal.setTotalDesiredAmount(request.getTotalDesiredAmount());
39          goal.setCreationDate(LocalDate.now());
40          goalRepository.save(goal);
41          out.info( key: "message", text: "Successfully created the goal",HttpStatus.CREATED);
42      }
43
```

Figura 31: Método createGoal() en GoalService.java

- **lookupGoal(Long userId, Long id, AggregateOutput<?>out)**

  Busca un objetivo por su ID y comprueba que pertenezca al usuario indicado. Si no existe o no pertenece al usuario, añade un error al objeto de salida. Devuelve el objetivo si se encuentra y es válido, o `null` en caso contrario.

```
44      public Goal lookupGoal(Long userId, Long id, AggregateOutput<?> out){
45          Optional<Goal> opt = goalRepository.findById(id);
46          if (opt.isEmpty()){
47              out.error(AggregateOutput.GOAL_NOT_FOUND, text: "Goal doesn't exist", HttpStatus.NOT_FOUND);
48              return null;
49          }
50          Goal goal = opt.get();
51          if(!Objects.equals(userId, goal.getUser().getId())){
52              out.error(AggregateOutput.GOAL_NOT_FOUND, text: "Goal doesn't exist", HttpStatus.NOT_FOUND);
53              return null;
54          }
55          return goal;
56      }
57
```

Figura 32: Método lookupGoal() en GoalService.java

- **listGoals(Long userId, AggregateOutput<GoalResponse>out)**

  Recupera todos los objetivos asociados a un usuario. Convierte cada `Goal` en un DTO `GoalResponse` y los agrega a la salida. Añade un mensaje de éxito con código HTTP 200.

```java
public void listGoals(Long userId, AggregateOutput<GoalResponse> out){

    List<Goal> goals = goalRepository.findByUserId(userId);
    List<GoalResponse> list = new ArrayList<>();
    for (Goal goal : goals) {
        list.add(GoalResponse.fromEntity(goal));
    }
    out.setData(list);
    out.info( key: "message", text: "Goals related to the user successfully retrieved", HttpStatus.OK);
}
```

Figura 33: Método listGoals() en GoalService.java

- **retrieveGoal(Long userId, Long id, AggregateOutput<GoalResponse>out)**

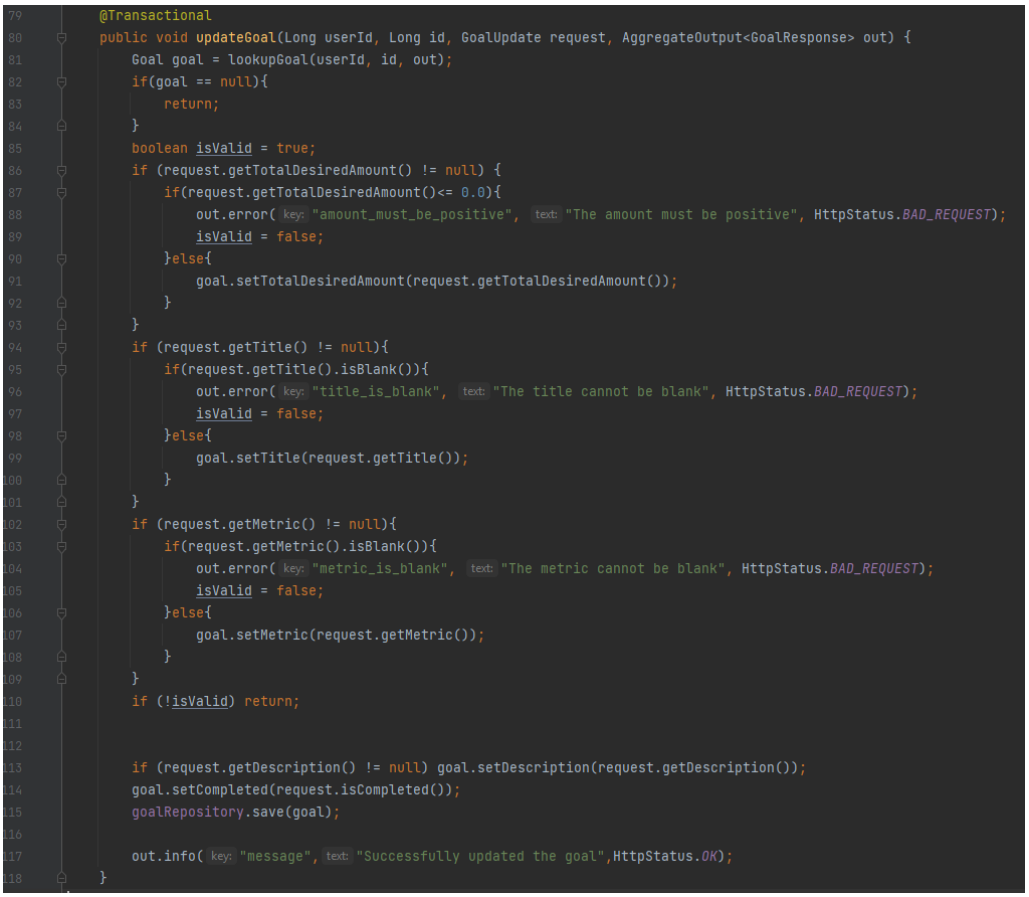
  Obtiene un objetivo concreto llamando a `lookupGoal`. Si existe, convierte el objetivo a DTO y lo añade a la salida junto a un mensaje de éxito.

```java
public void retrieveGoal(Long userId, Long id, AggregateOutput<GoalResponse> out){
    Goal goal = lookupGoal(userId, id, out);
    if(goal == null){
        return;
    }
    out.append(GoalResponse.fromEntity(goal));
    out.info( key: "message", text: "Successfully retrieved the goal",HttpStatus.OK);
}
```

Figura 34: Método retrieveGoal() en GoalService.java

- **updateGoal(Long userId, Long id, GoalUpdate request, AggregateOutput<GoalResponse>out)**

  Actualiza un objetivo existente tras validar que pertenezca al usuario. Valida los campos recibidos en la petición (título, métrica, cantidad deseada), aplicando restricciones (no vacío, cantidad positiva). Actualiza los campos y guarda el objetivo. Añade mensajes de error o éxito según corresponda. Se validan directamente en la funcion debido a que es una PATCH request, es decir, el cliente envía únicamente aquellos campos interesados en actualizar.



```
@Transactional
public void updateGoal(Long userId, Long id, GoalUpdate request, AggregateOutput<GoalResponse> out) {
    Goal goal = lookupGoal(userId, id, out);
    if(goal == null){
        return;
    }
    boolean isValid = true;
    if (request.getTotalDesiredAmount() != null) {
        if(request.getTotalDesiredAmount()<= 0.0){
            out.error( key: "amount_must_be_positive",  text: "The amount must be positive", HttpStatus.BAD_REQUEST);
            isValid = false;
        }else{
            goal.setTotalDesiredAmount(request.getTotalDesiredAmount());
        }
    }
    if (request.getTitle() != null){
        if(request.getTitle().isBlank()){
            out.error( key: "title_is_blank",  text: "The title cannot be blank", HttpStatus.BAD_REQUEST);
            isValid = false;
        }else{
            goal.setTitle(request.getTitle());
        }
    }
    if (request.getMetric() != null){
        if(request.getMetric().isBlank()){
            out.error( key: "metric_is_blank",  text: "The metric cannot be blank", HttpStatus.BAD_REQUEST);
            isValid = false;
        }else{
            goal.setMetric(request.getMetric());
        }
    }
    if (!isValid) return;


    if (request.getDescription() != null) goal.setDescription(request.getDescription());
    goal.setCompleted(request.isCompleted());
    goalRepository.save(goal);

    out.info( key: "message",  text: "Successfully updated the goal",HttpStatus.OK);
}
```

Figura 35: Método updateGoal() en GoalService.java

- **deleteGoal(Long userId, Long id, AggregateOutput<?>out)**

  Elimina un objetivo si pertenece al usuario y existe. Añade un mensaje de éxito con código HTTP 204 (sin contenido).

```java
@Transactional
public void deleteGoal(Long userId, Long id, AggregateOutput<?> out){
    Goal goal = lookupGoal(userId, id, out);
    if(goal == null){
        return;
    }
    goalRepository.delete(goal);
    out.info( key: "message", text: "Successfully deleted the goal", HttpStatus.NO_CONTENT);
}

```

Figura 36: Método deleteGoal() en GoalService.java

- **validGoalObject(String title, String metric, double totalDesiredAmount, AggregateOutput<?>out)**

  Valida que los parámetros esenciales para un objetivo (título, métrica, cantidad) cumplan condiciones básicas: cantidad positiva y strings no vacíos. Añade mensajes de error si alguna validación falla.

```java
public boolean validGoalObject(String title, String metric, double totalDesiredAmount, AggregateOutput<?> out){
    boolean isValid = true;
    if(totalDesiredAmount <= 0.0){
        out.error( key: "amount_must_be_positive", text: "The amount must be positive", HttpStatus.BAD_REQUEST);
        isValid = false;
    }
    if(title.isBlank()){
        out.error( key: "title_is_blank", text: "The title cannot be blank", HttpStatus.BAD_REQUEST);
        isValid = false;
    }
    if(metric.isBlank()){
        out.error( key: "metric_is_blank", text: "The metric cannot be blank", HttpStatus.BAD_REQUEST);
        isValid = false;
    }

    return isValid;

}
```

Figura 37: Método validGoalObject() en GoalService.java

JWTService.java Esta clase se encarga de la generación, validación y extracción de información de tokens JWT. Implementa la lógica para crear tokens firmados con una clave secreta generada dinámicamente, extraer datos de los tokens y validar su autenticidad y expiración.

- **JWTService()**

  Constructor que genera una clave secreta usando HmacSHA256 para firmar los tokens. La clave se codifica en Base64 URL-safe y se guarda para su uso posterior. La clave secreta se crea de manera aleatoria cada vez que se inicia el servicio y se imprime por consola (obviamente no es recomendado hacer eso). Esto tiene el efecto de invalidar todas las sesiones abiertas si el servidor es reiniciado. En un caso de verdad es clave sería fija y se cambiaría en caso de filtración para evitar que otros usuarios puedan crear tokens con las credenciales de otros.

Figura 38: Constructor de JWTService.java

- **generateToken(String email)**

  Genera un token JWT con los claims vacíos y el sujeto establecido como el email recibido. Define la fecha de emisión y expiración (1 hora después) y firma el token con la clave secreta.

```
36    public String generateToken(String email){
37
38        Map<String, Object> claims = new HashMap<>();
39
40        return Jwts.builder() JwtBuilder
41                .claims() BuilderClaims
42                .add(claims)
43                .subject(email)
44                .issuedAt(new Date(System.currentTimeMillis()))
45                .expiration(new Date(System.currentTimeMillis()+60*60*1000))
46                .and() JwtBuilder
47                .signWith(getKey())
48                .compact();
49    }
50
```

Figura 39: Método generateToken() en JWTService.java

- **getKey()**

  Método privado que decodifica la clave secreta Base64 y devuelve un objeto SecretKey para firmar y verificar tokens. Este método es utilizado internamente para asegurar que las firmas sean consistentes.

- **extractEmail(String token)**

  Extrae el email (subject) del token JWT, usando el método genérico `extractClaim` para obtener el claim correspondiente.

- **extractClaim(String token, Function<Claims, T>claimResolver)**

  Método genérico que permite extraer cualquier claim del token aplicando una función sobre los claims decodificados, facilitando así la reutilización del código para distintos tipos de claims.

- **extractAllClaims(String token)**

  Extrae todos los claims del token JWT validando la firma con la clave secreta. Usa el parser JWT para analizar el token.

```
66      private Claims extractAllClaims(String token) {
67
68          return Jwts.parser() JwtParserBuilder
69                  .verifyWith(getKey())
70                  .build() JwtParser
71                  .parseSignedClaims(token) Jws<Claims>
72                  .getPayload();
73      }
74
```

Figura 40: Método extractAllClaims() en JWTService.java

- **validateToken(String token, UserDetails userDetails)**

  Valida que el email extraído del token coincida con el username del usuario y que el token no haya expirado.

```
75  @   public boolean validateToken(String token, UserDetails userDetails) {
76          final String email = extractEmail(token);
77          return (email.equals(userDetails.getUsername()) && !isTokenExpired(token));
78      }
79
```

Figura 41: Método validateToken() en JWTService.java

- **isTokenExpired(String token)**

  Comprueba si la fecha de expiración del token es anterior a la fecha actual, indicando que el token ya no es válido.

- **extractExpiration(String token)**

  Extrae la fecha de expiración del token usando el método genérico `extractClaim`, que facilita la obtención de cualquier claim específico del token.

ProgressService.java Esta clase gestiona la lógica de negocio relacionada con los avances (`Progress`) de los objetivos (`Goal`). Proporciona métodos para listar, crear, obtener, actualizar y eliminar avances, además de validar la propiedad y los datos de cada progreso.

- **validOwner(Long userId, Goal goal, AggregateOutput<?>out)**

  Verifica que el usuario dado sea el propietario del objetivo. Si no lo es, registra un error en la salida.

```
30 @    private boolean validOwner(Long userId,  Goal goal, AggregateOutput<?> out){
31          if(!Objects.equals(userId, goal.getUser().getId())){
32              out.error(AggregateOutput.GOAL_NOT_FOUND, text: "Goal doesn't exist", HttpStatus.NOT_FOUND);
33              return false;
34          }
35          return true;
36      }
37
```

Figura 42: Método validOwner() en ProgressService.java

- **lookupProgress(Goal goal, Long progressId, AggregateOutput<?>out)**

  Busca un progreso por su ID, devuelve el objeto o `null` si no se encuentra, añadiendo un error en la salida.

```
38 @    private Progress lookupProgress(Goal goal, Long progressId, AggregateOutput<?> out){
39          Optional<Progress> progOpt = progressRepository.findById(progressId);
40          if(progOpt.isEmpty()){
41              out.error(AggregateOutput.PROGRESS_NOT_FOUND, text: "Progress wasnt found",HttpStatus.NOT_FOUND);
42              return null;
43          }
44          return progOpt.get();
45      }
46
```

Figura 43: Método lookupProgress() en ProgressService.java

- **progressRelatedToGoal(Progress prog, Goal goal, AggregateOutput<?>out)**

  Verifica que un progreso pertenezca al objetivo dado, añadiendo error en caso contrario.

```
47 @    private boolean progressRelatedToGoal(Progress prog, Goal goal, AggregateOutput<?> out){
48          if(!Objects.equals(prog.getGoal().getId(), goal.getId())){
49              out.error(AggregateOutput.PROGRESS_NOT_FOUND, text: "Progress wasnt found", HttpStatus.NOT_FOUND);
50              return false;
51          }
52          return true;
53      }
```

Figura 44: Método progressRelatedToGoal() en ProgressService.java

- **retrieveGoal(Long userId, Long goalId, AggregateOutput<?>out)**

  Obtiene un objetivo verificando que exista y que el usuario sea su propietario.

```
55      public Goal retrieveGoal (Long userId, Long goalId, AggregateOutput<?> out){
56          Optional<Goal> goalOpt = goalRepository.findById(goalId);
57          if(goalOpt.isEmpty()){
58              out.error(AggregateOutput.GOAL_NOT_FOUND, text: "Goal doesn't exist", HttpStatus.NOT_FOUND);
59              return null;
60          }
61          Goal goal = goalOpt.get();
62          if(!validOwner(userId, goal, out)){
63              return null;
64          }
65          return goal;
66      }
```

Figura 45: Método retrieveGoal() en ProgressService.java

- **listProgress(Goal goal, AggregateOutput<ProgressResponse>out)**

  Lista todos los avances asociados a un objetivo, convirtiéndolos en DTOs y añadiéndolos a la salida.

```
70 @    public void listProgress(Goal goal, AggregateOutput<ProgressResponse> out){
71
72          List<Progress> list = progressRepository.findByGoalId(goal.getId());
73
74          for (Progress p:
75               list) {
76              out.append(ProgressResponse.fromEntity(p));
77          }
78          out.info( key: "message", text: "Retrieved all progress related to the quota", HttpStatus.OK);
79      }
```

Figura 46: Método listProgress() en ProgressService.java

- **createProgress(Goal goal, ProgressCreate request, AggregateOutput<ProgressResponse>out)**

  Crea un nuevo progreso tras validar los datos, lo guarda y añade la respuesta.

```java
public void createProgress(Goal goal, ProgressCreate request, AggregateOutput<ProgressResponse> out){

    if(!validProgressObject(request.getAmount(),request.getUpdateNote(),out)){
        return;
    }
    Progress progress = new Progress();
    progress.setAmount(request.getAmount());
    progress.setUpdateNote(request.getUpdateNote());
    progress.setDate(LocalDate.now());
    progress.setGoal(goal);

    progressRepository.save(progress);
    out.append(ProgressResponse.fromEntity(progress));
    out.info( key: "message",  text: "Saved progress", HttpStatus.CREATED);
}
```

Figura 47: Método createProgress() en ProgressService.java

- **validProgressObject(Double amount, String updateNote, AggregateOutput<ProgressResponse>out)**

  Valida que la cantidad sea positiva y la nota no esté vacía, registrando errores si corresponde.

```java
private boolean validProgressObject(Double amount, String updateNote, AggregateOutput<ProgressResponse> out){
    boolean isValid = true;
    if(amount <= 0.0){
        out.error( key: "amount_must_be_positive",  text: "The amount must be positive", HttpStatus.BAD_REQUEST);
        isValid = false;
    }
    if(updateNote.isBlank()){
        out.error( key: "update_note_is_blank",  text: "The update note cannot be blank", HttpStatus.BAD_REQUEST);
        isValid = false;
    }
    return isValid;
}
```

Figura 48: Método validProgressObject() en ProgressService.java

- **retrieveProgress(Goal goal, Long progressId, AggregateOutput<ProgressResponse**

  Obtiene un progreso por ID y verifica que pertenezca al objetivo, añadiéndolo a la salida.

```java
public void retrieveProgress(Goal goal, Long progressId, AggregateOutput<ProgressResponse> out) {
    Progress prog = lookupProgress(goal,progressId,out);
    if(prog == null){
        return;
    }
    if(!progressRelatedToGoal(prog, goal, out)){
        return;
    }
    out.append(ProgressResponse.fromEntity(prog));
    out.info( key: "message", text: "Found progress", HttpStatus.OK);
}
```

Figura 49: Método retrieveProgress() en ProgressService.java

- **updateProgress(Goal goal, Long progressId, ProgressUpdate request, AggregateOutput<ProgressResponse>out)**

  Actualiza un progreso existente tras validarlo y guardarlo, añadiendo la respuesta y mensaje de éxito.

```java
public void updateProgress(Goal goal, Long progressId, ProgressUpdate request, AggregateOutput<ProgressResponse> out){
    Progress prog = lookupProgress(goal,progressId,out);
    if(prog == null){
        return;
    }
    if(!progressRelatedToGoal(prog, goal, out)){
        return;
    }

    if(!validProgressObject(request.getAmount(),request.getUpdateNote(),out)){
        return;
    }

    if(request.getUpdateNote() != null) prog.setUpdateNote(request.getUpdateNote());
    if(request.getAmount() != null) prog.setAmount(request.getAmount());

    progressRepository.save(prog);
    out.append(ProgressResponse.fromEntity(prog));
    out.info( key: "message", text: "Successfully updated progress", HttpStatus.OK);

}
```

Figura 50: Método updateProgress() en ProgressService.java

- **deleteProgress(Goal goal, Long progressId, AggregateOutput<ProgressResponse>**

  Elimina un progreso si pertenece al objetivo y existe, añadiendo un mensaje de éxito.

```java
public void deleteProgress(Goal goal, Long progressId, AggregateOutput<ProgressResponse> out) {
    Progress prog = lookupProgress(goal,progressId,out);
    if(prog == null){
        return;
    }
    if(!progressRelatedToGoal(prog, goal, out)){
        return;
    }

    progressRepository.delete(prog);
    out.append(ProgressResponse.fromEntity(prog));
    out.info( key: "message", text: "Successfully deleted progress", HttpStatus.NO_CONTENT);
}
```

Figura 51: Método deleteProgress() en ProgressService.java

UserService.java Esta clase gestiona la lógica de negocio relacionada con los usuarios (`User`). Proporciona métodos para autenticación, registro, gestión de perfil, roles, estado de cuenta y listado de usuarios, además de validar datos y permisos.

- **getProfile(String email, AggregateOutput<ProfileResponse>out)**

  Obtiene el perfil de usuario por correo electrónico. Si no se encuentra, registra error y devuelve 404; si existe, devuelve los datos del perfil.

```java
public ResponseEntity<?> getProfile(String email,AggregateOutput<ProfileResponse> out){

    User user =  userRepository.findByEmail(email);
    if (user == null){
        out.error( key: "user_not_found", text: "User couldn't be found");
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(out.getOutput());
    }else{
        out.append(new ProfileResponse(user.getId(), user.getName(),user.getEmail(),user.isAdmin()));
        return ResponseEntity.status(HttpStatus.OK).body(out.getOutput());
    }

}
```

Figura 52: Método getProfile() en UserService.java

- **verify(LoginRequest request)**

  Valida las credenciales de inicio de sesión. Autentica con Spring Security, genera tokens JWT y refresh, guarda el token de refresco, devuelve las cookies y la información o errores según el resultado.

```java
public ResponseEntity<?> verify(LoginRequest request) {

    AggregateOutput<String> out = new AggregateOutput<>();

    if (request.getEmail() == null || request.getEmail().isBlank() ||
            request.getPassword() == null || request.getPassword().isBlank()) {
        out.error( key: "message",  text: "Email and password must not be empty");
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }

    try {
        Authentication authentication =
                authManager.authenticate(new UsernamePasswordAuthenticationToken(request.getEmail(),
                        request.getPassword()));

        if(authentication.isAuthenticated()){
            User user = userRepository.findByEmail(request.getEmail());

            // If authentication is successful, generate the JWT token
            String accessToken = jwtService.generateToken(user.getEmail());
            String refreshToken = generateRefreshToken();

            user.setRefreshToken(refreshToken);
            user.setRefreshTokenExpiry(LocalDateTime.now().plusDays(7));
            userRepository.save(user);

            out.info( key: "accessToken", accessToken);
            out.info( key: "refreshToken", refreshToken);
            out.info( key: "message",  text: "Login successful");

            ResponseCookie accessTokenCookie = ResponseCookie.from( name: "JWT", accessToken)
                    .httpOnly(true)
                    .secure(true)
                    .path("/")
                    .maxAge(Duration.ofHours(1))
                    .sameSite("Lax")
                    .build();

            ResponseCookie refreshTokenCookie = ResponseCookie.from( name: "RefreshToken", refreshToken)
                    .httpOnly(true)
                    .secure(true)
                    .path("/")
                    .maxAge(Duration.ofDays(7))  // typically longer expiry for refresh token
                    .sameSite("Lax")
                    .build();

            return ResponseEntity
                    .ok()
                    .header(HttpHeaders.SET_COOKIE, accessTokenCookie.toString())
                    .header(HttpHeaders.SET_COOKIE, refreshTokenCookie.toString())
                    .body(out.getOutput());
        }else{
            out.error( key: "message",  text: "Invalid credentials");
            return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
        }
    } catch (InternalAuthenticationServiceException e) {
        Throwable cause = e.getCause();
        if (cause instanceof DisabledException) {
            out.error( key: "message",  text: "User account is banned");
            return out.setStatus(HttpStatus.FORBIDDEN).toResponseEntity();
        }

        out.error( key: "message",  text: "Authentication failed");
        return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
    } catch (BadCredentialsException e) {
        // Catching invalid credentials specifically
        out.error( key: "message",  text: "Invalid credentials");
        return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
    }catch (Exception e) {
        out.error( key: "message",  text: "An error occurred during login");
        return out.setStatus(HttpStatus.INTERNAL_SERVER_ERROR).toResponseEntity();
    }
}
```

Figura 53: Método verify() en UserService.java

- **generateRefreshToken()**

  Genera un token de refresco único con UUID aleatoria.

- **refresh(RefreshRequest request)**

  Valida y renueva el token de acceso mediante un token de refresco válido y no expirado. Devuelve el nuevo token o error.

```java
@   public ResponseEntity<?> refresh(RefreshRequest request){

        AggregateOutput<String> out = new AggregateOutput<>();

        String refreshToken = request.getRefreshToken();

        if (refreshToken == null || refreshToken.isEmpty()) {
            out.error( key: "message", text: "Refresh token is required");
            return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
        }

        User user = userRepository.findByRefreshToken(refreshToken);

        if (user == null) {
            out.error( key: "message", text: "Invalid refresh token");
            return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
        }

        if (request.getRefreshToken().equals(user.getRefreshToken())) {
            if(LocalDateTime.now().isBefore(user.getRefreshTokenExpiry())){
                String newAccessToken = jwtService.generateToken(user.getEmail());
                out.info( key: "accessToken", newAccessToken);
                out.info( key: "message", text: "Successful token refresh");
                return out.setStatus(HttpStatus.OK).toResponseEntity();
            }else{
                out.error( key: "message", text: "Please log in again");
                return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
            }
        } else {
            out.error( key: "message", text: "Invalid refresh token");
            return out.setStatus(HttpStatus.UNAUTHORIZED).toResponseEntity();
        }
    }
```

Figura 54: Método refresh() en UserService.java

- **register(RegisterRequest request)**

  Valida los datos del registro (nombre, email, contraseña y confirmación), verifica que no exista el email, crea un nuevo usuario con contraseña codificada, genera tokens y devuelve respuesta con cookies y mensajes.

```java
public ResponseEntity<?> register(RegisterRequest request) {

    AggregateOutput<String> out = new AggregateOutput<>();
    boolean validRegistration = true;

    if(invalidName(request.getName(), out)){
        validRegistration = false;
    }

    if (!request.getPassword().equals(request.getConfirmPassword())) {
        out.error( key: "passwords_dont_match", text: "Passwords do not match");
        validRegistration = false;
    }

    if(invalidPassword(request.getPassword(), out)){
        validRegistration = false;
    }

    if(invalidEmail(request.getEmail(), out)){
        validRegistration = false;
    }


    if (userRepository.findByEmail(request.getEmail()) != null) {
        out.info( key: "email_already_registered", text: "Email is already registered");
        validRegistration = false;
    }

    if(!validRegistration){
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }


    User user = new User();
    user.setEmail(request.getEmail());
    user.setName(request.getName());
    user.setPassword(encoder.encode(request.getPassword()));
    user.setRefreshTokenExpiry(LocalDateTime.now().plusDays(7));
    user.setRefreshToken(generateRefreshToken());

    userRepository.save(user);

    String accessToken = jwtService.generateToken(user.getEmail());

    out.info( key: "refreshToken",user.getRefreshToken());
    out.info( key: "accessToken", accessToken);
    out.info( key: "message", text: "Registration successful");

    ResponseCookie accessTokenCookie = ResponseCookie.from( name: "JWT", accessToken)
            .httpOnly(true)
            .secure(true)
            .path("/")
            .maxAge(Duration.ofHours(1))
            .sameSite("Lax")
            .build();

    ResponseCookie refreshTokenCookie = ResponseCookie.from( name: "RefreshToken", user.getRefreshToken())
            .httpOnly(true)
            .secure(true)
            .path("/")
            .maxAge(Duration.ofDays(7))  //longer expiry for refresh token
            .sameSite("Lax")
            .build();

    return ResponseEntity
            .status(HttpStatus.CREATED)
            .header(HttpHeaders.SET_COOKIE, accessTokenCookie.toString())
            .header(HttpHeaders.SET_COOKIE, refreshTokenCookie.toString())
            .body(out.getOutput());
}
```

Figura 55: Método register() en UserService.java

- **invalidName(String name, AggregateOutput<?>out)**

  Valida que el nombre no sea vacío, tenga caracteres válidos (letras, números, espacios, guiones y apóstrofes) y longitud adecuada, registrando errores si no cumple.

- **invalidPassword(String password, AggregateOutput<?>out)**

  Valida que la contraseña no sea vacía, contenga solo caracteres alfanuméricos, al menos una letra mayúscula, una minúscula, un número, y tenga longitud entre 8 y 50 caracteres.

- **invalidEmail(String email, AggregateOutput<?>out)**

  Valida que el correo no sea vacío, tenga formato válido según expresión regular dada por OWASP[8] y no supere 150 caracteres.

- **manageRole(String admin, Long userId, boolean makeAdmin)**

  Permite a un administrador modificar el rol de otro usuario (promover o degradar). No permite modificar al usuario administrador por defecto ni a sí mismo. Guarda cambios y registra mensajes.

```java
public ResponseEntity<?> manageRole(String admin, Long userId, boolean makeAdmin) {
    AggregateOutput<String> out = new AggregateOutput<>();

    User targetUser = this.getUser(userId, out);
    if(targetUser == null){
        return out.toResponseEntity();
    }

    //default admin
    if(Objects.equals(targetUser.getEmail(), b: "admin@goal.tracker")){
        out.error( key: "default_admin_protected", text: "This user cannot be modified.");
        return out.setStatus(HttpStatus.NOT_FOUND).toResponseEntity();
    }

    if(Objects.equals(admin, targetUser.getEmail())){
        out.warning( key: "operation_not_allowed", text: "You cannot demote/promote yourself");
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }

    targetUser.setAdmin(makeAdmin);
    this.userRepository.save(targetUser);
     if(makeAdmin){
        out.info( key: "promoted_admin", text: "User successfully promoted to admin");
     }else{
        out.info( key: "demoted_admin", text: "User successfully demoted from admin");
     }
    return out.setStatus(HttpStatus.OK).toResponseEntity();
}
```

Figura 56: Método manageRole() en UserService.java

- **manageAccountStatus(String admin, Long userId, boolean ban)**

  Permite a un administrador banear o desbanear a otro usuario, con las mismas restricciones que en `manageRole`. Guarda cambios y registra mensajes.

```java
public ResponseEntity<?> manageAccountStatus(String admin, Long userId, boolean ban) {
    AggregateOutput<String> out = new AggregateOutput<>();

    User targetUser = this.getUser(userId, out);
    if(targetUser == null){
        return out.toResponseEntity();
    }

    //default admin
    if(Objects.equals(targetUser.getEmail(), b: "admin@goal.tracker")){
        out.error( key: "default_admin_protected", text: "This user cannot be modified.");
        return out.setStatus(HttpStatus.NOT_FOUND).toResponseEntity();
    }

    if(Objects.equals(admin, targetUser.getEmail())){
        out.warning( key: "operation_not_allowed", text: "You cannot ban/unban yourself");
        return out.setStatus(HttpStatus.BAD_REQUEST).toResponseEntity();
    }
    System.out.println(targetUser);

    targetUser.setBanned(ban);
    this.userRepository.save(targetUser);
    if(ban){
        out.info( key: "user_banned", text: "User successfully banned");
    }else{
        out.info( key: "user_unbanned", text: "User successfully unbanned");
    }

    return out.setStatus(HttpStatus.OK).toResponseEntity();
}
```

Figura 57: Método manageAccountStatus() en UserService.java

- **getUsers()**

  Lista todos los usuarios en el sistema, transformándolos a DTOs y agregándolos a la salida con mensaje de éxito.

```
372     public ResponseEntity<?> getUsers() {
373         AggregateOutput<UserResponse> out = new AggregateOutput<>();
374         List<User> users = userRepository.findAll();
375
376         List<UserResponse> list = new ArrayList<>();
377         for (User user: users) {
378             list.add(UserResponse.fromEntity(user));
379         }
380
381         out.setData(list);
382         out.info( key: "message", text: "Users succesfully retrieved",HttpStatus.OK);
383         return out.toResponseEntity();
384     }
385
```

Figura 58: Método getUsers() en UserService.java

- **getUser(Long userId, AggregateOutput<?>out)**

  Busca un usuario por ID. Si no existe, registra error y devuelve null.

- **validAction(User user, Long userId, AggregateOutput<?>out)**

  Valida que el usuario que realiza una acción tenga permiso para hacerlo, es decir, que sea administrador o que actúe sobre sí mismo. Registra errores y devuelve booleano.

- **logout(User user)**

  Invalida el token de refresco del usuario, estableciendo la fecha de expiración a ahora mismo restando una hora, guarda el usuario y devuelve mensaje de cierre de sesión exitoso. Esto solo es una parte del logout, luego el frontend tiene que decirle al navegador web que borre sus cookies con los tokens.

```
415 @    public ResponseEntity<?> logout(User user) {
416          AggregateOutput<String> out = new AggregateOutput<>();
417          user.setRefreshTokenExpiry(LocalDateTime.now().minusHours(1));
418          this.userRepository.save(user);
419          out.info( key: "logged_out", text: "Successfully logged out");
420          return out.setStatus(HttpStatus.OK).toResponseEntity();
421      }
```

Figura 59: Método logout() en UserService.java

4.1.9. utils/

La carpeta `utils/` contiene únicamente la clase `AggregateOutput`, una utilidad genérica que centraliza la estructura de salida de los endpoints de la API. Su principal función es encapsular tanto los datos de respuesta como los posibles mensajes de error, advertencia o información que puedan generarse durante el procesamiento de una solicitud.

Esta clase permite estandarizar las respuestas de la API siguiendo el siguiente formato:

Un campo `data` con una lista de objetos genéricos de tipo `T`.

Tres diccionarios: `errors`, `warnings` e `info`, que almacenan mensajes categorizados por nivel de severidad.

Un código de estado HTTP asociado a la respuesta.

Funcionamiento general: La clase mantiene internamente una estructura `outputDict` que clasifica los mensajes en tres niveles: error, advertencia e información. Estos mensajes se pueden añadir mediante los métodos `error()`, `warning()` e `info()`. Cada uno permite también, opcionalmente, establecer el código de estado HTTP de la respuesta.

Los datos principales se almacenan en una lista de tipo `T`, accesible mediante los métodos `append()` y `setData()`. Finalmente, la respuesta completa se genera con `getOutput()`, mientras que el método `toResponseEntity()` permite devolver directamente un `ResponseEntity` con el cuerpo estructurado y el código de estado correspondiente.

Ejemplo de uso típico: Un controlador puede construir una respuesta añadiendo elementos con `append()`, registrar advertencias o errores según sea necesario, y retornar directamente el resultado con `toResponseEntity()`, simplificando la lógica de respuesta y manteniendo consistencia en el formato. Además, permite verificar si se produjeron errores mediante `haveErrors()` para controlar la lógica interna del servicio.

4.1.10. src/main/resources/

La carpeta `src/main/resources/`, aunque no forma parte directoriol (`src/main/java/com/proje` es fundamental en cualquier proyecto Spring Boot. Contiene recursos estáticos y de configuración que el framework carga automáticamente al arrancar la aplicación.

En particular, aquí se encuentran dos archivos clave:

`application.properties`: define propiedades globales de configuración de Spring, como el nombre de la aplicación, el puerto del servidor, ajustes de JPA/Hibernate, y la conexión a la base de datos. Este archivo actúa como punto central donde se orquestan múltiples aspectos del comportamiento de la aplicación. Muchas de las variables aquí están referenciadas con `${...}`, lo que indica que sus valores reales se cargan desde el entorno o desde archivos adicionales.

`application-secrets.properties`: este archivo complementa al anterior, conteniendo variables sensibles como credenciales o contraseñas. No se incluye en el control de versiones por razones de seguridad. Al mantener las credenciales fuera del `application.properties`, se sigue una buena práctica de separación entre configuración general y datos confidenciales.

A continuación se muestra el contenido del archivo `application.properties`:

```
1  spring.application.name=goal-tracker
2
3  # === MySQL Database Configuration ===
4  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
5
6
7  # === JPA / Hibernate ===
8  spring.jpa.hibernate.ddl-auto=update
9  spring.jpa.show-sql=true
10 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
11
12
13 server.port=8000
14
15 spring.datasource.url=${SPRING_DATASOURCE_URL}
16 spring.datasource.username=${SPRING_DATASOURCE_USERNAME}
17 spring.datasource.password=${SPRING_DATASOURCE_PASSWORD}
18 app.admin.password=${ADMIN_PASSWORD}
19
```

Figura 60: Contenido del archivo `application.properties`

4.2. Frontend

El Frontend esta implementado en Python usando el framework Flask. A continuación, se describe la estructura de carpetas del directorio `goal-tracker/frontend` y su propósito:

- `app/`: contiene los ficheros con los endoints.
- `deployment_files/`: contiene los ficheros de despliegue necesarios para Kubernetes y Docker.
- `static/`: contiene recursos estáticos, como ficheros `.js` o `.css`.
- `templates/`: contiene ficheros `.html` dinámicos que contienen bloques de código de jinja.
- `venv/`: contiene el entorno virtual de python, usado exclusivamente para probar la aplicación de manera local.
- `__init__.py`: indica que el fichero actual (`goal-tracker/frontend`) es un paquete de Python. Está vacío.
- `main.py`: es el punto de entrada principal de la aplicación cuando se ejecuta localmente en modo desarrollo.
- `requirements.txt`: define las dependencias del proyecto, facilitando su instalación en nuevos entornos.

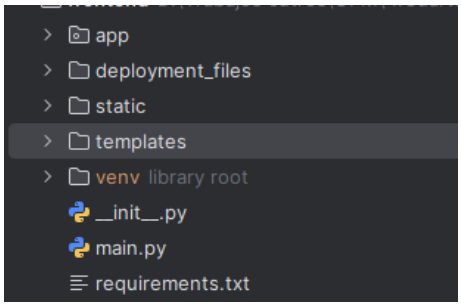


Figura 61: Los ficheros contenidos en `goal-tracker/frontend`

4.2.1. app/

La carpeta `app/` contiene toda la lógica relacionada con el renderizado dinámico de las vistas y la conexión con el backend. A continuación, se detallan sus componentes principales:

4.2.2. app/__init__.py

Este fichero no solo indica que `app/` es un paquete de Python, sino que también contiene la lógica necesaria para construir y devolver una instancia de la aplicación Flask. Para ello, registra las ubicaciones de los archivos estáticos (`static/`) y las plantillas HTML (`templates/`). Además, se integran los endpoints de la aplicación mediante el uso de `Blueprints`, una funcionalidad que permite organizar las rutas de manera modular y escalable. Finalmente, registra filtros definidos por el usuario para el motor de plantillas.

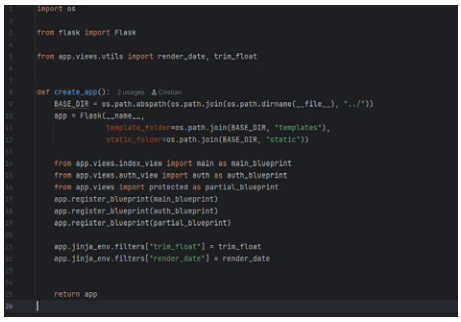


Figura 62: Contenido del fichero `__init__.py` en `app/`

4.2.3. app/config.py

Este fichero contiene configuraciones generales para la aplicación. En este caso, se define la variable `BACKEND_URL`, que se obtiene de las variables de entorno. Esto permite que la URL del backend sea fácilmente configurable sin modificar el código fuente.

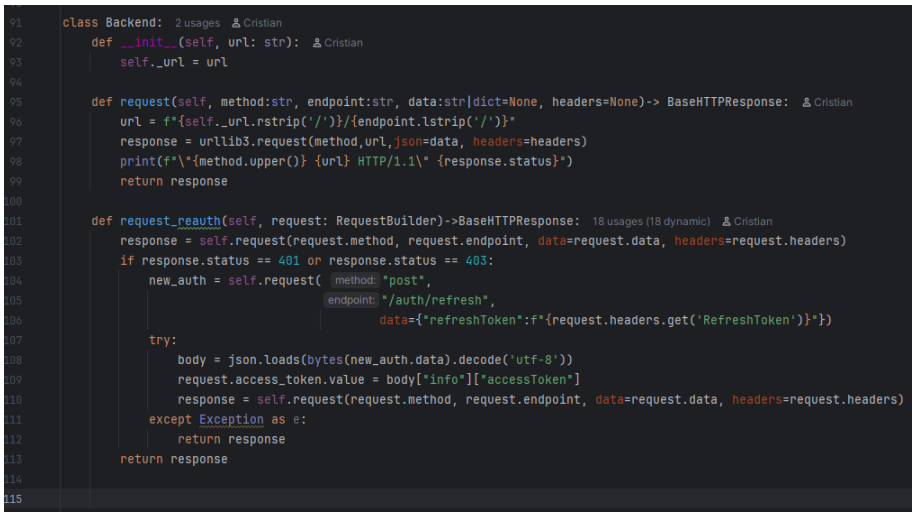
4.2.4. app/backend_client.py

Este fichero se encarga de centralizar la comunicación entre el frontend en Flask y el backend en Spring. Para ello, define una clase principal `Backend` que envía

peticiones HTTP mediante la librería `urllib3`. También incluye mecanismos para la autenticación y reautenticación automática cuando el token de acceso expira.

Además, se define un patrón singleton mediante la función `get_client()`, lo cual asegura que todas las peticiones usen una única instancia compartida del cliente backend. Por su parte, la clase `RequestBuilder` permite construir de forma estructurada las peticiones, facilitando la inclusión de headers, tokens y datos en formato JSON.

Un aspecto destacable de este módulo es que contiene lógica de reintento automático: si una petición obtiene una respuesta no autorizada (códigos 401 o 403), se realiza una petición al endpoint de refresco de tokens, y si esta tiene éxito, se reintenta la petición original con el nuevo token.



```
class Backend:  2 usages  Cristian
    def __init__(self, url: str):  Cristian
        self._url = url

    def request(self, method:str, endpoint:str, data:str|dict=None, headers=None)-> BaseHTTPResponse:  Cristian
        url = f"{self._url.rstrip('/')}/{endpoint.lstrip('/')}"
        response = urllib3.request(method,url,json=data, headers=headers)
        print(f"\"{method.upper()} {url} HTTP/1.1\" {response.status}")
        return response

    def request_reauth(self, request: RequestBuilder)->BaseHTTPResponse:  18 usages (18 dynamic)  Cristian
        response = self.request(request.method, request.endpoint, data=request.data, headers=request.headers)
        if response.status == 401 or response.status == 403:
            new_auth = self.request( method: "post",
                                     endpoint: "/auth/refresh",
                                             data={"refreshToken":f"{request.headers.get('RefreshToken')}"})
            try:
                body = json.loads(bytes(new_auth.data).decode('utf-8'))
                request.access_token.value = body["info"]["accessToken"]
                response = self.request(request.method, request.endpoint, data=request.data, headers=request.headers)
            except Exception as e:
                return response
        return response
```

Figura 63: La clase Backend en `backend_client.py` en `app/`

4.2.5. app/views/

Esta carpeta contiene todas las vistas necesarias. Tiene 7 ficheros: `__init__.py`, `auth_view.py`, `goals_view.py`, `index_view.py`, `progress_view.py`, `protected_view.py` y `utils.py`.

4.2.6. app/views/__init__.py

Este fichero sirve como un punto comun para cargar todos los endpoints protegidos. Se crea un Blueprint de Flask para que posteriormente los submodulos puedan

usar esa variable para registras las rutas. Finalmente se cargan todos los modulos.

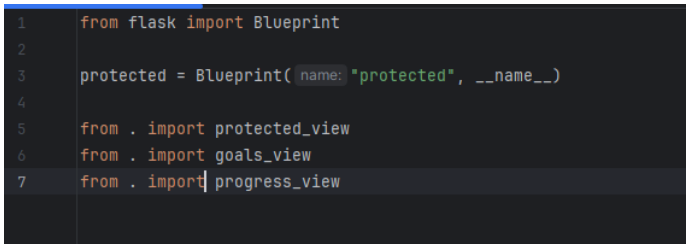


Figura 64: Fichero `__init__.py` en `app/views/`

4.2.7. app/views/auth_view.py

Este fichero contiene la vista encargada de gestionar toda la lógica de autenticación en el cliente. Utiliza el objeto `Blueprint` de Flask para agrupar las rutas bajo el nombre `'auth'` y permite login, registro y logout de usuarios. Internamente, utiliza el cliente definido en `backend_client.py` para comunicarse con el backend.

Rutas definidas

- `/login`: recibe una petición `POST` con credenciales de acceso. Esta es enviada al backend. Si la autenticación es exitosa, se establece el token de sesión en las cookies del navegador.

Figura 65: Método `login()` en `auth_view.py`

- `/register`: también recibe una petición `POST`, en este caso con los datos necesarios para registrar un nuevo usuario. La respuesta del backend contiene los tokens que se configuran en las cookies como en el login.

```python
@auth.route( rule: "/register", methods=['POST'])  Cristian
def register():
    data = request.get_json()
    client = get_client()
    response = client.request("POST", "/auth/register", data=data)

    return forward_token(response)
```

Figura 66: Método `register()` en `auth_view.py`

- `/logout`: limpia las cookies asociadas a la sesión del usuario (JWT y RefreshToken). Además, realiza una petición al backend para cerrar la sesión de forma segura.

```python
@auth.route( rule: "/logout", methods=["POST"])  Cristian
def logout():
    token = request.cookies.get('JWT')
    refresh = request.cookies.get('RefreshToken')
    if not refresh:
        return jsonify({"message":"Already logged out"}), 200

    client = get_client()

    client.request("POST", "/auth/logout")

    response = make_response(redirect("/"))

    # Clear the cookies (set Max-Age=0)
    response.set_cookie( key: "JWT",  value: "", max_age=0, path="/", httponly=True, secure=True, samesite='Lax')
    response.set_cookie( key: "RefreshToken",  value: "", max_age=0, path="/", httponly=True, secure=True, samesite='Lax')
    return response
```

Figura 67: Método `logout()` en `auth_view.py`

Función auxiliar `forward_token()` Esta función toma una respuesta HTTP del backend y la adapta a una respuesta válida para Flask. Se encarga de extraer las cookies `Set-Cookie` del backend y aplicarlas a la respuesta del frontend. También elimina los tokens del cuerpo de la respuesta por seguridad (Que no puedan ser accedidos por javascript).

```python
def forward_token(response: BaseHTTPResponse):  2 usages  Cristian
    resp_body_bytes = bytes(response.data)
    resp_body_str = resp_body_bytes.decode('utf-8')

    resp_body_dict = dict()
    try:
        # Parse JSON body into a Python dict
        resp_body_dict = json.loads(resp_body_str)
        resp_body_dict["info"].pop("accessToken", None)
        resp_body_dict["info"].pop("refreshToken", None)
    except json.JSONDecodeError:
        # Fall back if it's not JSON
        resp_body_dict["info"] = {"message": "Something went wrong"}

    flask_response = Response(
        response=json.dumps(resp_body_dict),
        status=response.status,
        mimetype='application/json'
    )

    set_cookies = response.headers.getlist("Set-Cookie")
    for cookie in set_cookies:
        flask_response.headers.add( key: "Set-Cookie", cookie)

    return flask_response
```

Figura 68: Función `forward_token()` en `auth_view.py`

4.2.8. app/views/index_view.py

Este fichero contiene la vista principal que renderiza la pantalla de inicio. Define una única ruta, asociada a la raíz del dominio, y decide si redirigir al usuario dependiendo de si tiene un token de sesión válido.

Rutas definidas

- `/`: Esta ruta muestra la página principal de inicio de sesión. Si el usuario ya tiene una cookie JWT válida, se consulta su perfil en el backend. En base a la respuesta, se decide si redirigir al área de administración o al panel de objetivos del usuario. En caso de no haber sesión activa, se renderiza directamente el fichero `index.html`.

```python
main = Blueprint( name: 'main', __name__)


@main.route("/")  Cristian
def login_form():
    token = request.cookies.get('JWT')
    if token:
        client = get_client()

        headers = {'Authorization': f'Bearer {token}'}
        resp = client.request("get", "user/profile", headers=headers)

        if resp.status == 200:
            user_id = json.loads(resp.data)["data"][0]["id"]
            is_admin = json.loads(resp.data)["data"][0]["isAdmin"]
            if is_admin:
                return redirect("/admin")
            else:
                return redirect(f'/user/{user_id}/goals')


    return render_template("index.html")

```

Figura 69: Ruta principal en `index_view.py`

4.2.9. app/views/goals_view.py

Este módulo contiene vistas específicas para la gestión de metas (*goals*). Permite a los usuarios visualizar, crear, editar, y eliminar metas, así como gestionar los avances relacionados a estas.

- **/user/<user_id>/goals (GET):** Muestra una lista con todas las metas del usuario. La vista también incluye información de progreso, mostrando el porcentaje alcanzado y si la meta ha sido completada.

```python
@protected.route( rule: "/user/<int:user_id>/goals",methods=["GET"])  Cristian
def goals_list(user_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')
    client = get_client()

    user_data,status =  profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]
    token_user_id = user_data.pop("id", None)

    target_user = dict()
    if token_user_id != user_id:
        target_user, token = target_profile(token,refresh, user_id)


    backend_request = ((((RequestBuilder()
                      .auth(token))
                      .refresh(refresh))
                      .set_method("get"))
                      .set_endpoint(f"/user/{user_id}/goals"))
    response = client.request_reauth(backend_request)

    if response.status in [401,403]:
        return redirect('/')

    goals = json.loads(response.data)["data"]
    filtered_goals = filter_goals_list(request.args, goals)

    resp = make_response(render_template( template_name_or_list: "goals.html",
                                         goals=filtered_goals,
                                         user_id=user_id,
                                         **user_data,
                                         **target_user))
    return create_response(resp, token)
```

Figura 70: Vista de listado de metas

- **/user/<user_id>/goals/create-form (GET, POST):** Muestra el formulario para la creación de una nueva meta. Si la solicitud es `POST` y válida,

se guarda la meta en el backend y el usuario es redirigido. Si hay errores, estos se muestran en pantalla vaciando los valores.

```python
@protected.route( rule: "/user/<int:user_id>/goals/create-form", methods=["GET", "POST"])  Cristian
def create_goal_form(user_id):
    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    if request.method == "POST":
        token = Token(request.cookies.get('JWT'))
        refresh = Token(request.cookies.get('RefreshToken'))
        if not refresh or not refresh.value:
            return redirect('/')

        client = get_client()

        backend_request = ((((RequestBuilder()
                              .auth(token))
                             .refresh(refresh))
                            .set_method("post"))
                           .set_endpoint(f"/user/{user_id}/goals")
                           .set_json({
                               "title": request.form["title"],
                               "description": request.form.get("description", ""),
                               "totalDesiredAmount": float(request.form["totalDesiredAmount"]),
                               "metric": request.form["metric"]
                           }))

        response = client.request_reauth(backend_request)

        if response.status == 403 or response.status == 401:
            return redirect('/')

        if response.status == 201:
            return redirect(f'/user/{user_id}/goals')

        errors = json.loads(response.data)["errors"]

        resp = render_template( template_name_or_list: "goal_create_form.html",
                               user_id=user_id,
                               **errors,
                               **user_data)

        return create_response(resp,token)

    return render_template( template_name_or_list: "goal_create_form.html",
                           user_id=user_id,
                           **user_data)
```

Figura 71: Formulario de creación de meta

- **/user/<user_id>/goals/<goal_id>(GET):** Presenta una vista detallada de una meta en específico, junto con todos los progresos relacionados. También permite acceder a formularios para editar la meta o registrar progreso.

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>", methods=["GET"])  Cristian
def single_goal(user_id, goal_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')
    client = get_client()


    backend_request = ((((RequestBuilder()
                          .auth(token))
                         .refresh(refresh))
                        .set_method("get"))
                       .set_endpoint(f"/user/{user_id}/goals/{goal_id}"))
    response = client.request_reauth(backend_request)

    if response.status in [401,403]:
        return redirect('/')
    goal = json.loads(response.data)["data"][0]

    backend_request.set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress")
    response = client.request_reauth(backend_request)
    progress_list = json.loads(response.data)["data"]

    filtered_progress = filter_single_goal(request.args, progress_list)

    total_amount = 0.0
    for p in progress_list:
        total_amount += p["amount"]

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]
    token_user_id = user_data.pop("id", None)

    target_user = dict()
    if token_user_id != user_id:

        profile_request = RequestBuilder()
        (profile_request
         .auth(token)
         .refresh(refresh)
         .set_method("get")
         .set_endpoint(f"/user/{user_id}/profile"))
        profile_response = client.request_reauth(profile_request)
        if profile_response.status == 200:
            body_profile = json.loads(profile_response.data)["data"][0]
            target_user["target_name"] = body_profile["name"]
            target_user["target_email"] = body_profile["email"]


    resp = make_response(render_template( template_name_or_list: "goal.html",
                                         goal=goal,
                                         progress_list=filtered_progress,
                                         progress_amount=total_amount,
                                         user_id=user_id,
                                         **user_data,
                                         **target_user))

    return create_response(resp, token)
```

Figura 72: Vista detallada de una meta

- **/user/<user_id>/goals/<goal_id>/update-form (GET, POST):** Carga un formulario para editar una meta existente. En caso de una petición `POST` válida, se actualiza la información en el backend. Si hay errores, se vuelve a renderizar el formulario con los mensajes correspondientes.

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>/update-form", methods=["GET"])  Cristian
def update_goal_form(user_id, goal_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    client = get_client()
    backend_request = ((((RequestBuilder()
                          .auth(token))
                         .refresh(refresh))
                        .set_method("get"))
                       .set_endpoint(f"/user/{user_id}/goals/{goal_id}"))
    response = client.request_reauth(backend_request)

    if response.status in [401, 403]:
        return redirect('/')

    goal = json.loads(response.data)["data"][0]

    resp = render_template( template_name_or_list: "goal_update_form.html",
                           goal=goal,
                           user_id=user_id,
                           **user_data,
                           goal_id=goal_id)
    return create_response(resp, token)
```

Figura 73: Formulario de actualización de meta

- **/api/user/<user_id>/goal/<goal_id>(DELETE):** Aunque no se usa directamente en la interfaz final, esta ruta permite eliminar una meta específica mediante una llamada a la API. En producción, todas las llamadas a `/api/` se redirigen automaticamente al backend, pero dependiendo del entorno de pruebas, esto no esta disponible.

```python
@protected.route( rule: "/api/user/<int:user_id>/goal/<int:goal_id>", methods=["DELETE"])  Cristian
def goal_delete(user_id, goal_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')

    client = get_client()

    backend_request = ((((RequestBuilder()
                      .auth(token))
                      .refresh(refresh))
                      .set_method(request.method))
                      .set_endpoint(f"/user/{user_id}/goals/{goal_id}"))

    response = client.request_reauth(backend_request)

    if response.status in [401,403]:
        return redirect('/')

    if response.status == 204:
        return "", response.status
    return json.loads(response.data), response.status
```

Figura 74: Eliminación de meta vía API

4.2.10. app/views/progress_view.py

Este módulo contiene las vistas necesarias para gestionar el progreso registrado sobre metas de usuario. Permite crear, editar y eliminar entradas de progreso, así como visualizar los detalles asociados a cada una de ellas. Todas las operaciones requieren autenticación mediante tokens.

- **/user/<user_id>/goals/<goal_id>/progress-form (GET, POST):** Esta ruta permite registrar una nueva entrada de progreso. Si la petición es `GET`, se muestra el formulario correspondiente. En caso de una petición `POST` válida, se envía la información al backend y se redirige al usuario. Si hay errores, se renderiza el formulario con los mensajes correspondientes.

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>/progress-form", methods=["GET", "POST"])  Cristian
def create_progress(user_id, goal_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')
    client = get_client()

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    backend_request = ((((RequestBuilder()
                          .auth(token))
                         .refresh(refresh))
                        .set_method("get"))
                       .set_endpoint(f"/user/{user_id}/goals/{goal_id}"))
    response = client.request_reauth(backend_request)
    if response.status != 200:
        return redirect(f"/user/{user_id}/goals")

    goal = json.loads(response.data)["data"][0]
    goal_title = goal["title"]

    if request.method == "POST":
        form_data = {
            "amount": float(request.form["amount"]),
            "updateNote": request.form["updateNote"]
        }

        backend_request = ((((RequestBuilder()
                              .auth(token))
                             .refresh(refresh))
                            .set_method("post"))
                           .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress")
                           .set_json(form_data))

        post_response = client.request_reauth(backend_request)

        body = json.loads(post_response.data)

        if post_response.status == 201:
            return redirect(f"/user/{user_id}/goals/{goal_id}")
        else:
            resp = render_template( template_name_or_list: "progress_create_form.html",
                                    goal_id=goal_id,
                                    goal_title=goal_title,
                                    user_id=user_id,
                                    **body["errors"],
                                    **user_data)
            return create_response(resp, token)

    resp = render_template( template_name_or_list: "progress_create_form.html",
                            goal_id=goal_id,
                            user_id=user_id,
                            goal_title=goal_title,
                            **user_data)
    return create_response(resp, token)
```

Figura 75: Formulario de creación de progreso

- **/user/<user_id>/goals/<goal_id>/progress/<progress_id>/update-form (GET):** Esta ruta permite actualizar una entrada de progreso ya existente. El formulario se carga con los datos actuales y cuando se rellena y se envia una petición `POST` se ocupa otro endpoint de aplicar los cambios. El diseño de separar los endpoints GET y POST en dos diferentes se hizo para evitar métodos grandes y con lógica mezclada.

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>/progress/<int:progress_id>/update-form", methods=["GET"])
def update_progress_form(user_id, goal_id, progress_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    client = get_client()
    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress/{progress_id}")
    )

    response = client.request_reauth(backend_request)

    if response.status in [401, 403]:
        return redirect('/')

    progress = json.loads(response.data).get("data", [{}])[0]

    resp = render_template( template_name_or_list: "progress_update_form.html",
                           progress=progress,
                           goal_id=goal_id,
                           user_id=user_id,
                           **user_data)

    return create_response(resp, token)

```

Figura 76: Formulario de actualización de progreso

- **/user/<user_id>/goals/<goal_id>/progress/<progress_id>/update-form (POST):** Esta ruta permite actualizar una entrada de progreso ya existente. Los datos provienen directamente del formulario, y en caso de fallo se envia el mismo formulario con los errores.

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>/progress/<int:progress_id>/update-form", methods=["POST"])
def update_progress(user_id, goal_id, progress_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    form_data = {
        "updateNote": request.form.get("updateNote"),
        "amount": float(request.form.get("amount")),
    }

    client = get_client()
    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("patch")
        .set_json(form_data)
        .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress/{progress_id}")
    )

    response = client.request_reauth(backend_request)

    if response.status in [200, 201]:
        return redirect(f"/user/{user_id}/goals/{goal_id}")
    elif response.status in [401, 403]:
        return redirect('/')

    get_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress/{progress_id}")
    )
    get_response = client.request_reauth(get_request)
    progress = json.loads(get_response.data).get("data", [{}])[0]

    body = json.loads(response.data)

    resp = render_template( template_name_or_list: "progress_update_form.html",
                           progress=progress,
                           goal_id=goal_id,
                           user_id=user_id,
                           **body["errors"],
                           **user_data)

    return create_response(resp, token)
```

Figura 77: Formulario de actualización de progreso

- **/api/user/<user_id>/goal/<goal_id>/progress/<progress_id>(DELETE):** Se usa unicamente para pruebas. En el despliegue en kubernetes todas las llamadas `/api/` se dirigen al backend directamente.

```python
@protected.route( rule: "/api/user/<int:user_id>/goal/<int:goal_id>/progress/<int:progress_id>", methods=["DELETE"])
def delete_progress(user_id, goal_id, progress_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect('/')

    client = get_client()

    backend_request = ((((RequestBuilder()
                      .auth(token))
                      .refresh(refresh))
                      .set_method("DELETE"))
                      .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress/{progress_id}"))

    response = client.request_reauth(backend_request)

    if response.status in [403,401]:
        return redirect('/')

    if response.status == 204:
        return "", response.status
    return json.loads(response.data), response.status
```

Figura 78: Eliminación de progreso vía API

4.2.11. app/views/protected_view.py

Este módulo define las rutas relacionadas con administración sumado a algunas funciones que se usan en otros lugares.

- **/profile (GET):** Obtiene el perfil del usuario autenticado. Valida el token y realiza una solicitud al backend para obtener los datos del perfil.

```python
@protected.route( rule: "/profile", methods=["GET"])  10 usages  Cristian
def profile():
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return jsonify({'error': 'unauthorized'}), 401

    client = get_client()

    backend_request = RequestBuilder()
    backend_request.auth(token).refresh(refresh).set_method("get").set_endpoint("/user/profile")
    response_b = client.request_reauth(backend_request)

    if response_b .status in [401,403]:
        return jsonify({'error': 'unauthorized'}), 401

    body = json.loads(response_b.data)

    return body, 200
```

Figura 79: Ruta /profile

- **target_profile(token, refresh, user_id):** Método auxiliar para obtener el perfil de otro usuario usando su ID. Devuelve nombre y correo electrónico. Se usa exclusivamente para los endpoints relacionados a administración.

```python
def target_profile(token:Token, refresh:Token, user_id):  1 usage  Cristian
    client = get_client()
    target_user = dict()
    profile_request = RequestBuilder()
    (profile_request
     .auth(token)
     .refresh(refresh)
     .set_method("get")
     .set_endpoint(f"/user/{user_id}/profile"))
    profile_response = client.request_reauth(profile_request)
    if profile_response.status == 200:
        body_profile = json.loads(profile_response.data)["data"][0]
        target_user["target_name"] = body_profile["name"]
        target_user["target_email"] = body_profile["email"]
    return target_user, profile_request.access_token
```

Figura 80: Función auxiliar target_profile

- **/admin (GET):** Muestra el panel de administración, con la posibilidad de buscar usuarios. Sólo accesible por usuarios con privilegios de administrador.

```python
@protected.route( rule: "/admin", methods=["GET"])  Cristian
def admin_dashboard():
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh or not refresh.value:
        return redirect("/")

    client = get_client()

    user_data, status = profile()
    if status in [401, 403]:
        return redirect('/')
    user_data = user_data["data"][0]

    if not user_data.get("isAdmin", False):
        return redirect(f"/user/{user_data['id']}/goals")

    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint("/user")
    )
    response = client.request_reauth(backend_request)

    if response.status != 200:
        return redirect("/")

    q = request.args.get('q', '').lower()

    users = json.loads(response.data)["data"]

    if q:
        pattern = re.compile(re.escape(q), re.IGNORECASE)
        filtered_users = [u for u in users if pattern.search(u['name']) or pattern.search(u['email'])  ]
    else:
        filtered_users = users

    resp = render_template( template_name_or_list: "admin_dashboard.html",
                           users=filtered_users,
                           **user_data)
    return create_response(resp, token)
```

Figura 81: Panel de administración

4.2.12. app/views/utils.py

Este módulo contiene funciones utilitarias que se emplean a lo largo de la aplicación para tareas de filtrado, transformación de datos y respuestas HTTP. Aunque no define rutas directamente, su funcionalidad es clave para la experiencia de usuario en vistas relacionadas con metas y progreso.

- **filter_goals_list(request_args, goals):** Filtra una lista de metas en función de un término de búsqueda (parámetro `q`) presente en el título de cada meta. Si no se proporciona ningún parámetro, devuelve la lista original sin filtrar.

```python
# pass request.args
def filter_goals_list(request_args, goals):  2 usages  Cristian
    q = request_args.get('q', '').lower()
    filtered_goals = goals
    if q:
        pattern = re.compile(re.escape(q), re.IGNORECASE)
        filtered_goals = [g for g in goals if pattern.search(g['title'])]

    return filtered_goals
```

Figura 82: Filtrado de metas por título

- **filter_single_goal(request_args, progress_list):** Filtra una lista de entradas de progreso asociadas a una meta. Permite filtrar por fecha específica (`date`) o por texto contenido en la nota del progreso (`q`). Además, ordena los resultados de forma descendente por fecha.

```python
def filter_single_goal(request_args, progress_list):  2 usages  Cristian
    date = request_args.get('date', '').lower()
    filtered_progress = progress_list

    if date:
        filtered_progress  = [p for p in progress_list if p["date"] == date]

    q = request_args.get('q', '').lower()
    if q:
        pattern = re.compile(re.escape(q), re.IGNORECASE)
        filtered_progress = [p for p in filtered_progress if pattern.search(p['updateNote'])]

    filtered_progress = sorted(filtered_progress, key=lambda p: p["date"], reverse=True)

    return filtered_progress
```

Figura 83: Filtrado de progreso por fecha y nota

- **render_date(date):** Convierte una fecha en formato `YYYY-MM-DD` a un formato legible para humanos (`DD/MM/YYYY`), usado en Jinja para renderizar correctamente la fecha.

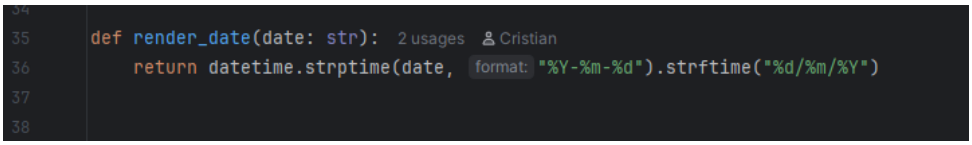


Figura 84: Conversión de fechas para visualización

- **trim_float(value):** Redondea valores numéricos decimales, eliminando decimales innecesarios si el valor es un entero, o limitando a dos decimales en caso contrario. Se emplea al mostrar cantidades en metas o progresos.

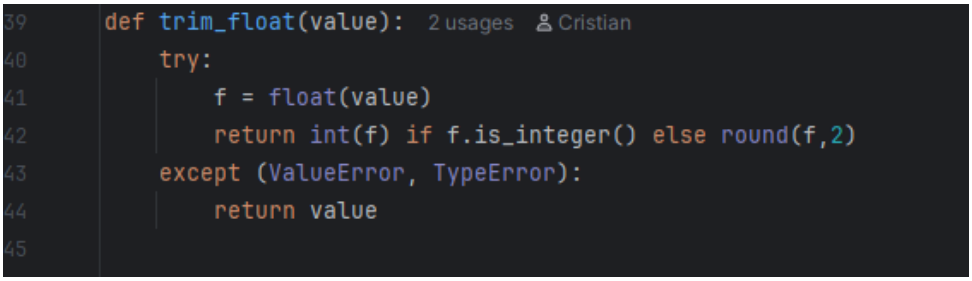


Figura 85: Formateo de números decimales

- **create_response(content, access_token):** Crea una respuesta HTTP e inserta el token JWT actualizado en la cookie del cliente. Esta función se reutiliza en múltiples rutas protegidas para mantener la sesión del usuario activa.

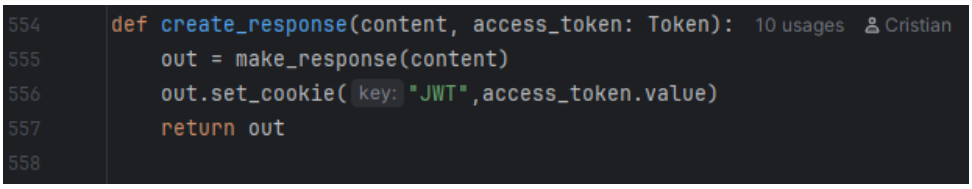


Figura 86: Generación de respuesta con token JWT

4.2.13. static/

Esta carpeta contiene todos los ficheros estáticos que se sirven en la página web. Solo se comentaran aquellos bajo la subcarpeta `js`, ya que la carpeta `css` contiene un fichero de estilo que no gran importancia y ayuda unicamente en la estética.

4.2.14. static/js/login.js

Este archivo contiene la lógica del cliente para manejar el proceso de inicio de sesión de usuarios. El script se ejecuta al cargarse el documento y realiza las siguientes funciones principales:

- Captura el formulario de login y escucha el evento `submit`.
- Valida que los campos de correo electrónico y contraseña no estén vacíos.
- Envía una solicitud `POST` al endpoint `/login` con las credenciales del usuario.
- Si la autenticación es exitosa, se realiza una consulta adicional a `/profile` para determinar si el usuario es administrador o regular.
- Redirige al panel de administración o al listado de metas del usuario, según corresponda.
- En caso de error (credenciales incorrectas o problemas de red), muestra mensajes adecuados en pantalla.
- Utiliza una función auxiliar `clearMessage()` para eliminar mensajes previos al nuevo intento de inicio de sesión.

```
document.addEventListener('DOMContentLoaded', () => {
  const form        = document.getElementById('login-form');
  const emailInput  = document.getElementById('login-email');
  const passInput   = document.getElementById('login-password');
  const responseDiv = document.getElementById('login-response');

  function clearMessage() {
    responseDiv.innerText = '';
    responseDiv.classList.add('d-none');
  }

  form.addEventListener('submit', async e => {
    e.preventDefault();
    clearMessage();

    const email    = emailInput.value.trim();
    const password = passInput.value;

    if (!email || !password) {
      responseDiv.classList.remove('d-none');
      responseDiv.style.color = 'red';
      responseDiv.innerText = 'Email and password must not be empty.';
      return;
    }

    try {
      const res = await fetch('/login', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ email, password }),
        credentials: 'include'
      });

      const data = await res.json();

      if (res.ok) {
        const res_id = await fetch('/profile', {
            method: 'GET',
            headers: { 'Content-Type': 'application/json' },
            credentials: 'include'
        });

        const body_res = await res_id.json();
        if (body_res.data[0].isAdmin){
            window.location.href = `/admin`;
        }else{
            window.location.href = `/user/${body_res.data[0].id}/goals`;
        }

      } else {
        const msg = data.message
          || (data.errors ? Object.values(data.errors).join('\n') : 'Login failed.');
        responseDiv.classList.remove('d-none');
        responseDiv.classList.add('alert', 'alert-danger');
        responseDiv.innerText = msg;
      }
    } catch (err) {
      console.error('Login error:', err);
      responseDiv.classList.remove('d-none');
      responseDiv.style.color = 'red';
      responseDiv.innerText = 'An error occurred. Please try again.';
    }
  });
});
```

Figura 87: Lógica general del archivo `login.js`

4.2.15. static/js/register.js

Este script implementa la lógica de registro de nuevos usuarios en el lado del cliente. Se ejecuta una vez que el documento ha sido cargado completamente y permite capturar la información del formulario, validarla y enviarla al servidor. Sus funcionalidades principales son:

- Captura los campos del formulario de registro: nombre, correo, contraseña y confirmación de contraseña.
- Valida que los campos estén completos antes de enviar los datos.
- Realiza una solicitud `POST` al endpoint `/register` con los datos ingresados por el usuario.
- Si el registro es exitoso (código 201), redirige al usuario al panel de administración (si es administrador) o al listado de metas personales.
- Si ocurre un error, analiza los campos con problemas y muestra los mensajes correspondientes en el lugar adecuado del formulario.
- Utiliza la función `clearErrors()` para limpiar los errores mostrados antes de cada intento de envío.

```
document.addEventListener('DOMContentLoaded', () => {
  const form             = document.getElementById('register-form');
  const nameInput        = document.getElementById('reg-name');
  const emailInput       = document.getElementById('reg-email');
  const passInput        = document.getElementById('reg-password');
  const confirmInput     = document.getElementById('reg-confirm-password');
  const nameErrDiv       = document.getElementById('name-error');
  const emailErrDiv      = document.getElementById('email-error');
  const passErrDiv       = document.getElementById('password-error');
  const confirmErrDiv    = document.getElementById('confirm-error');
  const generalErrDiv    = document.getElementById('general-error');

  function clearErrors() {
  [nameErrDiv, emailErrDiv, passErrDiv, confirmErrDiv, generalErrDiv]
    .forEach(d => {
      d.innerText = '';
      d.style.color = '';
      d.classList.add('d-none');
    });
}

  form.addEventListener('submit', async e => {
    e.preventDefault();
    clearErrors();

    const payload = {
      name:             nameInput.value.trim(),
      email:            emailInput.value.trim(),
      password:         passInput.value,
      confirmPassword:  confirmInput.value
    };

    try {
      const res  = await fetch('/register', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        credentials: 'include',
        body: JSON.stringify(payload)
      });
      const data = await res.json();

      if (res.status === 201) {
        const res_id = await fetch('/profile', {
            method: 'GET',
            headers: { 'Content-Type': 'application/json' },
            credentials: 'include'
        });

        const body_res = await res_id.json();
        if (body_res.data[0].isAdmin){
            window.location.href = `/admin`;
        }else{
            window.location.href = `/user/${body_res.data[0].id}/goals`;
        }
        return;
      }

      // on error: map fields
      if (data.errors) {
        Object.entries(data.errors).forEach(([key, msg]) => {
          if (key.startsWith('name_')) {
            nameErrDiv.style.color = 'red';
            nameErrDiv.innerText = msg;
            nameErrDiv.classList.remove('d-none');
          } else if (key.startsWith('email_')) {
            emailErrDiv.style.color = 'red';
            emailErrDiv.innerText = msg;
            emailErrDiv.classList.remove('d-none');
          } else if (key === 'passwords_dont_match' || key.startsWith('confirm')) {
            confirmErrDiv.style.color = 'red';
            confirmErrDiv.innerText = msg;
            confirmErrDiv.classList.remove('d-none');
          } else if (key.startsWith('password_')) {
            passErrDiv.style.color = 'red';
            passErrDiv.innerText = msg;
            passErrDiv.classList.remove('d-none');
          } else {
            generalErrDiv.style.color = 'red';
            generalErrDiv.innerText += msg + '\n';
            generalErrDiv.classList.remove('d-none');

          }
        });
      }
      if (data.info) {
        if (data.info.email_already_registered) {
          emailErrDiv.style.color = 'red';
          emailErrDiv.innerText = data.info.email_already_registered;
```

4.2.16. templates/

La carpeta `templates/` contiene los ficheros base de `.html` para generar dinamicamente las vistas y contiene 9 ficheros, `base.html`, `admin_dashboard.html`, `goal.html`, `goal_create_form.html`, `goal_update_form.html`, `goals.html`, `index.html`, `progress_update_form.html` y `progress_create_form.html`.

4.2.17. templates/base.html

Este archivo representa la plantilla base de toda la aplicación, sobre la cual se construyen las demás vistas HTML. Utiliza el sistema de plantillas de Jinja2 para permitir herencia de bloques, y adapta dinámicamente el contenido según el tipo de usuario (administrador o no). Las características principales de esta plantilla son:

- Define un diseño base con Bootstrap 5 para una apariencia aceptable.
- El título de la página y el encabezado principal cambian dinámicamente según si el usuario autenticado es administrador (`isAdmin`).
- Muestra la información del usuario autenticado (nombre y correo electrónico) en la barra superior.
- Incluye un botón de cierre de sesión que envía un formulario `POST` al endpoint `/logout`.
- Contiene bloques de Jinja2: `title`, `subtitle` y `content`, para que otras plantillas los sobreescriban con contenido específico.
- Agrega estilos personalizados en línea para mejorar la presentación de tarjetas (`.card`) y botones de acción.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>{% block title %}{% if isAdmin %}Admin Dashboard{% else %}Goal Dashboard{% endif %}{% endblock %}</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
  <style>
    body {
      background-color: #f8f9fa;
    }
    .dashboard-container {
      padding: 20px;
    }
    .card {
      box-shadow: 0 0.125rem 0.25rem rgba(0,0,0,.075);
      margin-bottom: 1rem;
      transition: transform 0.2s;
    }
    .card:hover {
      transform: translateY(-3px);
      box-shadow: 0 0.5rem 1rem rgba(0,0,0,.15);
    }
    .action-button {
      min-width: 90px;
      margin: 0.25rem;
    }
  </style>
</head>
<body>
  <!-- Top Bar -->
  <nav class="navbar navbar-expand-lg navbar-light bg-light border-bottom">
    <div class="container-fluid">
      <div class="user-info">
        <span class="navbar-brand">Welcome, {{ name }}</span>
        <span class="text-muted small">{{ email }}</span>
      </div>
      <form action="{{ url_for('auth.logout') }}" method="POST">
        <button type="submit" class="btn btn-outline-danger">Logout</button>
      </form>
    </div>
  </nav>

  <div class="container dashboard-container">
    <h1 class="mb-4">{% if isAdmin %}Admin Dashboard{% else %}Goal Dashboard{% endif %}</h1>
    {% block subtitle %}{% endblock %}
    <div id="app-view">
      {% block content %}{% endblock %}
    </div>
  </div>

  <!-- Bootstrap JS -->
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Figura 89: Estructura general de `base.html`

4.2.18. templates/admin_dashboard.html

Esta plantilla extiende de `base.html` y define la vista del panel de administración. Está compuesta por una sección de HTML estático que representa el contenido visual, y un bloque de JavaScript embebido al final, que maneja las acciones de interacción con los usuarios (banear, promover, etc.).

Sección HTML

- Contiene un formulario de búsqueda que permite filtrar usuarios.
- Muestra una lista de tarjetas por usuario con su nombre, correo, ID y etiquetas visuales si es `Admin` o está `Banned`.
- Cada tarjeta tiene botones de acción: `View Goals`, `Ban/Unban`, y `Promote/Demote`.
- Si no hay usuarios encontrados, se muestra una alerta informativa.

```
<!-- admin_dashboard.html -->
{% extends "base.html" %}

{% block content %}
<div class="container">
  <!-- Search Form -->
  <form method="get" action="{{ url_for('protected.admin_dashboard') }}" class="mb-4">
    <div class="input-group">
      <input type="text" name="q" class="form-control" placeholder="Search users..." value="{{ request.args.get('q', '') }}">
      <button class="btn btn-primary" type="submit">Search</button>
    </div>
  </form>

  <!-- User List -->
  {% if users %}
    {% for user in users %}
      <div class="card">
        <div class="card-body">
          <div class="d-flex justify-content-between align-items-center">
            <div>
              <h5 class="card-title">
                {{ user.name }}
                {% if user.admin %}<span class="badge bg-success">Admin</span>{% endif %}
                {% if user.banned %}<span class="badge bg-danger">Banned</span>{% endif %}
              </h5>
              <p class="card-text text-muted mb-1">{{ user.email }}</p>
              <p class="card-text"><small class="text-muted">User ID: {{ user.id }}</small></p>
            </div>
            <div>
              <a href="{{ url_for('protected.goals_list', user_id=user.id) }}" class="btn btn-sm btn-outline-primary me-1">
                View Goals
              </a>
              <button class="btn btn-sm btn-danger ban-toggle action-button"
                      data-user-id="{{ user.id }}"
                      data-action="{{ 'unban' if user.banned else 'ban' }}">
                {{ 'Unban' if user.banned else 'Ban' }}
              </button>
              <button class="btn btn-sm btn-warning promote-toggle action-button"
                      data-user-id="{{ user.id }}"
                      data-action="{{ 'demote' if user.admin else 'promote' }}">
                {{ 'Demote' if user.admin else 'Promote' }}
              </button>
            </div>
          </div>
        </div>
      </div>
    {% endfor %}
  {% else %}
    <div class="alert alert-info">No users found</div>
  {% endif %}
</div>

<script...>
{% endblock %}
```

Figura 90: Estructura HTML del panel de administración

Sección JavaScript

- Define la función `handleAction`, que se encarga de gestionar los clics en los botones de ban o promote.
- Solicita confirmación al usuario antes de hacer una petición `POST` a la API correspondiente.
- En caso de éxito, muestra un mensaje y recarga la página. En caso de error, muestra una alerta con el mensaje recibido o un error genérico.
- Se ejecuta para los selectores `.ban-toggle` y `.promote-toggle`.

```
<script>
  function handleAction(buttonClass, confirmMessageBase) {
    document.querySelectorAll(buttonClass).forEach(button => {
      button.addEventListener('click', async e => {
        e.preventDefault();
        const userId = button.getAttribute('data-user-id');
        const action = button.getAttribute('data-action');
        const confirmMsg = `${confirmMessageBase} this user?`;
        if (!confirm(confirmMsg)) return;

        try {
          const res = await fetch(`/api/user/${userId}/${action}`, {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json',
            },
            credentials: 'include'
          });

          const data = await res.json();
          console.log(data);

          if (res.ok) {
            const message = Object.values(data?.info || {})[0] || "Action completed successfully.";
            alert(message);

            window.location.reload();
          } else {
            const msg =
               Object.values(data?.errors || {})[0] ||
              "Action failed. Please try again.";
            alert(msg);
          }
        } catch (err) {
          console.error("Request failed:", err);
          alert("Unexpected error occurred while performing action.");
        }
      });
    });
  }

  handleAction('.ban-toggle', 'Are you sure you want to perform this action');
  handleAction('.promote-toggle', 're you sure you want to perform this action');
</script>
```

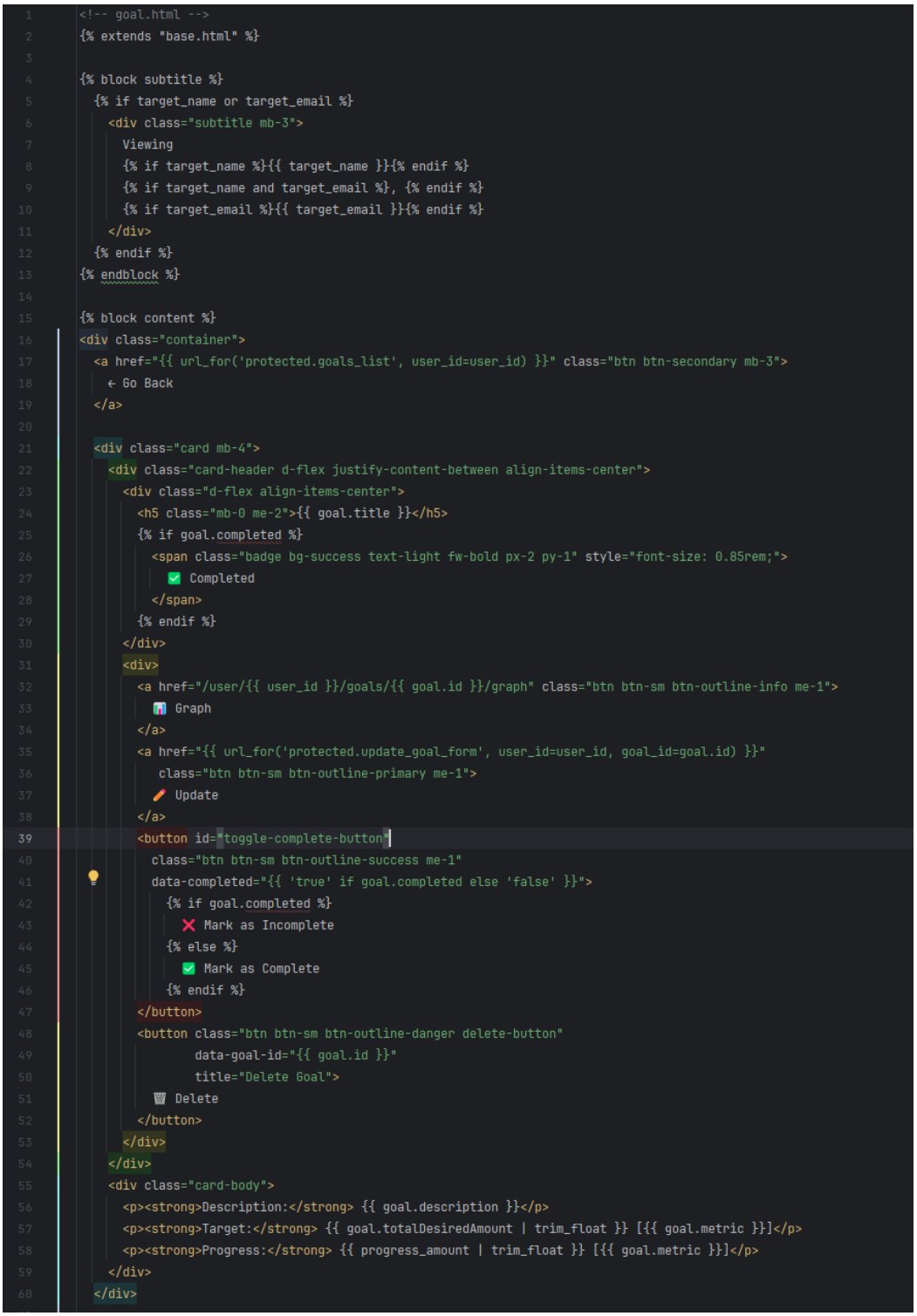
Figura 91: Script embebido para acciones administrativas (banear/promover)

4.2.19. templates/goal.html

Esta plantilla extiende de `base.html` y presenta la vista detallada de una meta individual. Está dividida en dos secciones principales: el encabezado que muestra la información de la meta, y una sección de entradas de progreso. Además, incluye tres bloques de scripts para manejar las acciones de eliminar metas, eliminar entradas de progreso, y marcar metas como completas/incompletas.

Sección HTML (1/2) — Información de la Meta

- Muestra un botón para regresar al listado de metas.
- Presenta el título de la meta, descripción, objetivo, progreso y un grupo de acciones: graficar, actualizar, marcar como completada o eliminar.
- Si la meta ya está completada, se muestra un distintivo verde `Completed`.



```
<!-- goal.html -->
{% extends "base.html" %}

{% block subtitle %}
  {% if target_name or target_email %}
    <div class="subtitle mb-3">
      Viewing
      {% if target_name %}{{ target_name }}{% endif %}
      {% if target_name and target_email %}, {% endif %}
      {% if target_email %}{{ target_email }}{% endif %}
    </div>
  {% endif %}
{% endblock %}

{% block content %}
<div class="container">
  <a href="{{ url_for('protected.goals_list', user_id=user_id) }}" class="btn btn-secondary mb-3">
    ← Go Back
  </a>

  <div class="card mb-4">
    <div class="card-header d-flex justify-content-between align-items-center">
      <div class="d-flex align-items-center">
        <h5 class="mb-0 me-2">{{ goal.title }}</h5>
        {% if goal.completed %}
          <span class="badge bg-success text-light fw-bold px-2 py-1" style="font-size: 0.85rem;">
            ✅ Completed
          </span>
        {% endif %}
      </div>
      <div>
        <a href="/user/{{ user_id }}/goals/{{ goal.id }}/graph" class="btn btn-sm btn-outline-info me-1">
          📊 Graph
        </a>
        <a href="{{ url_for('protected.update_goal_form', user_id=user_id, goal_id=goal.id) }}"
           class="btn btn-sm btn-outline-primary me-1">
          🖍️ Update
        </a>
        <button id="toggle-complete-button"
          class="btn btn-sm btn-outline-success me-1"
          data-completed="{{ 'true' if goal.completed else 'false' }}">
          {% if goal.completed %}
            ❌ Mark as Incomplete
          {% else %}
            ✅ Mark as Complete
          {% endif %}
        </button>
        <button class="btn btn-sm btn-outline-danger delete-button"
                data-goal-id="{{ goal.id }}"
                title="Delete Goal">
          🗑️ Delete
        </button>
      </div>
    </div>
    <div class="card-body">
      <p><strong>Description:</strong> {{ goal.description }}</p>
      <p><strong>Target:</strong> {{ goal.totalDesiredAmount | trim_float }} [{{ goal.metric }}]</p>
      <p><strong>Progress:</strong> {{ progress_amount | trim_float }} [{{ goal.metric }}]</p>
    </div>
  </div>
```

Figura 92: Vista superior de la meta individual

Sección HTML (2/2) — Entradas de Progreso

- Incluye un formulario de búsqueda para filtrar entradas de progreso.
- Si hay entradas, se listan en forma de tarjetas con fecha, cantidad y notas.
- Cada entrada tiene botones para editar o eliminar.
- Si no hay progresos, se muestra una alerta informativa.

```
<div class="d-flex justify-content-between align-items-center mb-3">
  <h4>Progress Entries</h4>
  <a href="{{ url_for('protected.create_progress', user_id=user_id, goal_id=goal.id) }}"
     class="btn btn-success">
    + Add Progress
  </a>
</div>

<!-- Search Form -->
<form method="get" action="{{ url_for('protected.single_goal', user_id=user_id, goal_id=goal.id) }}" class="mb-3">
  <div class="input-group">
    <input type="text" name="q" class="form-control" placeholder="Search progress..." value="{{ request.args.get('q', '') }}">
    <button class="btn btn-primary" type="submit">Search</button>
  </div>
</form>

{% if progress_list %}
  <div class="list-group">
    {% for progress in progress_list %}
      <div class="list-group-item">
        <div class="d-flex justify-content-between align-items-center mb-2">
          <div><strong>Date:</strong> {{ progress.date | render_date }}</div>
          <div>
            <a href="{{ url_for('protected.update_progress_form', user_id=user_id, goal_id=goal.id, progress_id=progress.id) }}"
               class="btn btn-sm btn-outline-primary me-1" title="Edit Progress">
              Edit
            </a>
            <button class="btn btn-sm btn-outline-danger progress-delete-button"
                    data-goal-id="{{ goal.id }}"
                    data-progress-id="{{ progress.id }}"
                    title="Delete Progress">
              Delete
            </button>
          </div>
        </div>
        <p class="mb-1"><strong>Amount:</strong> {{ progress.amount | trim_float }}</p>
        <p class="mb-0"><strong>Notes:</strong> {{ progress.updateNote }}</p>
      </div>
    {% endfor %}
  </div>
{% else %}
  <div class="alert alert-info">No progress entries available</div>
{% endif %}
</div>

<script...>
<script...>
<script...>

{% endblock %}
```

Figura 93: Entradas de progreso asociadas a la meta

Script (1/3) — Eliminar Meta

- Al hacer clic en el botón de eliminar, se solicita confirmación.
- Si el usuario acepta, se hace una petición DELETE a la API.
- Si es exitosa, redirige al listado de metas; si falla, muestra un mensaje de error.

```
<script>
  document.querySelector('.delete-button').addEventListener('click', async function (e) {
    e.preventDefault();
    const goalId = this.getAttribute('data-goal-id');
    const confirmDelete = confirm("Are you sure you want to delete this goal?");
    if (!confirmDelete) return;

    const userId = {{ user_id }};
    try {
      const res = await fetch(`/api/user/${userId}/goals/${goalId}`, {
        method: 'DELETE',
        headers: {
          'Content-Type': 'application/json',
        },
        credentials: 'include'
      });

      if (res.status === 204) {
        window.location.href = `/user/${userId}/goals`;
      } else {
        const data = await res.json();
        alert(data.message || "Failed to delete goal.");
      }
    } catch (err) {
      console.error("Delete failed:", err);
      alert("Error occurred while deleting.");
    }
  });
</script>
```

Figura 94: Script para eliminar una meta

Script (2/3) — Eliminar Entrada de Progreso

- Se aplica a todos los botones `.progress-delete-button`.
- Envía una petición DELETE al endpoint correspondiente de la entrada seleccionada.
- Recarga la página al completar o muestra un mensaje de error.

```
<script>
document.querySelectorAll('.progress-delete-button').forEach(button => {
  button.addEventListener('click', async function (e) {
    e.preventDefault();
    const goalId = this.getAttribute('data-goal-id');
    const progressId = this.getAttribute('data-progress-id');
    const confirmDelete = confirm("Delete this progress entry?");
    if (!confirmDelete) return;

    try {
      const userId = {{ user_id }};
      const res = await fetch(`/api/user/${userId}/goals/${goalId}/progress/${progressId}`, {
        method: 'DELETE',
        headers: {
          'Content-Type': 'application/json',
        },
        credentials: 'include'
      });

      if (res.status === 204) {
        window.location.reload();
      } else {
        const data = await res.json();
        alert(data.message || "Failed to delete progress.");
      }
    } catch (err) {
      console.error("Delete failed:", err);
      alert("Error occurred while deleting progress.");
    }
  });
});
</script>
```

Figura 95: Script para eliminar una entrada de progreso

Script (3/3) — Marcar Meta como Completa/Incompleta

- Detecta el estado actual de la meta usando un atributo `data-completed`.
- Envía una petición PATCH para actualizar el estado.
- Recarga la página si la operación fue exitosa; si falla, muestra un mensaje.

```
<script>
  document.getElementById('toggle-complete-button').addEventListener('click', async function (e) {
    e.preventDefault();

    const button = this;
    const userId = {{ user_id }};
    const goalId = {{ goal.id }};
    const currentlyCompleted = button.getAttribute('data-completed') === 'true';

    try {
      const res = await fetch(`/api/user/${userId}/goals/${goalId}`, {
        method: 'PATCH',
        headers: {
          'Content-Type': 'application/json',
        },
        credentials: 'include',
        body: JSON.stringify({ completed: !currentlyCompleted })
      });

      if (res.ok) {
        // Reload to reflect the updated status
        window.location.reload();
      } else {
        const data = await res.json();
        alert(data.message || "Failed to update completion status.");
      }
    } catch (err) {
      console.error("Completion toggle failed:", err);
      alert("Error occurred while updating goal status.");
    }
  });
</script>
```

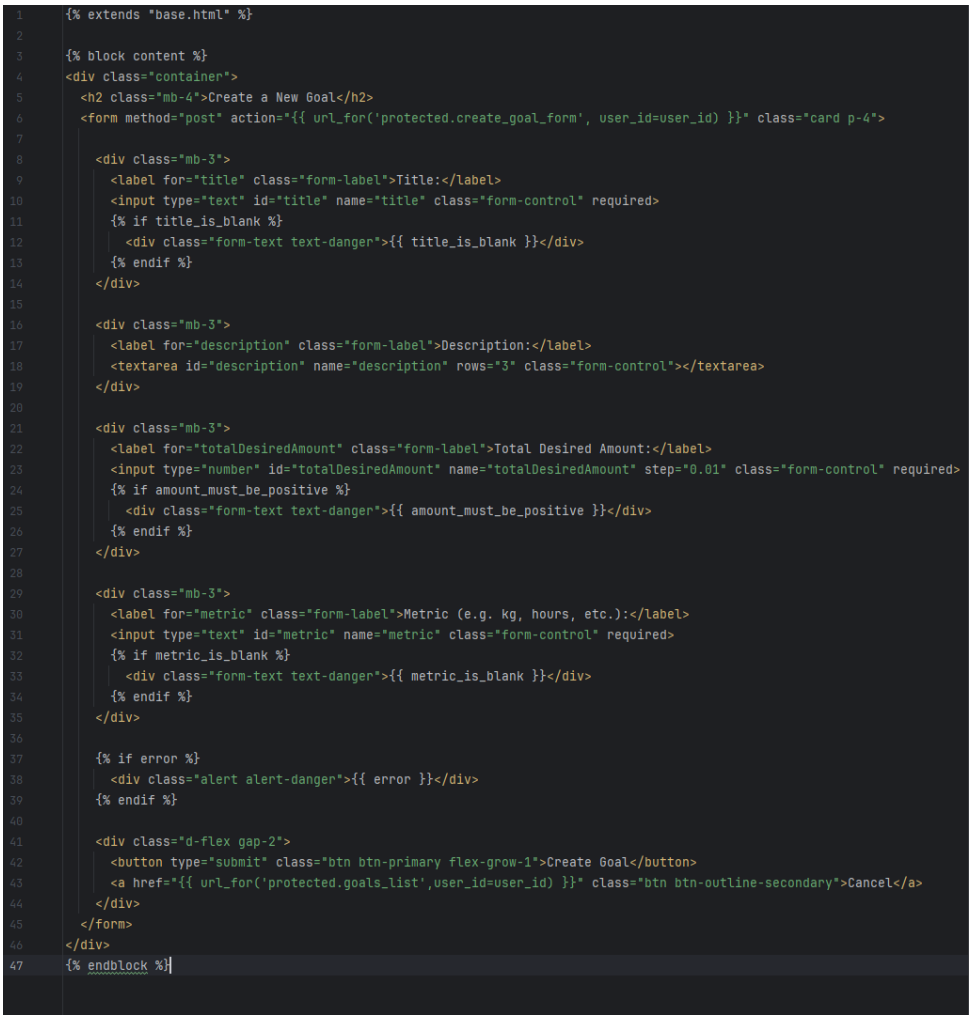
Figura 96: Script para alternar estado de completado de la meta

4.2.20. templates/goal_create_form.html

Esta plantilla corresponde al formulario de creación de una nueva meta. Se extiende de `base.html` y contiene una estructura simple pero funcional para que los usuarios ingresen los detalles de su nueva meta.

Contenido del Formulario

- **Título:** Campo obligatorio para el nombre de la meta.
- **Descripción:** Campo de texto libre opcional.
- **Cantidad Deseada:** Campo numérico obligatorio con validación de positividad.
- **Métrica:** Unidad de medida asociada a la meta (por ejemplo, "kg", "horas").
- **Errores:** Se muestran mensajes específicos si el título, cantidad o métrica son inválidos.
- **Botones:** Uno para enviar el formulario y otro para cancelar la operación y volver al listado de metas.



```
{% extends "base.html" %}

{% block content %}
<div class="container">
  <h2 class="mb-4">Create a New Goal</h2>
  <form method="post" action="{{ url_for('protected.create_goal_form', user_id=user_id) }}" class="card p-4">

    <div class="mb-3">
      <label for="title" class="form-label">Title:</label>
      <input type="text" id="title" name="title" class="form-control" required>
      {% if title_is_blank %}
        <div class="form-text text-danger">{{ title_is_blank }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="description" class="form-label">Description:</label>
      <textarea id="description" name="description" rows="3" class="form-control"></textarea>
    </div>

    <div class="mb-3">
      <label for="totalDesiredAmount" class="form-label">Total Desired Amount:</label>
      <input type="number" id="totalDesiredAmount" name="totalDesiredAmount" step="0.01" class="form-control" required>
      {% if amount_must_be_positive %}
        <div class="form-text text-danger">{{ amount_must_be_positive }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="metric" class="form-label">Metric (e.g. kg, hours, etc.):</label>
      <input type="text" id="metric" name="metric" class="form-control" required>
      {% if metric_is_blank %}
        <div class="form-text text-danger">{{ metric_is_blank }}</div>
      {% endif %}
    </div>

    {% if error %}
      <div class="alert alert-danger">{{ error }}</div>
    {% endif %}

    <div class="d-flex gap-2">
      <button type="submit" class="btn btn-primary flex-grow-1">Create Goal</button>
      <a href="{{ url_for('protected.goals_list',user_id=user_id) }}" class="btn btn-outline-secondary">Cancel</a>
    </div>
  </form>
</div>
{% endblock %}
```

Figura 97: Formulario de creación de una nueva meta

4.2.21. templates/goal_update_form.html

Esta plantilla permite al usuario editar una meta previamente creada. Al igual que el formulario de creación, se extiende de `base.html` y muestra un formulario precargado con los valores actuales de la meta.

Elementos del Formulario

- **Título:** Campo obligatorio con el valor actual ya completado.
- **Descripción:** Campo de texto prellenado con la descripción actual.
- **Cantidad Deseada:** Campo numérico obligatorio con el valor existente.
- **Métrica:** Campo de texto obligatorio con la unidad ya almacenada.
- **Mensajes de Validación:** Se despliegan en caso de errores como valores vacíos o inválidos.
- **Acciones:** Botón para confirmar los cambios y otro para cancelar y volver a la lista de metas.

```
{% extends "base.html" %}

{% block content %}
<div class="container">
  <h2 class="mb-4">Update Goal</h2>
  <form method="post" action="{{ url_for('protected.update_goal', user_id=user_id, goal_id=goal_id) }}" class="card p-4">

    <div class="mb-3">
      <label for="title" class="form-label">Title:</label>
      <input type="text" id="title" name="title" value="{{ goal.title }}" class="form-control" required>
      {% if title_is_blank %}
        <div class="form-text text-danger">{{ title_is_blank }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="description" class="form-label">Description:</label>
      <textarea id="description" name="description" rows="3" class="form-control">{{ goal.description }}</textarea>
    </div>

    <div class="mb-3">
      <label for="totalDesiredAmount" class="form-label">Total Desired Amount:</label>
      <input type="number" id="totalDesiredAmount" name="totalDesiredAmount" step="0.01" value="{{ goal.totalDesiredAmount }}" class="form-control" required>
      {% if amount_must_be_positive %}
        <div class="form-text text-danger">{{ amount_must_be_positive }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="metric" class="form-label">Metric (e.g. kg, hours, etc.):</label>
      <input type="text" id="metric" name="metric" value="{{ goal.metric }}" class="form-control" required>
      {% if metric_is_blank %}
        <div class="form-text text-danger">{{ metric_is_blank }}</div>
      {% endif %}
    </div>

    <div class="d-flex gap-2">
      <button type="submit" class="btn btn-success flex-grow-1">Update Goal</button>
      <a href="{{ url_for('protected.goals_list', user_id=user_id) }}" class="btn btn-outline-secondary">Cancel</a>
    </div>
  </form>
</div>
{% endblock %}

```

Figura 98: Formulario para actualizar una meta existente

4.2.22. templates/goals.html

Esta plantilla muestra un listado de metas del usuario. Está diseñada para facilitar la navegación, gestión y exportación de metas. Incluye funcionalidades de búsqueda, actualización, eliminación y generación de reportes en CSV o PDF. También muestra el nombre o correo del objetivo si está disponible.

Sección HTML

- Muestra un encabezado con título, botón para crear nuevas metas, y un menú desplegable con opciones de exportación.
- Incluye un formulario de búsqueda para filtrar metas por texto.
- Si existen metas, se muestran en tarjetas dentro de un grid. Cada tarjeta incluye título, descripción, meta deseada y botones para ver detalles, actualizar o eliminar.
- Si no hay metas, se muestra una alerta informativa.

```
{% extends "base.html" %}

{% block subtitle %}
  {% if target_name or target_email %}
    <div class="subtitle mb-3">
      Viewing
      {% if target_name %}{{ target_name }}{% endif %}
      {% if target_name and target_email %}, {% endif %}
      {% if target_email %}{{ target_email }}{% endif %}
    </div>
  {% endif %}
{% endblock %}

{% block content %}
<div class="container">
  {% if isAdmin %}
    <div class="mb-3">
      <a href="{{ url_for('protected.admin_dashboard') }}" class="btn btn-secondary">
        ← Go Back
      </a>
    </div>
  {% endif %}

  <div class="d-flex justify-content-between align-items-center mb-4">
    <h3>Your Goals</h3>
      <div class="d-flex align-items-center gap-2">
        <a href="{{ url_for('protected.create_goal_form', user_id=user_id) }}" class="btn btn-success">
          + New Goal
        </a>

        <!-- Dropdown -->
        <div class="position-relative">
          <button class="btn btn-light border dropdown-button" onclick="toggleDropdown()" title="More Options">⋮</button>
          <div id="dropdownMenu" class="dropdown-menu shadow rounded bg-white border p-2"
               style="display: none; position: absolute; right: 0; top: 100%; z-index: 1050;">
            <button onclick="exportCSV()" class="dropdown-item btn btn-link text-start w-100">Export as CSV</button>
            <button onclick="exportPDF()" class="dropdown-item btn btn-link text-start w-100">Export as PDF</button>
          </div>
        </div>
      </div>

  </div>

  <form method="get" action="{{ url_for('protected.goals_list', user_id=user_id) }}" class="mb-4">
    <div class="input-group">
      <input type="text" name="q" class="form-control" placeholder="Search goals..." value="{{ request.args.get('q', '') }}">
      <button class="btn btn-primary" type="submit">Search</button>
    </div>
  </form>

  {% if goals %}
    <div class="row">
      {% for goal in goals %}
        <div class="col-md-6 mb-4">
          <div class="card h-100">
            <div class="card-header d-flex justify-content-between align-items-center">
              <div class="d-flex align-items-center">
                <h5 class="mb-0 me-2">{{ goal.title }}</h5>
                {% if goal.completed %}
                  <span class="badge bg-success text-light fw-bold px-2 py-1" style="font-size: 0.85rem;">
                    ✅ Completed
                  </span>
                {% endif %}
              </div>
              <div>
                <a href="{{ url_for('protected.update_goal_form', user_id=user_id, goal_id=goal.id) }}"
                   class="btn btn-sm btn-outline-primary me-1">
                  ✏️ Update
                </a>
                <button class="btn btn-sm btn-outline-danger delete-button"
                        data-goal-id="{{ goal.id }}"
                        title="Delete Goal">
                  🗑️ Delete
                </button>
              </div>
            </div>
            <div class="card-body">
              <p class="card-text">{{ goal.description }}</p>
              <p class="card-text"><strong>Target:</strong> {{ goal.totalDesiredAmount | trim_float }} [{{ goal.metric }}]</p>
            </div>
            <div class="card-footer">
              <a href="/user/{{ user_id }}/goals/{{ goal.id }}" class="btn btn-primary w-100">
                View Details
              </a>
            </div>
          </div>
        </div>
      {% endfor %}
    </div>
  {% else %}
    <div class="alert alert-info">No goals found</div>
  {% endif %}
</div>
```

Script (1/2) — Eliminar Meta

- Aplica a todos los botones con la clase `.delete-button`.
- Solicita confirmación antes de eliminar.
- Realiza una petición DELETE al backend para eliminar la meta.
- Si se elimina exitosamente, recarga la página. En caso de error, muestra una alerta.

```
<script>
  document.querySelectorAll('.delete-button').forEach(button => {
    button.addEventListener('click', async e => {
      e.preventDefault();
      const goalId = button.getAttribute('data-goal-id');
      const confirmDelete = confirm("Are you sure you want to delete this goal?");
      if (!confirmDelete) return;

      try {
        const userId = {{ user_id }};
        const res = await fetch(`/api/user/${userId}/goals/${goalId}`, {
          method: 'DELETE',
          headers: {
            'Content-Type': 'application/json',
          },
          credentials: 'include'
        });

        if (res.status === 204) {
          window.location.reload();
        } else {
          const data = await res.json();
          alert(data.message || "Failed to delete goal.");
        }
      } catch (err) {
        console.error("Delete failed:", err);
        alert("Error occurred while deleting.");
      }
    });
  });
</script>
```

Figura 100: Script para eliminar metas del listado

Script (2/2) — Menú Desplegable y Exportación

- Gestiona la visibilidad del menú desplegable con opciones adicionales.
- Escucha clicks fuera del botón para cerrar el menú automáticamente.
- Incluye funciones para exportar las metas como archivos CSV o PDF, abriendo un enlace en una nueva pestaña.

```
<script>
  function toggleDropdown() {
    const menu = document.getElementById("dropdownMenu");
    menu.style.display = (menu.style.display === "block") ? "none" : "block";
  }

  window.addEventListener('click', function (e) {
    if (!e.target.matches('.dropdown-button')) {
      document.getElementById("dropdownMenu").style.display = "none";
    }
  });

    function exportCSV() {
      const userId = {{ user_id }};
      window.open(`/user/${userId}/csv-report`, '_blank');
    }
  z
    function exportPDF() {
      const userId = {{ user_id }};
      window.open(`/user/${userId}/pdf-report`, '_blank');
    }
</script>
```

Figura 101: Script para opciones de exportación de metas

4.2.23. templates/index.html

La plantilla `index.html` sirve como página de autenticación principal, permitiendo al usuario iniciar sesión o registrarse en la misma vista. Esta plantilla está dividida en dos partes fundamentales: la estructura HTML de los formularios y el script de alternancia que gestiona la visibilidad entre ambos formularios.

Estructura HTML El diseño se apoya en Bootstrap 5 para ofrecer una interfaz aceptable. Se incluyen dos formularios:

- **Formulario de Login**: Incluye campos para email y contraseña, un botón para enviar, y un mensaje de error condicional.
- **Formulario de Registro**: Contiene campos para nombre, email, contraseña, confirmación de contraseña y mensajes de validación específicos.

Ambos formularios están contenidos dentro de un `div.card` con estilos predefinidos. Inicialmente, solo el formulario de login está visible, mientras que el formulario de registro está oculto usando la clase `d-none`.

```html
<!-- index.html (login/register) -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Login / Register</title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
  <style...>
</head>
<body>
  <div class="container">
    <div class="auth-container card p-4">
      <h1 class="text-center mb-4" id="form-title">Login</h1>
      <!-- LOGIN FORM -->
      <form id="login-form">
        <div class="mb-3">
          <label for="login-email" class="form-label">Email</label>
          <input type="email" id="login-email" class="form-control" placeholder="Your email address" required>
        </div>

        <div class="mb-3">
          <label for="login-password" class="form-label">Password</label>
          <input type="password" id="login-password" class="form-control" placeholder="Enter your password" required>
        </div>

        <div id="login-response" class="alert alert-danger d-none mb-3"></div>

        <button type="submit" class="btn btn-primary w-100 mb-3">Login</button>
        <p class="text-center">
          Don't have an account?
          <a href="#" id="to-register" class="text-decoration-none">Register here</a>
        </p>
      </form>
      <!-- REGISTER FORM -->
      <form id="register-form" class="d-none">
        <div class="mb-3">
          <label for="reg-name" class="form-label">Name</label>
          <input type="text" id="reg-name" class="form-control" placeholder="Your full name">
          <div id="name-error" class="form-text text-danger d-none"></div>
        </div>

        <div class="mb-3">
          <label for="reg-email" class="form-label">Email</label>
          <input type="email" id="reg-email" class="form-control" placeholder="Your email address">
          <div id="email-error" class="form-text text-danger d-none"></div>
        </div>

        <div class="mb-3">
          <label for="reg-password" class="form-label">Password</label>
          <input type="password" id="reg-password" class="form-control" placeholder="Enter a password">
          <div id="password-error" class="form-text text-danger d-none"></div>
        </div>

        <div class="mb-3">
          <label for="reg-confirm-password" class="form-label">Confirm Password</label>
          <input type="password" id="reg-confirm-password" class="form-control" placeholder="Repeat your password">
          <div id="confirm-error" class="form-text text-danger d-none"></div>
        </div>

        <div id="general-error" class="alert alert-danger d-none mb-3"></div>

        <button type="submit" class="btn btn-primary w-100 mb-3">Register</button>
        <p class="text-center">
          Already have an account?
          <a href="#" id="to-login" class="text-decoration-none">Login here</a>
        </p>
      </form>
    </div>
  </div>
  <script src="{{ url_for('static', filename='js/login.js') }}"></script>
  <script src="{{ url_for('static', filename='js/register.js') }}"></script>
  <!-- toggle logic -->
  <script...>
</body>
</html>
```

Figura 102: Estructura de los formularios de login y registro en la plantilla index.html

Lógica de cambio Al final del archivo se incluye un bloque `<script>` que permite alternar entre el formulario de login y el de registro mediante el uso de JavaScript puro. Este comportamiento se logra a través de la manipulación de clases CSS (`d-none`) y la actualización del texto del título del formulario.

- Si se hace clic en el enlace Register here", el formulario de login se oculta y se muestra el formulario de registro.
- Si se hace clic en "Login here", se realiza la operación contraria.

```
<script>
  const loginForm = document.getElementById('login-form');
  const registerForm = document.getElementById('register-form');
  const formTitle = document.getElementById('form-title');

  document.getElementById('to-register').addEventListener('click', e => {
    e.preventDefault();
    loginForm.classList.add('d-none');
    registerForm.classList.remove('d-none');
    formTitle.innerText = 'Register';
  });

  document.getElementById('to-login').addEventListener('click', e => {
    e.preventDefault();
    registerForm.classList.add('d-none');
    loginForm.classList.remove('d-none');
    formTitle.innerText = 'Login';
  });
</script>
```

Figura 103: Script que permite alternar entre formularios en la vista index.html

4.2.24. templates/progress_create_form.html

La plantilla `progress_create_form.html` permite al usuario registrar un nuevo avance asociado a una meta específica. Es una vista sencilla, clara y centrada en la acción principal: añadir progreso.

Contenido de la vista El contenido se organiza mediante Bootstrap. La vista incluye:

- Un botón de navegación que permite regresar a la vista detallada de la meta correspondiente.
- Un encabezado dinámico que indica el título de la meta a la que se le agregará progreso.
- Un formulario con:
  - Campo numérico `amount` para registrar el avance realizado (con validación en caso de valores no positivos).
  - Área de texto `updateNote` para incluir una nota descriptiva del progreso.
  - Un botón para enviar el formulario.

```
<!-- progress_create_form.html -->
{% extends "base.html" %}

{% block content %}
<div class="container">
  <a href="{{ url_for('protected.single_goal', user_id=user_id , goal_id=goal_id) }}" class="btn btn-outline-secondary mb-3">
    ← Back to Goal
  </a>

  <h2 class="mb-4">Add Progress for: {{ goal_title }}</h2>

  <form method="post" action="{{ url_for('protected.create_progress', user_id=user_id, goal_id=goal_id) }}" class="card p-4">
    <div class="mb-3">
      <label for="amount" class="form-label">Amount:</label>
      <input type="number" step="0.01" id="amount" name="amount" class="form-control" required>
      {% if amount_must_be_positive %}
      <div class="form-text text-danger">{{ amount_must_be_positive }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="updateNote" class="form-label">Note:</label>
      <textarea id="updateNote" name="updateNote" rows="3" class="form-control"></textarea>
      {% if update_note_is_blank %}
      <div class="form-text text-danger">{{ update_note_is_blank }}</div>
      {% endif %}
    </div>

    <button type="submit" class="btn btn-success">Add Progress</button>
  </form>
</div>
{% endblock %}
```

Figura 104: Vista del formulario para registrar progreso asociado a una meta

4.2.25. templates/progress_update_form.html

La plantilla `progress_update_form.html` permite modificar un registro de progreso previamente creado. Esta vista reutiliza el estilo general de la aplicación con Bootstrap para mantener la coherencia visual.

Contenido de la vista El contenido principal consiste en un formulario que carga los datos actuales del progreso y permite editarlos. Incluye:

- Un título de encabezado que indica claramente que se está en la sección de actualización.
- Un mensaje de error general que puede mostrarse en la parte superior si hay un fallo al enviar el formulario.
- Dos campos de entrada:
  - `amount`: valor numérico del progreso realizado, precargado con el dato actual.
  - `updateNote`: nota del usuario asociada a ese progreso, también precargada.

- Validaciones condicionales que muestran errores en rojo si los datos no cumplen los requisitos.
- Un botón para confirmar la actualización y otro para cancelar y volver a la vista de la meta.

```
<!-- progress_update_form.html -->
{% extends "base.html" %}

{% block content %}
<div class="container">
  <h2 class="mb-4">Update Progress</h2>

  {% if error %}
    <div class="alert alert-danger mb-3">{{ error }}</div>
  {% endif %}

  <form method="post" action="{{ url_for('protected.update_progress', user_id=user_id , goal_id=goal_id, progress_id=progress.id) }}" class="card p-4">
    <div class="mb-3">
      <label for="amount" class="form-label">Amount:</label>
      <input type="number" id="amount" name="amount" step="0.01" value="{{ progress.amount }}" class="form-control" required>
      {% if amount_must_be_positive %}
      <div class="form-text text-danger">{{ amount_must_be_positive }}</div>
      {% endif %}
    </div>

    <div class="mb-3">
      <label for="updateNote" class="form-label">Note:</label>
      <textarea id="updateNote" name="updateNote" rows="3" class="form-control">{{ progress.updateNote }}</textarea>
      {% if update_note_is_blank %}
      <div class="form-text text-danger">{{ update_note_is_blank }}</div>
      {% endif %}
    </div>

    <div class="d-flex gap-2">
      <button type="submit" class="btn btn-success flex-grow-1">Update Progress</button>
      <a href="{{ url_for('protected.single_goal', user_id=user_id, goal_id=goal_id) }}" class="btn btn-outline-secondary">Cancel</a>
    </div>
  </form>
</div>
{% endblock %}
```

Figura 105: Vista del formulario para actualizar un registro de progreso

4.3. Graphing Service

El Graphing Service está implementado en Python usando el framework Flask y la librería Plotly. Este microservicio tendrá como función crear una gráfica de un mes en concreto, mostrando la cantidad de progreso por cada día. A continuación, se describe la estructura de carpetas del directorio `goal-tracker/graph_microservice` y su propósito:

- `graph/`: contiene los ficheros con los endoints.
- `deployment_files/`: contiene los ficheros de despliegue necesarios para Kubernetes y Docker.
- `static/`: guarda ficheros estáticos, como `.js` y `.css`
- `templates/`: contiene ficheros `.html` dinámicos que contienen bloques de código de jinja.
- `venv/`: contiene el entorno virtual de python, usado exclusivamente para probar la aplicación de manera local.
- `main.py`: es el punto de entrada principal de la aplicación cuando se ejecuta localmente en modo desarrollo.
- `requirements.txt`: define las dependencias del proyecto, facilitando su instalación en nuevos entornos.

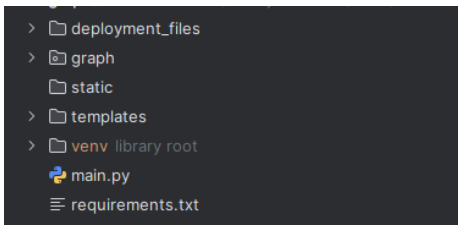


Figura 106: Los ficheros contenidos en `goal-tracker/graph_microservice`

4.3.1. graph/

La carpeta `graph/` contiene la lógica necesaria para renderizar en un gráfico los datos. Esta compuesta de 4 ficheros: `__init__.py`, `backend_client.py`, `graph_view.py` y `config.py` Realmente los ficheros `backend_client.py` y `config.py` son iguales que en el `goal-tracker/frontend` y su explicación será omitida.

4.3.2. graph/__init__.py

Este fichero es muy similar al que se encuentra en la carpeta de `goal-tracker/frontend/app` y tiene la misma función, tener una función `create_app()` que permita generar la aplicación de Flask con las rutas registradas.

```python
import os

from flask import Flask


def create_app():  2 usages  Cristian
    BASE_DIR = os.path.abspath(os.path.join(os.path.dirname(__file__), "../"))
    app = Flask(__name__,
                template_folder=os.path.join(BASE_DIR, "templates"),
                static_folder=os.path.join(BASE_DIR, "static"))
    from graph.graph_view import protected as protected_blueprint

    app.register_blueprint(protected_blueprint)


    return app
```

Figura 107: Contenido del fichero __init__.py en `graph/`

4.3.3. graph/graph_view.py

El fichero `graph_view.py` consiste de un único endpoint, el cual gestiona la renderización.

`render_graph` Este método es extenso, pero su explicación se puede dividir por secciones. Para facilitar la lectura y comprensión, se ha dividido el código en fragmentos mostrados en las figuras 108, 109 y 110.

En este primer fragmento se muestra la definición del endpoint `render_graph`, que gestiona las solicitudes GET para visualizar el progreso de un objetivo de un usuario. También se comprueba la validez de los tokens de autenticación. Si estos son

```python
@protected.route( rule: "/user/<int:user_id>/goals/<int:goal_id>/graph", methods=["GET"])  Cristian
def render_graph(user_id, goal_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh.value:
        return redirect('/')

    user_data, _ = profile(token, refresh)
    user_data = user_data["data"][0]

    client = get_client()

    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress")
    )
    response = client.request_reauth(backend_request)

    if response.status in [401, 403]:
        return redirect('/')

    progress_list = json.loads(response.data)["data"]
    progress_list.sort(key=lambda x: x["date"])

    selected_month = request.args.get("month")
    if selected_month is None:
        selected_month = datetime.now().strftime("%Y-%m")

    filtered_progress = [p for p in progress_list if p['date'].startswith(selected_month)]

    year, month = map(int, selected_month.split("-"))
    num_days = calendar.monthrange(year, month)[1]
    all_dates = [date(year, month, day) for day in range(1, num_days + 1)]
```

Figura 108: Código desde la línea 15 hasta la línea 50: Definición del endpoint `render_graph`.

correctos, el microservicio solicita la lista de progreso, filtrando todas las entradas de progreso para que únicamente queden aquellas que coincidan con el mes elegido. En caso de que no haya un mes elegido, se usa el mes actual. Posteriormente se inicializa la lista `all_dates`, el cual contiene todos los días para un cierto mes, de esta manera el gráfico se ajusta al número de días de un mes en específico.

```
52      if not filtered_progress:
53          if request.args.get("partial") == "true":
54              fig = go.Figure()
55              fig.update_layout(
56                  xaxis=dict(showgrid=False, zeroline=False, visible=False),
57                  yaxis=dict(showgrid=False, zeroline=False, visible=False),
58                  annotations=[dict(
59                      text="No data for the month picked",
60                      xref="paper", yref="paper",
61                      showarrow=False,
62                      font=dict(size=20)
63                  )]
64              )
65              graph_json = json.loads(fig.to_json())
66              return jsonify(graph_json)
67          else:
68              return render_template(
69                  template_name_or_list: "graph_template.html",
70                  graph_html="",
71                  **user_data,
72                  user_id=user_id,
73                  goal_id=goal_id
74              )
75
76      date_amount_map = {}
77      for p in filtered_progress:
78          d = datetime.strptime(p["date"], format: "%Y-%m-%d").date()
79          date_amount_map[d] = date_amount_map.get(d, 0) + p["amount"]
80
81      amounts = [date_amount_map.get(d, 0) for d in all_dates]
82
83      fig = go.Figure()
84      fig.add_trace(go.Bar(
85          x=all_dates,
86          y=amounts,
87          name='Progress',
88          hovertemplate='Date: %{x|%Y-%m-%d}<br>Amount: %{y}<extra></extra>'
89      ))
90
91      fig.update_xaxes(
92          type='date',
93          tickformat="%b %d",
94          dtick=86400000,
95          range=[all_dates[0], all_dates[-1]]
96      )
97
98      max_amount = max(amounts) if amounts else 1
99      fig.update_yaxes(range=[0, max_amount * 1.1])
```

Figura 109: Código desde la línea 52 hasta la línea 99: Obtención y filtrado de los datos de progreso.

En este segundo bloque se comprueba si existe entradas de progreso filtradas, en caso de que no haya ninguna, se devuelve simplmente un texto informando al usuario de que no hay datos para el mes seleccionado. La razón por la que hay dos posibilidades es porque la vista puede pedir una actualización parcial, lo que implica que el plotly ya está cargado en el navegador. Esto obliga a cambiar la manera en la que se muestra la falta de datos. Después de eso tenemos la lógica de agrupar por fecha, esto se debe a que pueden existir varias entradas de progreso en un mismo día. La solución es sumar todas en un mismo día. Finalmente se crea la figura, y se configura

sus opciones para que el gráfico representado sea coherente.

```
101     if request.args.get("partial") == "true":
102         graph_json = json.loads(fig.to_json())
103         return jsonify(graph_json)
104
105     graph_html = ""
106
107     resp = make_response(render_template(
108         template_name_or_list: "graph_template.html",
109         graph_html=graph_html,
110         **user_data,
111         user_id=user_id,
112         goal_id=goal_id,
113         STATIC_URL=STATIC_URL
114     ))
115     resp.set_cookie( key: "JWT", token.value)
116     return resp
```

Figura 110: Código desde la línea 100 hasta la línea 123: Creación y configuración del gráfico, y renderización final.

Finalmente, se envía los datos del gráfico si la request es `?partial=true`, de lo contrario se envia la plantilla con el gráfico vacío, dejando que el cliente posteriormente pida los datos.

4.3.4. templates/

La carpeta `templates/` contiene los ficheros base de `.html` para generar dinamicamente las vistas y contiene 2 ficheros, `base.html` y `graph_template.html`.

4.3.5. templates/base.html

Este fichero `.html` define la estructura general de la interfaz web, funcionando como una plantilla base que será heredada por otras plantillas, como `graph_template.html`. Utiliza `Bootstrap 5` para el diseño y define una cabecera con información del usuario, junto con una estructura de bloques reutilizables mediante Jinja2 (`{% block ... %}`).

La cabecera HTML define metadatos y estilos que aplican a toda la vista. Se importa una hoja de estilos local (`style.css`).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Progress Chart</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-light border-bottom">
    <div class="container-fluid">
      <div class="user-info">
        <span class="navbar-brand">Welcome, {{ name }}</span>
        <span class="text-muted small">{{ email }}</span>
      </div>
      <form action="/logout" method="POST">
        <button type="submit" class="btn btn-outline-danger">Logout</button>
      </form>
    </div>
  </nav>

  <div class="container dashboard-container">
    <h1 class="mb-4">Progress Chart</h1>
    {% block subtitle %}{% endblock %}
    <div id="app-view">
      {% block content %}{% endblock %}
    </div>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Figura 111: Estructura del fichero `base.html`

En el `body`, la barra de navegación muestra el nombre y correo electrónico del usuario (variables dinámicas proporcionadas por Flask) y un botón de logout. El contenido principal se encuentra dentro de un contenedor `dashboard-container`, donde se definen dos bloques: `subtitle` y `content`. Estos bloques permiten que otras plantillas hereden de `base.html` y añadan su propio contenido, manteniendo una estructura uniforme y reutilizable en toda la aplicación.

4.3.6. templates/graph_template.html

Este fichero hereda de `base.html` y define el contenido específico para mostrar el gráfico de progreso de un objetivo.

La plantilla incluye:

- Un botón `Go Back` que redirige a la vista del objetivo del usuario.
- Un formulario con selectores para elegir año y mes, que permite filtrar el gráfico por fecha. Este formulario está diseñado con Bootstrap para mantener una buena experiencia visual y responsiva.
- Un contenedor donde se inserta dinámicamente el código HTML del gráfico

```
{% extends "base.html" %}

{% block content %}
<div>
  <a href="/user/{{ user_id }}/goals/{{ goal_id }}" class="btn btn-secondary mb-3">
    ← Go Back
  </a>

  <div class="card shadow-sm mb-4">
    <div class="card-body">
      <form class="row g-2 align-items-end">
        <div class="col-auto">
          <label for="yearSelect" class="form-label mb-0">Pick month:</label>
        </div>
        <div class="col-auto">
          <select id="yearSelect" name="year" class="form-select"></select>
        </div>
        <div class="col-auto">
          <select id="monthSelect" name="month" class="form-select"...>
        </div>
        <div class="col-auto">
          <button id="applyMonthBtn" type="button" class="btn btn-primary">Search</button>
        </div>
      </form>
    </div>
  </div>

  <div id="graph-container" style="min-height: 400px;">
  {{ graph_html|safe }}
  </div>
</div>

<!-- Load Plotly library -->
<script src="https://cdn.plot.ly/plotly-latest.min.js"></script>

<script src="{{ STATIC_URL }}/js/progress.js"></script>


<script>
  // Pass user_id and goal_id as global variables for JS usage
  window.userId = {{ user_id }};
  window.goalId = {{ goal_id }};
</script>
{% endblock %}
```

Figura 112: Fragmento del fichero `graph_template.html`.

generado en Python y pasado como variable `graph_html`. Se usa la función `safe` de Jinja2 para renderizarlo como código HTML, no texto.

- Inclusión de la librería Plotly para renderizar los gráficos interactivos.
- Carga de un fichero JavaScript propio (`progress.js`) que contiene la lógica para manejar la interacción del selector de fecha y actualización dinámica del gráfico.
- Variables globales `userId` y `goalId` expuestas para uso del JavaScript, facilitando la construcción de URLs.

4.3.7. static/js/progress.js

Este script JavaScript es responsable de la interactividad en la página del gráfico de progreso:

- Inicializa los selectores de mes y año con el mes y año actuales, agregando opciones para los últimos 5 años y los próximos 5 años.

```
const currentYear = new Date().getFullYear();
for (let y = currentYear - 5; y <= currentYear + 5; y++) {
  const option = document.createElement('option');
  option.value = y;
  option.textContent = y;
  if (y === currentYear) option.selected = true;
  yearSelect.appendChild(option);
}
const currentMonth = String(new Date().getMonth() + 1).padStart(2, '0');
monthSelect.value = currentMonth;
```

Figura 113: Fragmento del fichero `js/progress.js`.

- Define la función `fetchAndRenderGraph`, que se ejecuta al pulsar el botón "Search". Esta función:

```
function fetchAndRenderGraph(yearMonth) {
  const url = `/user/${window.userId}/goals/${window.goalId}/graph?month=${yearMonth}&partial=true`;

  fetch(url)
    .then(res => {
      if (!res.ok) throw new Error('Failed to fetch graph JSON');
      return res.json();
    })
    .then(graphJson => {
      const config = {
        staticPlot: false,
        modeBarButtonsToRemove: [
          'lasso2d', 'select2d', 'zoom2d',
          'zoomIn2d', 'zoomOut2d', 'autoScale2d', 'pan2d'
        ],
        displayModeBar: false
      };

      Plotly.react(graphContainer, graphJson.data, {
        ...graphJson.layout,
        yaxis: {
          ...graphJson.layout.yaxis,
          fixedrange: true,
          range: graphJson.layout.yaxis?.range || [0, 1]
        },
        xaxis: {
          ...graphJson.layout.xaxis,
          fixedrange: true,
        },
        dragmode: false,
      }, config);

      graphContainer.on('plotly_click', (eventData) => {
        if (!eventData.points || eventData.points.length === 0) return;
        const clickedDate = eventData.points[0].x;
        window.location.href = `/user/${window.userId}/goals/${window.goalId}?date=${clickedDate}`;
      });
    })
    .catch(err => {
      console.error(err);
      graphContainer.innerHTML = `<p class="text-danger">Failed to load graph data.</p>`;
    });
}

// Initial load
fetchAndRenderGraph(`${yearSelect.value}-${monthSelect.value}`);

applyBtn.addEventListener('click', () => {
  fetchAndRenderGraph(`${yearSelect.value}-${monthSelect.value}`);
});
});
```

Figura 114: Fragmento del fichero `js/progress.js`.

- Obtiene el mes y año seleccionados.

    - Construye una URL para hacer una petición `fetch` con los parámetros seleccionados y `partial=true` para obtener sólo los datos del gráfico en formato JSON.
    - Recibe la respuesta, que contiene el gráfico en formato JSON, y usa `Plotly.react` para actualizar dinámicamente el gráfico en el contenedor `graph-container` sin recargar la página al completo.

- Añade un manejador de eventos para detectar clics en las barras del gráfico. Cuando se hace clic en un punto del gráfico, redirige a una URL específica del usuario y objetivo, pasando la fecha del punto clicado para mostrar detalles relacionados. Esto permite ver el progreso en un día particular.

4.4. Export Service

El Export Service esta implementado en Python usando el framework Flask y la librería reportlab para poder producir pdf de manera automatica. Este microservicio tendrá como función crear un reporte de tipo csv o pdf, con todos los datos del usuario. A continuación, se describe la estructura de carpetas del directorio `goal-tracker/export_microservice` y su propósito:

- `exporter/`: contiene los ficheros con los endoints.
- `deployment_files/`: contiene los ficheros de despliegue necesarios para Kubernetes y Docker.
- `venv/`: contiene el entorno virtual de python, usado exclusivamente para probar la aplicación de manera local.
- `main.py`: es el punto de entrada principal de la aplicación cuando se ejecuta localmente en modo desarrollo.
- `requirements.txt`: define las dependencias del proyecto, facilitando su instalación en nuevos entornos.

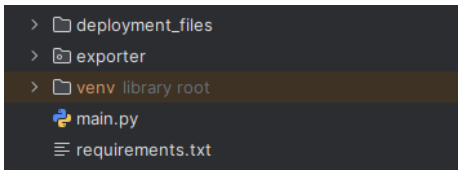


Figura 115: Los ficheros contenidos en `goal-tracker/export_microservice`

4.4.1. exporter/

La carpeta `exporter/` contiene la lógica necesaria para crear los reportes. Esta compuesta de 4 ficheros: `__init__.py`, `backend_client.py`, `export_view.py` y `config.py` Realmente los ficheros `backend_client.py`,`__init__.py` y `config.py` son iguales o muy parecidos a los que hay en el `goal-tracker/frontend` y `goal-tracker/graph_microservice`, por lo tanto su explicación será omitida.

4.4.2. exporter/export_view.py

El fichero `export_view.py` define dos endpoints principales: uno para la generación de reportes en PDF y otro para la exportación en formato CSV. A continuación, se describe el propósito de cada función contenida en el módulo, junto con diagramas que resumen su funcionamiento.

Función `profile` Esta función obtiene la información del perfil del usuario a través del backend. Usa un cliente autenticado y gestiona la autenticación mediante `Token` y `RefreshToken`.

```python
def profile(token: Token, refresh: Token):  1 usage  Cristian
    client = get_client()
    backend_request = RequestBuilder()
    backend_request.auth(token).refresh(refresh).set_method("get").set_endpoint("/user/profile")
    response = client.request_reauth(backend_request)

    if response.status in [401, 403]:
        return jsonify({'error': 'unauthorized'}), 401

    body = json.loads(response.data)
    return body, 200
```

Figura 116: Flujo de autenticación y obtención del perfil del usuario.

Función `get_progress_by_goal_id` Dado un usuario y un ID de objetivo, esta función realiza una solicitud autenticada al backend para obtener todos los progresos registrados para ese objetivo.

```python
def get_progress_by_goal_id(user_id,goal_id, token:Token, refresh:Token):  2 usages  Cristian
    client = get_client()
    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals/{goal_id}/progress")
    )
    response = client.request_reauth(backend_request)
    return json.loads(response.data)["data"]
```

Figura 117: Obtención de progresos asociados a un objetivo.

Endpoint `/user/<user_id>/csv-report` (función `export_csv`) Este endpoint genera un archivo CSV que contiene todos los objetivos de un usuario junto con sus respectivos progresos. La estructura del CSV se construye usando `DictWriter` y se devuelve como una respuesta HTTP con el tipo `text/csv`. Cada fila contiene la meta, seguida del progreso específico.

```python
@export.route('/user/<int:user_id>/csv-report')  Cristian *
def export_csv(user_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh.value:
        return redirect('/')

    client = get_client()
    backend_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals")
    )
    response = client.request_reauth(backend_request)

    if response.status in [401, 403]:
        return redirect('/')

    goals = json.loads(response.data)["data"]

    output = io.StringIO()
    writer = csv.DictWriter(output, fieldnames=[
        'goal_id', 'goal_title', 'goal_description', 'goal_metric', 'goal_totalDesiredAmount', 'goal_creationDate',
        'goal_isCompleted',
        'progress_id', 'progress_note', 'progress_amount', 'progress_date'
    ])
    writer.writeheader()

    for goal in goals:
        progress_list = get_progress_by_goal_id(user_id, goal["id"],token, refresh)
        for progress in progress_list:
            writer.writerow({
                'goal_id': goal['id'],
                'goal_title': goal['title'],
                'goal_description': goal['description'],
                'goal_metric': goal['metric'],
                'goal_totalDesiredAmount': goal['totalDesiredAmount'],
                'goal_creationDate': goal['creationDate'],
                'goal_isCompleted': goal['completed'],
                'progress_id': progress['id'],
                'progress_note': progress['updateNote'],
                'progress_amount': progress['amount'],
                'progress_date': progress['date'],
            })

    response = Response(output.getvalue(), mimetype='text/csv')
    response.headers["Content-Disposition"] = "attachment; filename=data-report.csv"
    response.headers["JWT"] = token.value
    return response
```

Figura 118: Proceso de exportación de reportes en formato CSV.

Endpoint `/user/<user_id>/pdf-report` (función `export_pdf`) Este endpoint genera un reporte PDF que contiene la información del usuario, sus objetivos, y los progresos asociados. Se usa `reportlab` para la generación visual del PDF.

```python
@export.route('/user/<int:user_id>/pdf-report')  Cristian
def export_pdf(user_id):
    token = Token(request.cookies.get('JWT'))
    refresh = Token(request.cookies.get('RefreshToken'))
    if not refresh.value:
        return redirect('/')

    client = get_client()

    goals_request = (
        RequestBuilder()
        .auth(token)
        .refresh(refresh)
        .set_method("get")
        .set_endpoint(f"/user/{user_id}/goals")
    )
    goals_response = client.request_reauth(goals_request)
    if goals_response.status in [401, 403]:
        return redirect('/')
    goals = json.loads(goals_response.data)["data"]

    user_data, status = profile(token, refresh)
    if status in [401, 403]:
        return redirect('/')
    user = user_data["data"][0]

    buffer = io.BytesIO()
    pdf = canvas.Canvas(buffer, pagesize=letter)
    width, height = letter

    draw_pdf_header(pdf, user, width, height)
    y = height - 70
    line_height = 16

    for goal in goals:
        y = draw_goal_section(pdf, user_id, goal, token, refresh, y, line_height, width, height)

    pdf.save()
    buffer.seek(0)

    return send_file(
        buffer,
        as_attachment=True,
        download_name="goal-progress-report.pdf",
        mimetype='application/pdf'
    )

```

Figura 119: Flujo de generación del reporte PDF.

Función `draw_pdf_header` Dibuja el encabezado del PDF, que incluye el nombre del usuario, correo electrónico y su ID. Este encabezado se coloca al principio del documento.

Figura 120: Ejemplo visual del encabezado generado en el PDF.

Función `draw_goal_section` Esta función escribe en el PDF los detalles de cada objetivo, incluyendo título, descripción, métricas. Una vez escrito el objetivo, se escriben todos los progresos a continuación de ese objetivo específico. Si la página no tiene suficiente espacio, crea una nueva página automáticamente.

```python
def draw_goal_section(pdf, user_id, goal, token, refresh, y, line_height, width, height):  1 usage  Cristian
    if y < 70:
        pdf.showPage()
        y = height - 50

    # Goal header
    pdf.setFont("Helvetica-Bold", 12)
    pdf.drawString(30, y, f"Goal: {goal['title']} (ID: {goal['id']})")
    y -= line_height

    # Goal metadata
    pdf.setFont("Helvetica-Oblique", 9)
    pdf.drawString(40, y, f"Description: {goal.get('description', '')}")
    y -= line_height
    pdf.drawString(
        40, y,
        f"Metric: {goal.get('metric', '')} | Total Desired: {goal.get('totalDesiredAmount', '')} | "
        f"Created: {goal.get('creationDate', '')} | Completed: {goal.get('completed', False)}"
    )
    y -= line_height

    # Progress entries
    pdf.setFont("Helvetica", 10)
    progress_list = get_progress_by_goal_id(user_id, goal["id"], token, refresh)
    if not progress_list:
        pdf.drawString(50, y, "No progress recorded.")
        y -= line_height
    else:
        for progress in progress_list:
            if y < 50:
                pdf.showPage()
                y = height - 50
                pdf.setFont("Helvetica", 10)

            pdf.drawString(
                50, y,
                f"- [{progress['date']}] {progress['updateNote']} (Amount: {progress['amount']})"
            )
            y -= line_height

    return y - line_height
```

Figura 121: Dibujo de secciones de objetivos y sus progresos en el PDF.

4.5. Kubernetes y Docker

En esta sección se explican los ficheros relacionados con el despliegue de los distintos servicios. Aunque Kubernetes suele utilizarse sobre un clúster o alguna infraestructura en la nube, para esta demostración se ha empleado Minikube[9], un clúster local que utiliza Docker para desplegar sus contenedores. La ventaja de usar Minikube es que está diseñado para entornos de desarrollo, lo que permite probar diferentes configuraciones y estrategias de despliegue de forma sencilla.

Los ficheros relacionados con el despliegue de cada servicio se encuentran en la carpeta `deployment_files/`. Todas las subcarpetas dentro de `deployment_files/` contienen dos carpetas: `prod` y `test`. En esta sección se explicarán únicamente los ficheros de producción. Los ficheros de despliegue de testing son imágenes estáticas que, al ser reiniciadas, cargan los cambios actualizados del proyecto, útil para agilizar el desarrollo.

También cabe destacar que es necesario seguir un tutorial de implementar self-signed certs, necesario para que los tokens se guarden en el navegador y que se pueda conectar usando HTTPS. Esto nos permite usar TLS, únicamente para poder encriptar HTTP. El navegador mostrará una advertencia indicando que la conexión no es segura, ya que un certificado autofirmado no se reconoce por ninguna autoridad de certificados (CA).

No se ha hecho uso de `configMap` ya que la cantidad de variables de configuración es reducida y estáticas. Sin embargo, en entornos de producción reales, se recomienda externalizar la configuración para facilitar su mantenimiento y reutilización.

4.5.1. Backend

La carpeta `deployment_files/prod/` del servicio Backend contiene dos archivos principales: `Dockerfile` y `deploy.yaml`. A continuación se describe el contenido y función de este último.

4.5.2. Backend/deploy.yaml

El fichero `deploy.yaml` define el despliegue del servicio Backend en el clúster de Kubernetes. Está compuesto por dos recursos: un `Deployment` y un `Service`.

El recurso `Deployment` crea una réplica del contenedor que ejecuta el servicio Backend, utilizando una imagen local llamada `backend.prod.goal.tracker:0.0.1`. Se configuran variables de entorno a partir de secretos definidos previamente en el clúster, como las credenciales de la base de datos y la contraseña de administrador.

El recurso `Service` expone el contenedor a través de un servicio interno de tipo `ClusterIP`, lo que permite la comunicación con otros servicios dentro del mismo clúster.

A continuación se muestra el contenido del fichero `deploy.yaml` como imagen:

4.5.3. Backend/Dockerfile

El archivo `Dockerfile` define la imagen Docker que será utilizada por el servicio Backend. En este caso, se utiliza como base la imagen oficial de `openjdk:21-jdk-slim`, una versión ligera del JDK 21 optimizada para contenedores.

Primero, se actualiza el sistema y se instala `curl`, una herramienta útil para depuración o pruebas HTTP en contenedores. Después, se eliminan los archivos temporales y listas de paquetes para reducir el tamaño final de la imagen.

El directorio de trabajo se establece en `/app`, donde se copiará el fichero `goal-tracker-0.0.1-SNAPSHOT.jar`, que debe haberse generado previamente con `Maven` en el contexto de construcción (es decir, en la carpeta `goal-tracker`).

Finalmente, se especifica el comando por defecto que se ejecutará al iniciar el contenedor, que consiste en lanzar la aplicación Spring Boot usando el JAR copiado.

A continuación se muestra el contenido del fichero `Dockerfile` como imagen:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: prod
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: backend.prod.goal.tracker:0.0.1
          imagePullPolicy: Never
          ports:
            - containerPort: 8000
          env:
            - name: SPRING_DATASOURCE_USERNAME
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_USER
            - name: SPRING_DATASOURCE_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_PASSWORD
            - name: SPRING_DATASOURCE_URL
              valueFrom:
                secretKeyRef:
                  name: backend-secret
                  key: DB_URL
            - name: ADMIN_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: backend-secret
                  key: ADMIN_PASSWORD
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: backend
  ports:
    - port: 8000
      targetPort: 8000
```

Figura 122: Fichero `deploy.yaml` para el servicio Backend

```
1  FROM openjdk:21-jdk-slim
2
3  RUN apt-get update && \
4      apt-get install -y --no-install-recommends curl && \
5      apt-get clean && \
6      rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
7
8  #Build context should be the the goal tracker folder
9
10 WORKDIR /app
11
12 COPY backend/target/goal-tracker-0.0.1-SNAPSHOT.jar goal-tracker.jar
13
14 CMD ["java", "-jar", "goal-tracker.jar"]
```

Figura 123: Fichero `Dockerfile` para el servicio Backend

4.5.4. Frontend

La carpeta `deployment_files/prod/` del servicio Frontend contiene dos archivos principales: `Dockerfile` y `deploy.yaml`. A continuación se describe el contenido y función de cada uno de ellos.

4.5.5. Frontend/deploy.yaml

El fichero `deploy.yaml` define el despliegue del servicio Frontend, implementado con Flask, en el clúster de Kubernetes. Al igual que en el backend, este archivo incluye dos recursos: un `Deployment` y un `Service`.

El recurso `Deployment` lanza una réplica de un contenedor construido a partir de la imagen local `flask-frontend.prod.goal.tracker:0.0.1`. Se configuran variables de entorno necesarias para el correcto funcionamiento de Flask y para establecer la URL del backend al que debe conectarse.

El recurso `Service` expone el contenedor mediante un servicio de tipo `ClusterIP`, que lo hace accesible internamente dentro del clúster de Kubernetes. Este servicio escucha en el puerto 80 y redirige el tráfico al puerto 5000 del contenedor.

A continuación se muestra el contenido del fichero `deploy.yaml` como imagen:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-frontend
  namespace: prod
  labels:
    app: flask-frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flask-frontend
  template:
    metadata:
      labels:
        app: flask-frontend
    spec:
      containers:
        - name: flask-frontend
          image: flask-frontend.prod.goal.tracker:0.0.1
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
          env:
            - name: FLASK_RUN_HOST
              value: 0.0.0.0
            - name: BACKEND_URL
              value: http://backend-service.prod.svc.cluster.local:8000/
---
apiVersion: v1
kind: Service
metadata:
  name: flask-frontend
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: flask-frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

Figura 124: Fichero `deploy.yaml` para el servicio Frontend

4.5.6. Frontend/Dockerfile

El archivo `Dockerfile` define cómo se construye la imagen Docker del servicio Frontend, que ejecuta una aplicación Flask servida mediante Gunicorn.

Se utiliza como base la imagen `python:3.13.0a4-slim`, una versión ligera y actual del intérprete de Python. Se actualiza el sistema, se instala `curl` como herramienta auxiliar y se limpian los archivos temporales para reducir el tamaño de la imagen.

Luego se copia el fichero `requirements.txt` en el directorio de trabajo y se instalan las dependencias necesarias mediante `pip`. A continuación, se copia el resto del código fuente ubicado en la carpeta `frontend`.

Finalmente, se define la variable de entorno `BACKEND_URL` con la dirección del servicio backend en el clúster, y se indica como comando de arranque el uso de Gunicorn para servir la aplicación Flask, invocando la función `create_app()` desde el paquete `app`.

A continuación se muestra el contenido del fichero `Dockerfile` como imagen:

```
1  FROM python:3.13.0a4-slim
2
3  RUN apt-get update && \
4      apt-get install -y --no-install-recommends curl && \
5      apt-get clean && \
6      rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
7
8  WORKDIR /app
9
10 COPY frontend/requirements.txt .
11
12 RUN pip install --no-cache-dir -r requirements.txt
13
14 COPY frontend .
15
16 ENV BACKEND_URL=http://backend-service.prod.svc.cluster.local:8000/
17
18 CMD ["gunicorn", "-w", "2", "-b", "0.0.0.0:5000", "app.__init__:create_app()"]
19
```

Figura 125: Fichero `Dockerfile` para el servicio Frontend

4.5.7. Graph Service

La carpeta `deployment_files/prod/` del servicio Graph Service contiene dos archivos principales: `Dockerfile` y `deploy.yaml`. A continuación se describe el contenido y función de cada uno de ellos.

4.5.8. Graph Service/deploy.yaml

El fichero `deploy.yaml` define el despliegue del servicio Graph Service, un microservicio Flask que genera gráficos, dentro del clúster de Kubernetes. Este archivo contiene dos recursos esenciales: un `Deployment` y un `Service`.

El recurso `Deployment` lanza una réplica de un contenedor generado a partir de la imagen local `graph-microservice.prod.goal.tracker:0.0.1`. Se definen variables de entorno necesarias para el correcto funcionamiento del microservicio, como la dirección del backend y una ruta para los recursos estáticos.

El recurso `Service` expone el microservicio mediante un servicio de tipo `ClusterIP`, accesible solo desde dentro del clúster. Escucha en el puerto 80 y redirige las solicitudes al puerto 5000 del contenedor.

A continuación se muestra el contenido del fichero `deploy.yaml` como imagen:

4.5.9. Graph Service/Dockerfile

El archivo `Dockerfile` define cómo se construye la imagen Docker del microservicio Graph Service. Esta imagen se basa en `python:3.13.1-slim`, una versión ligera y moderna del intérprete de Python.

Durante la construcción de la imagen, se instala `curl` como utilidad auxiliar, y se eliminan archivos temporales para mantener el contenedor lo más liviano posible.

Posteriormente, se copia el fichero `requirements.txt` desde la carpeta `graph_microservice` y se instalan las dependencias necesarias. Luego se copia todo el código fuente del microservicio en el directorio de trabajo del contenedor.

Se define la variable de entorno `BACKEND_URL` apuntando al servicio backend, y finalmente, el contenedor se configura para ejecutar el servidor Gunicorn, que lanza la aplicación Flask a través de la función `create_app()` definida en el módulo `graph`.

A continuación se muestra el contenido del fichero `Dockerfile` como imagen:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: graph-microservice
  namespace: prod
  labels:
    app: graph-microservice
spec:
  replicas: 1
  selector:
    matchLabels:
      app: graph-microservice
  template:
    metadata:
      labels:
        app: graph-microservice
    spec:
      containers:
        - name: graph-microservice
          image: graph-microservice.prod.goal.tracker:0.0.1
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
          env:
            - name: FLASK_RUN_HOST
              value: 0.0.0.0
            - name: BACKEND_URL
              value: http://backend-service.prod.svc.cluster.local:8000/
            - name: STATIC_URL
              value: /graph-static
---
apiVersion: v1
kind: Service
metadata:
  name: graph-service
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: graph-microservice
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000

```

Figura 126: Fichero `deploy.yaml` para el servicio Graph Service

```
FROM python:3.13.1-slim

RUN apt-get update && \
    apt-get install -y --no-install-recommends curl && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

WORKDIR /app

COPY graph_microservice/requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY graph_microservice .

ENV BACKEND_URL=http://backend-service.prod.svc.cluster.local:8000/

CMD ["gunicorn", "-w", "2", "-b", "0.0.0.0:5000", "graph.__init__:create_app()"]
```

Figura 127: Fichero `Dockerfile` para el servicio Graph Service

4.5.10. Export Service

La carpeta `deployment_files/prod/` del servicio Export Service contiene dos archivos principales: `Dockerfile` y `deploy.yaml`. A continuación se describe el contenido y la función de cada uno de ellos.

4.5.11. Export Service/deploy.yaml

El fichero `deploy.yaml` define el despliegue del microservicio Export Service, encargado de exportar datos, en el clúster de Kubernetes. Este archivo contiene dos recursos esenciales: un `Deployment` y un `Service`.

El recurso `Deployment` lanza una réplica de un contenedor construido a partir de la imagen local `export-microservice.prod.goal.tracker:0.0.1`. Se especifican variables de entorno necesarias para configurar la ejecución del servidor Flask, como el valor `FLASK_RUN_HOST`, que permite que la aplicación escuche en todas las interfaces del contenedor.

El recurso `Service` expone el microservicio dentro del clúster mediante un servicio de tipo `ClusterIP`, lo cual permite la comunicación interna entre servicios. El puerto 80 del servicio se mapea al puerto 5000 del contenedor.

A continuación se muestra el contenido del fichero `deploy.yaml` como imagen:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: export-microservice
  namespace: prod
  labels:
    app: export-microservice
spec:
  replicas: 1
  selector:
    matchLabels:
      app: export-microservice
  template:
    metadata:
      labels:
        app: export-microservice
    spec:
      containers:
        - name: export-microservice
          image: export-microservice.prod.goal.tracker:0.0.1
          imagePullPolicy: Never
          ports:
            - containerPort: 5000
          env:
            - name: FLASK_RUN_HOST
              value: 0.0.0.0
---
apiVersion: v1
kind: Service
metadata:
  name: export-service
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: export-microservice
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

Figura 128: Fichero `deploy.yaml` para el servicio Export Service

4.5.12. Export Service/Dockerfile

El archivo `Dockerfile` especifica cómo se construye la imagen Docker del microservicio Export Service. Se utiliza como base la imagen `python:3.13.1-slim`, una distribución ligera del intérprete de Python.

Durante el proceso de construcción de la imagen, se actualiza el sistema y se instala `curl` como utilidad auxiliar. A continuación, se eliminan archivos temporales para mantener el tamaño de la imagen al mínimo.

Posteriormente, se copia el fichero `requirements.txt` desde la carpeta `export_microservice` y se instalan las dependencias necesarias. Luego se copia el resto del código fuente del microservicio dentro del contenedor.

Finalmente, se define la variable de entorno `BACKEND_URL`, y se especifica como comando de inicio el uso del servidor `Gunicorn`, el cual ejecuta la aplicación Flask a través de la función `create_app()` del paquete `exporter`.

A continuación se muestra el contenido del fichero `Dockerfile` como imagen:

```
1  FROM python:3.13.1-slim
2
3  RUN apt-get update && \
4      apt-get install -y --no-install-recommends curl && \
5      apt-get clean && \
6      rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
7
8  WORKDIR /app
9
10 COPY export_microservice/requirements.txt .
11
12 RUN pip install --no-cache-dir -r requirements.txt
13
14 COPY export_microservice .
15
16 ENV BACKEND_URL=http://backend-service.prod.svc.cluster.local:8000/
17
18 CMD ["gunicorn", "-w", "2", "-b", "0.0.0.0:5000", "exporter.__init__:create_app()"]
```

Figura 129: Fichero `Dockerfile` para el servicio Export Service

4.5.13. Database

El servicio de base de datos utiliza una instancia de MySQL 8.0 desplegada dentro del clúster de Kubernetes. A diferencia de otros servicios, esta configuración incluye tres recursos principales definidos en un único fichero `deploy.yaml`: un `PersistentVolumeClaim`, un `Deployment` y un `Service`.

En primer lugar, el recurso `PersistentVolumeClaim` (PVC) denominado `mysql-pvc` solicita 1 GiB de almacenamiento persistente para guardar los datos de la base de datos. Este almacenamiento se monta posteriormente en el contenedor de MySQL, garantizando que los datos se mantengan incluso si el contenedor se reinicia.

El recurso `Deployment` crea una réplica del contenedor que ejecuta la base de datos MySQL. La imagen utilizada es la oficial de Docker Hub: `mysql:8.0`. La configuración del contenedor incluye las variables de entorno necesarias para establecer las credenciales y el esquema de la base de datos, todas ellas definidas mediante referencias a un secreto denominado `mysql-secret`. Además, se monta el volumen persistente en la ruta `/var/lib/mysql`, que es el directorio predeterminado donde MySQL almacena sus datos.

Por último, el recurso `Service` expone el contenedor a través de un servicio interno sin dirección IP asignada (`clusterIP: None`), lo cual permite acceder directamente al pod mediante DNS. Este patrón es típico en servicios de base de datos donde se busca una comunicación directa entre servicios dentro del mismo clúster.

A continuación se muestra el contenido completo del fichero `deploy.yaml` como imagen:

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
  namespace: prod
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: prod
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:8.0
          imagePullPolicy: IfNotPresent
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_ROOT_PASSWORD
            - name: MYSQL_DATABASE
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_DATABASE
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_USER
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_PASSWORD
          ports:
            - containerPort: 3306
          volumeMounts:
            - name: mysql-storage
              mountPath: /var/lib/mysql
      volumes:
        - name: mysql-storage
          persistentVolumeClaim:
            claimName: mysql-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: mysql
  namespace: prod
spec:
  selector:
    app: mysql
  ports:
    - port: 3306
      targetPort: 3306
  clusterIP: None
```

Figura 130: Fichero `deploy.yaml` para el servicio de base de datos

4.5.14. Ingress de NGINX

Este servicio permite el enrutamiento de las solicitudes HTTP y HTTPS a los distintos microservicios, tal como se muestra en el gráfico de la arquitectura en la figura 2. Los manifiestos se encuentran en la carpeta `goal-tracker/kubernetes`, y el fichero que se explica en esta sección es `ingress_prod.yaml`, el cual contiene tres recursos `Ingress` que redirigen peticiones a distintos servicios internos del clúster. Se han divido los ingress para aumentar la claridad.

El primer recurso `Ingress`, llamado `api-ingress`, se encarga de redirigir las solicitudes dirigidas a rutas que comienzan por `/api` hacia el servicio interno `backend-service`. Para lograrlo, utiliza expresiones regulares y la anotación `rewrite-target` para reescribir las rutas correctamente.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: api-ingress
  namespace: prod
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  tls:
    - hosts:
        - goal-tracker
      secretName: goal-tracker-tls-secret
  rules:
    - host: goal-tracker
      http:
        paths:
          - path: /api(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: backend-service
                port:
                  number: 8000

```

Figura 131: Recurso `Ingress` para redirigir solicitudes API al backend

El segundo recurso `Ingress`, `frontend-ingress`, es responsable de enrutar las solicitudes del usuario hacia los distintos servicios frontend y de exportación. Utiliza rutas con expresiones regulares para distinguir entre peticiones que deben ser gestionadas por el microservicio de gráficos, el frontend en Flask o el servicio de exportación. Además, configura un certificado TLS utilizando el secreto `goal-tracker-tls-secret`, lo que permite el uso de HTTPS.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend-ingress
  namespace: prod
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/use-regex: "true"
spec:
  tls:
    - hosts:
      - goal-tracker
      secretName: goal-tracker-tls-secret
  rules:
    - host: goal-tracker
      http:
        paths:
          - path: /user/.*/goals/.*/graph
            pathType: ImplementationSpecific
            backend:
              service:
                name: graph-service
                port:
                  number: 80
          - path: /user/.*/goal/.*
            pathType: ImplementationSpecific
            backend:
              service:
                name: flask-frontend
                port:
                  number: 80
          - path: /user/.*/csv-report
            pathType: ImplementationSpecific
            backend:
              service:
                name: export-service
                port:
                  number: 80
          - path: /user/.*/pdf-report
            pathType: ImplementationSpecific
            backend:
              service:
                name: export-service
                port:
                  number: 80
          - path: /
            pathType: Prefix
            backend:
              service:
                name: flask-frontend
                port:
                  number: 80
```

Figura 132: Recurso `Ingress` para el enrutamiento de rutas del usuario con soporte TLS

El tercer recurso, `graph-static-ingress`, está dedicado a servir contenido estático generado por el servicio de gráficos, expuesto bajo la ruta `/graph-static`. Se configura también mediante expresiones regulares para redirigir correctamente las rutas al backend del servicio gráfico, realizando una reescritura del path hacia la carpeta `/static`.

```
78  apiVersion: networking.k8s.io/v1
79  kind: Ingress
80  metadata:
81    name: graph-static-ingress
82    namespace: prod
83    annotations:
84      kubernetes.io/ingress.class: "nginx"
85      nginx.ingress.kubernetes.io/use-regex: "true"
86      nginx.ingress.kubernetes.io/rewrite-target: /static/$2
87  spec:
88    rules:
89      - host: goal-tracker
90        http:
91          paths:
92            - path: /graph-static(/|$)(.*)
93              pathType: ImplementationSpecific
94              backend:
95                service:
96                  name: graph-service
97                  port:
98                    number: 80
99
```

Figura 133: Recurso `Ingress` para el contenido estático del servicio de gráficos

4.5.15. Secrets

Los secretos (`Secrets`) en Kubernetes se utilizan para almacenar información sensible, como credenciales de acceso o cadenas de conexión, de forma segura. En este proyecto se definen dos recursos de tipo `Secret`.

El primer secreto, `mysql-secret`, contiene la configuración necesaria para el despliegue del servicio MySQL. Define las variables `MYSQL_ROOT_PASSWORD`, `MYSQL_DATABASE`, `MYSQL_USER` y `MYSQL_PASSWORD`, las cuales son consumidas por el contenedor de la base de datos y también por el backend.

El segundo secreto, `backend-secret`, incluye la URL de conexión a la base de datos (`DB_URL`) y la contraseña del usuario administrador (`ADMIN_PASSWORD`), las cuales son utilizadas únicamente por el servicio Backend.

Estos secretos son de tipo `Opaque` y se definen utilizando el campo `stringData`, lo que permite almacenar los valores como texto plano en el manifiesto, siendo luego convertidos automáticamente a base64 por Kubernetes al ser aplicados al clúster.

A continuación, se muestra el contenido del fichero de definición de los secretos como imagen:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
  namespace: prod
type: Opaque
stringData:
  MYSQL_ROOT_PASSWORD:
  MYSQL_DATABASE:
  MYSQL_USER:
  MYSQL_PASSWORD:
---
apiVersion: v1
kind: Secret
metadata:
  name: backend-secret
  namespace: prod
type: Opaque
stringData:
  DB_URL: jdbc:mysql://mysql.prod.svc.cluster.local/springboot_prod?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true
  ADMIN_PASSWORD:
```

Figura 134: Definición de secretos `mysql-secret` y `backend-secret`, que estan vacios de valores.

4.5.16. Despliegue

Para poder desplegar el proyecto es necesario usar `docker build` para cada uno de los servicios. Pero previo a ese paso es necesario ejecutar el comando:

```
eval $(minikube docker-env)
```

Este comando permite ver y usar el docker que usa minikube, esto se hace para evitar subir al registro las imagenes, ya que no es proyecto que vaya a ser despleguedo en un entorno real. Por esta razón, en todos los `deploy.yaml`, la `imagePullPolicy` esta establecida a `Never`, para que no realice docker pull para traerse la imagen y que únicamente use la imagen local guardada.

A continuación se ejecutan estos comandos en la carpeta `goal-tracker/`:

```
docker build -f backend/deployment_files/prod/Dockerfile -t
backend.prod.goal.tracker:0.0.1 .
```

```
docker build -f graph_microservice/deployment_files/prod/Dockerfile
-t graph-microservice.prod.goal.tracker:0.0.1 .
```

```
docker build -f frontend/deployment_files/prod/Dockerfile -t
flask-frontend.prod.goal.tracker:0.0.1 .
```

```
docker build -f export_microservice/deployment_files/prod/Dockerfile
-t export-microservice.prod.goal.tracker:0.0.1 .
```

Finalmente, se hace un port-forward para que se pueda acceder a la web desde windows:

```
kubectl port-forward -n ingress-nginx svc/ingress-nginx-controller 8080:443
```

Pero es necesario darle nombre al host, por lo tanto es necesario añadir en la carpeta de windows `/WINDOWS/system32/drivers/etc/hosts`:

```
127.0.0.1 goal-tracker
```

Esto permite conectarnos a la página web usando cualquier navegador, a través de la url https://goal-tracker:8080/.

5. Resultados

A continuación se van a mostrar pruebas y ejemplos de uso de la aplicación.

5.1. Registro

Primero registramos un usuario con datos válidos:

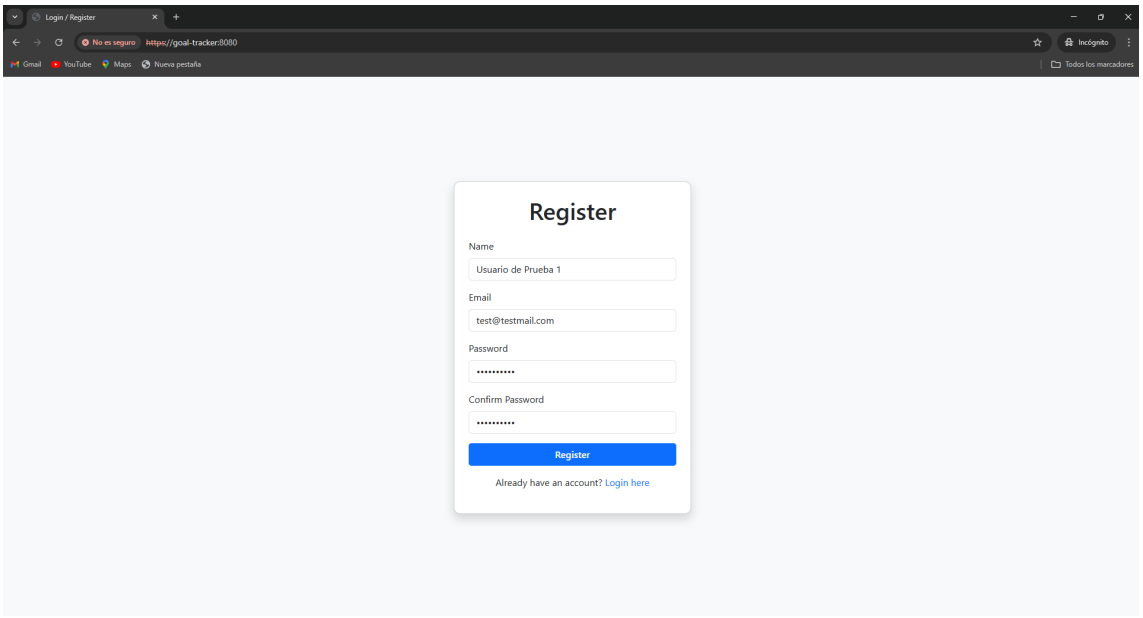


Figura 135: Registro de usuario válido

Al ser válido la página redirige automáticamente a la vista principal donde estan las metas asociadas al usuario, que está vacío debido a que es la primera vez que el usuario entra a la aplicación:

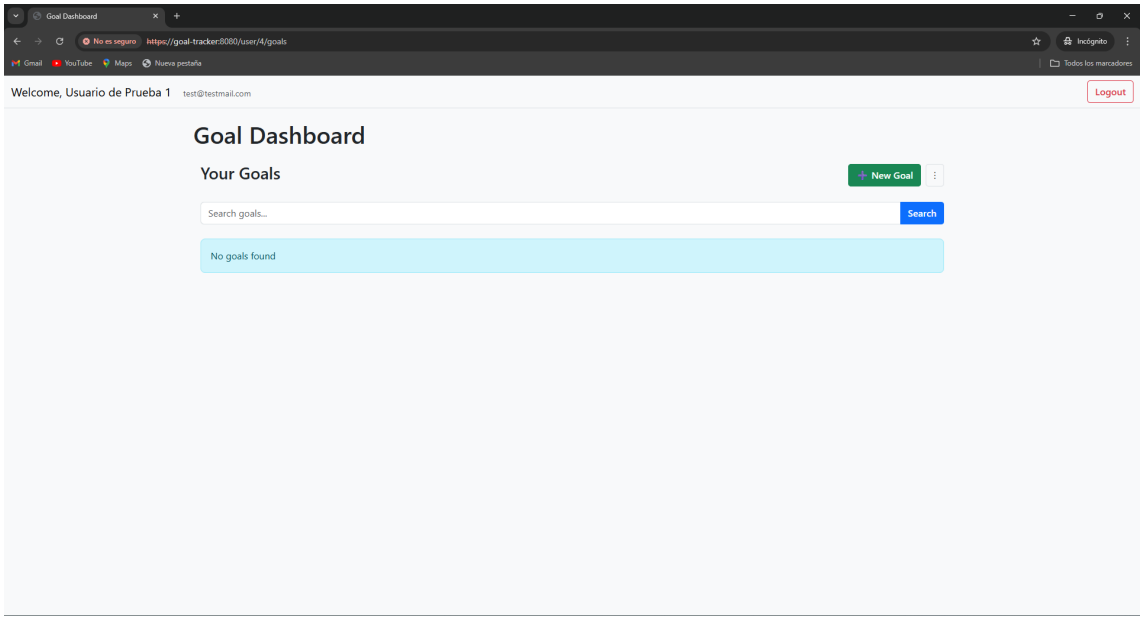


Figura 136: Redirección después de registrarse

También se prueba el registro con datos inválidos para mostrar los errores:

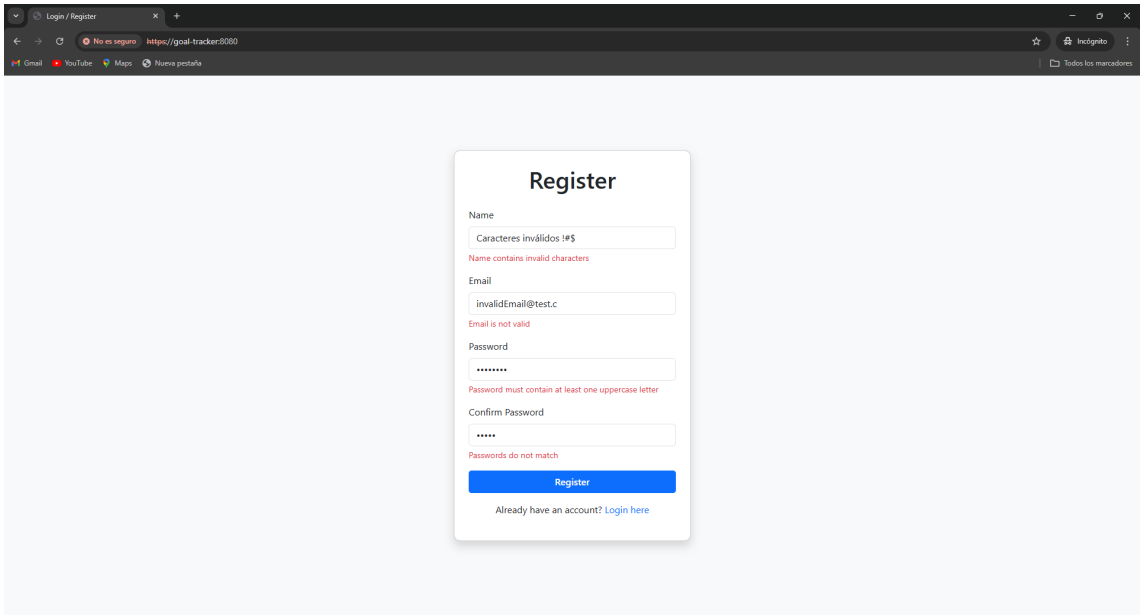


Figura 137: Registro de usuario inválido

5.2. Login

Usando la cuenta anterior,se realiza el login:

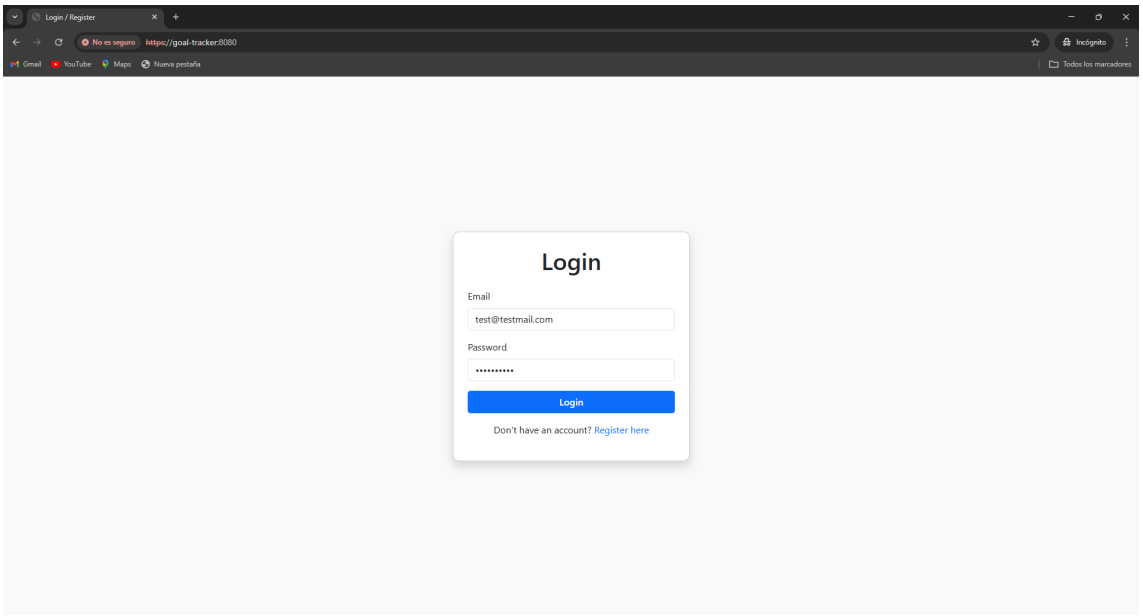


Figura 138: Inicio de sesión válido

Que redirige a la vista principal:

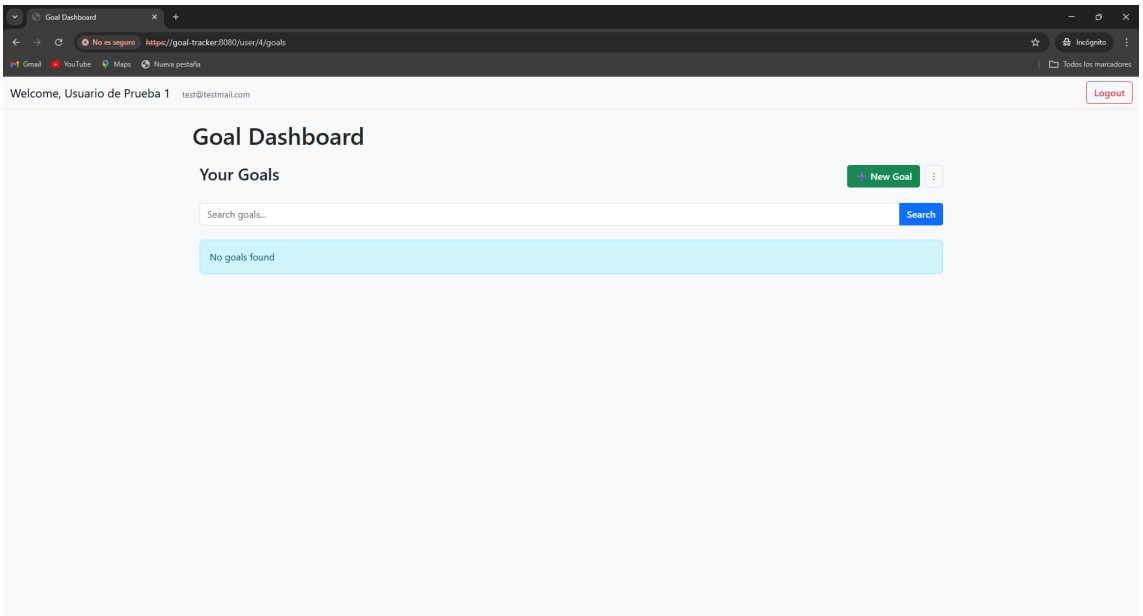


Figura 139: Redirección después de iniciar sesión

Y por último se prueba con credenciales inválidas:

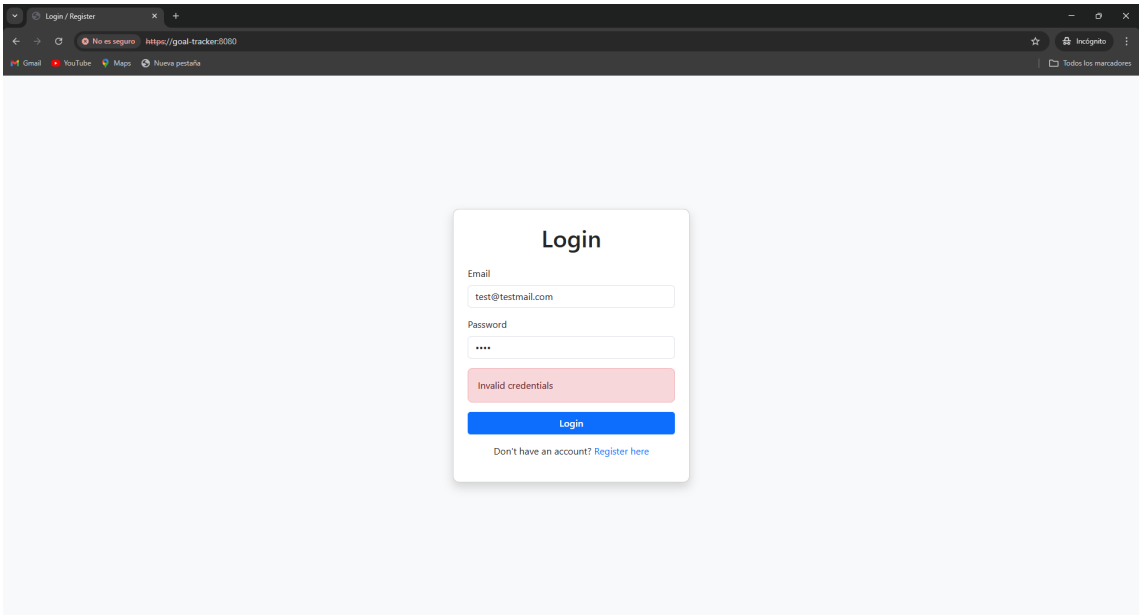


Figura 140: Inicio de sesión inválido

5.3. Gestionar Metas

A continuación se van a probar las operaciones básicas CRUD con las metas:

5.3.1. Añadir meta

Podemos probar la funcionalidad de añadir una meta:

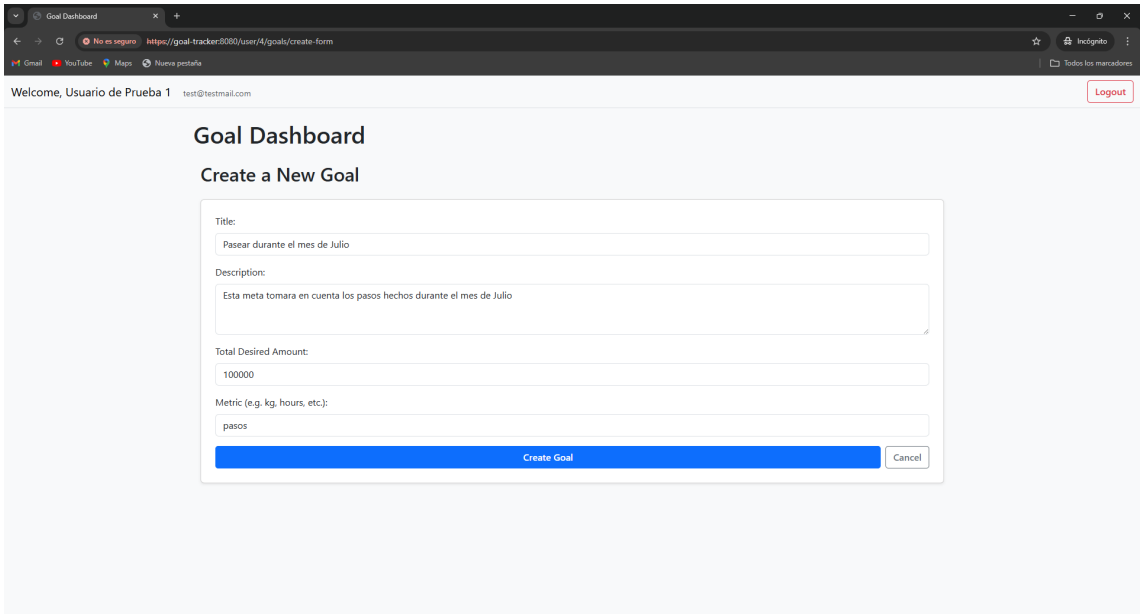


Figura 141: Crear una meta nueva

Y una vez creado redirige a la vista anterior, donde se puede apreciar la meta creada:

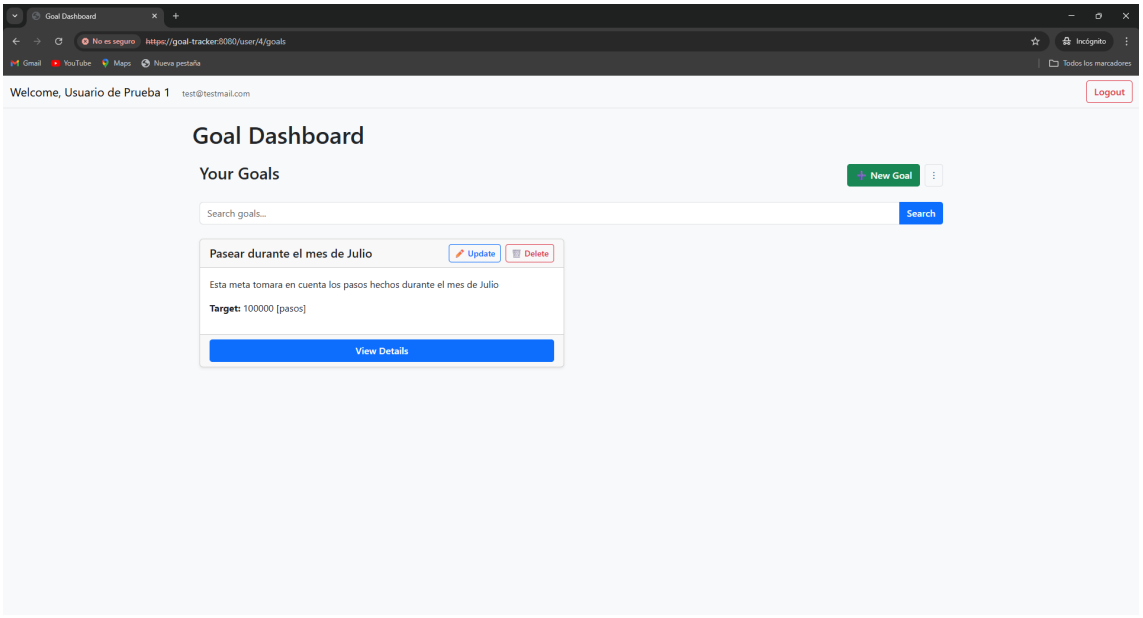


Figura 142: Creación de una meta exitosa

5.3.2. Actualizar meta

Al pulsar el botón de `update` de una meta, se carga un formulario con los datos originales que podemos cambiar. En este caso se va a cambiar el mes de Julio por Agosto:

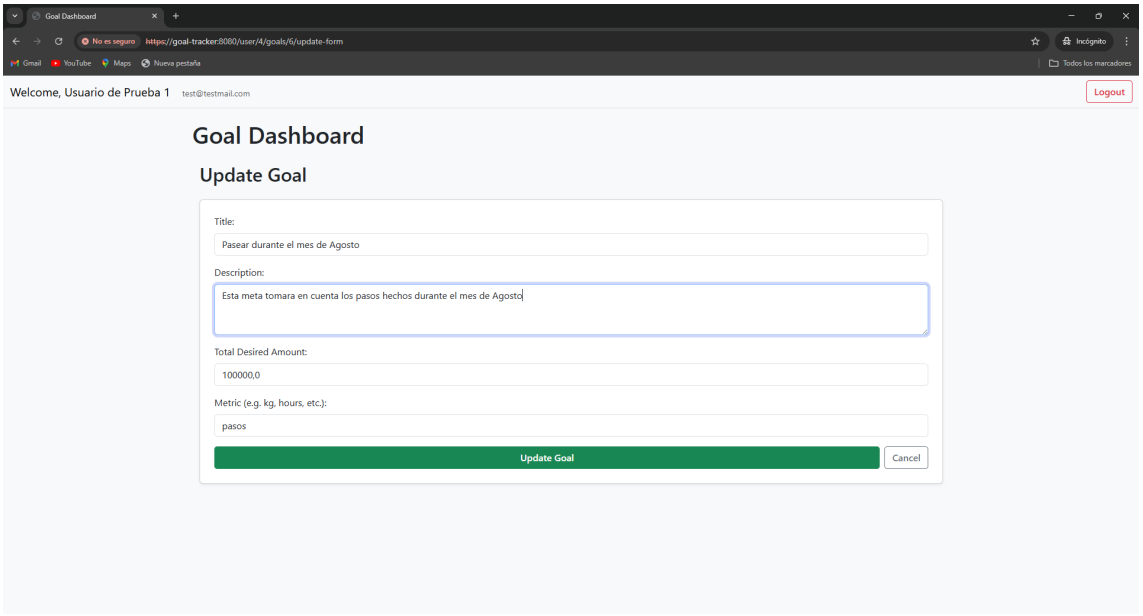


Figura 143: Actualizar una meta

Y redirige a la vista individual de la propia meta, donde se visualizan los nuevos cambios. Si se pulsa sobre el boton `go back`, se vuelve a la vista donde se ven todas las metas. Esta vista en específico se ve también al darle click a `View Details`

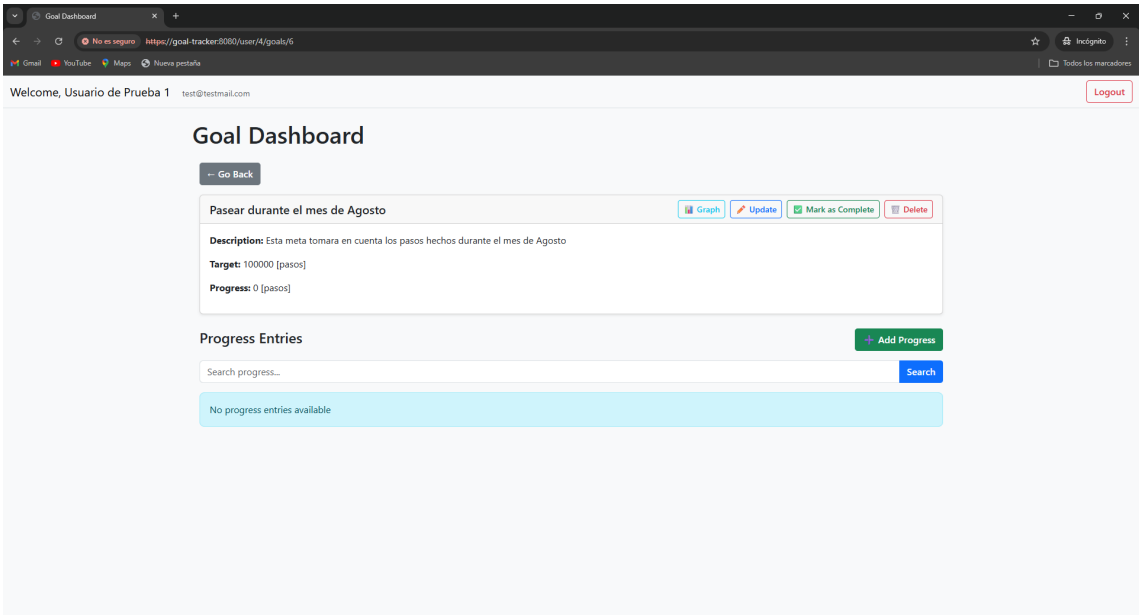


Figura 144: Actualización de una meta exitosa, redirige a la vista individual de la meta

5.3.3. Borrar meta

También se pueden borrar las metas:

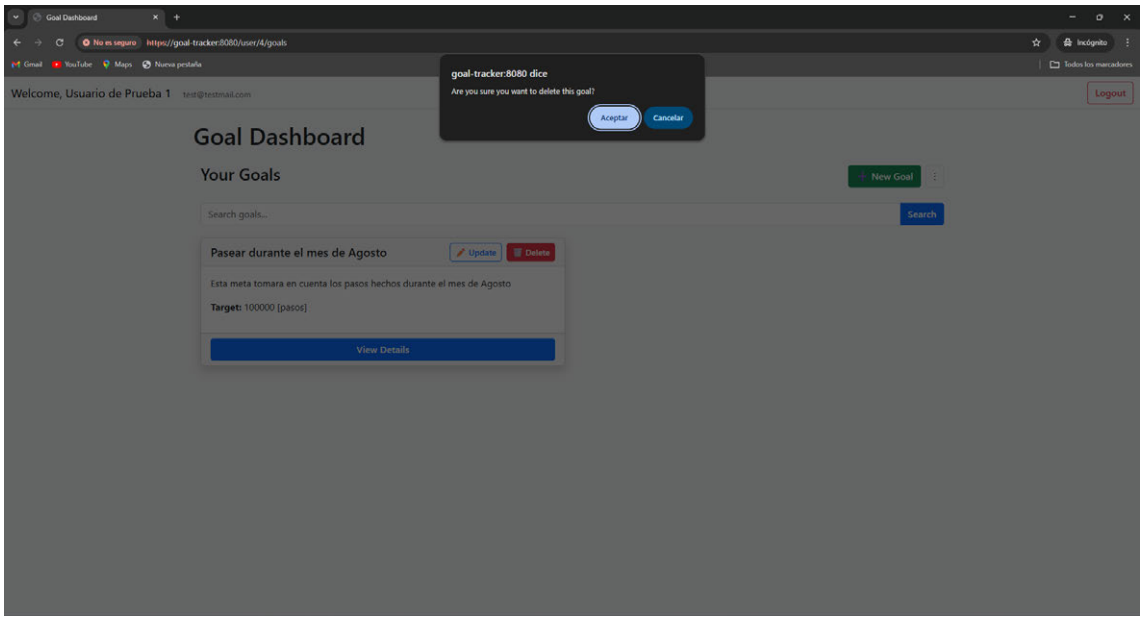


Figura 145: Borrar una meta

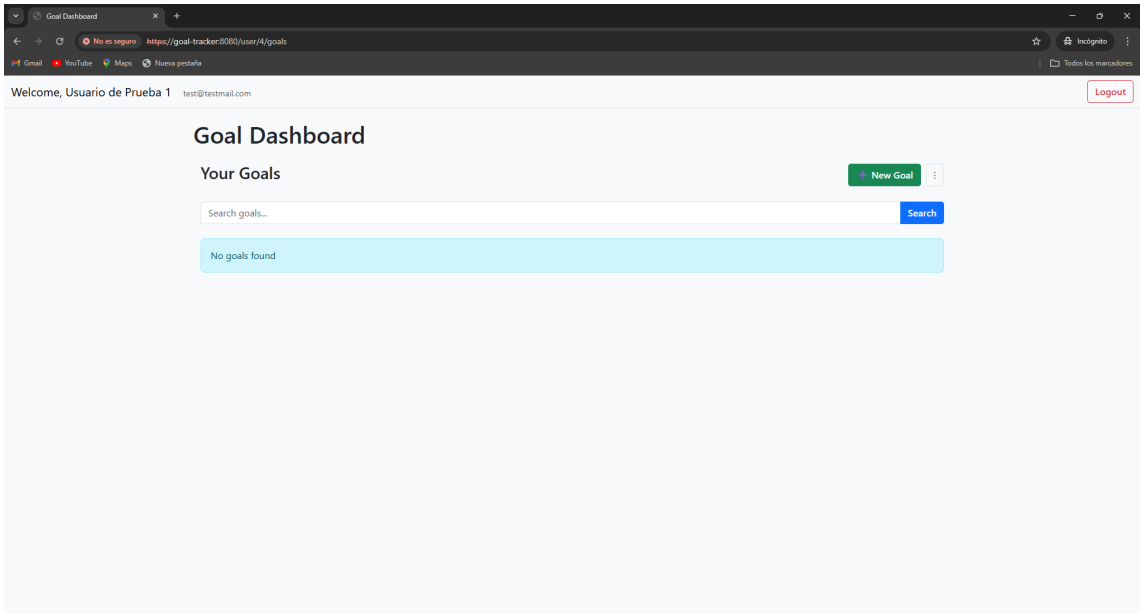


Figura 146: Meta borrada

5.4. Gestionar Progreso/Avances

A continuación se van a probar las operaciones básicas CRUD con el progreso:

5.4.1. Añadir progreso

Se prueba la funcionalidad de añadir un registro de progreso:

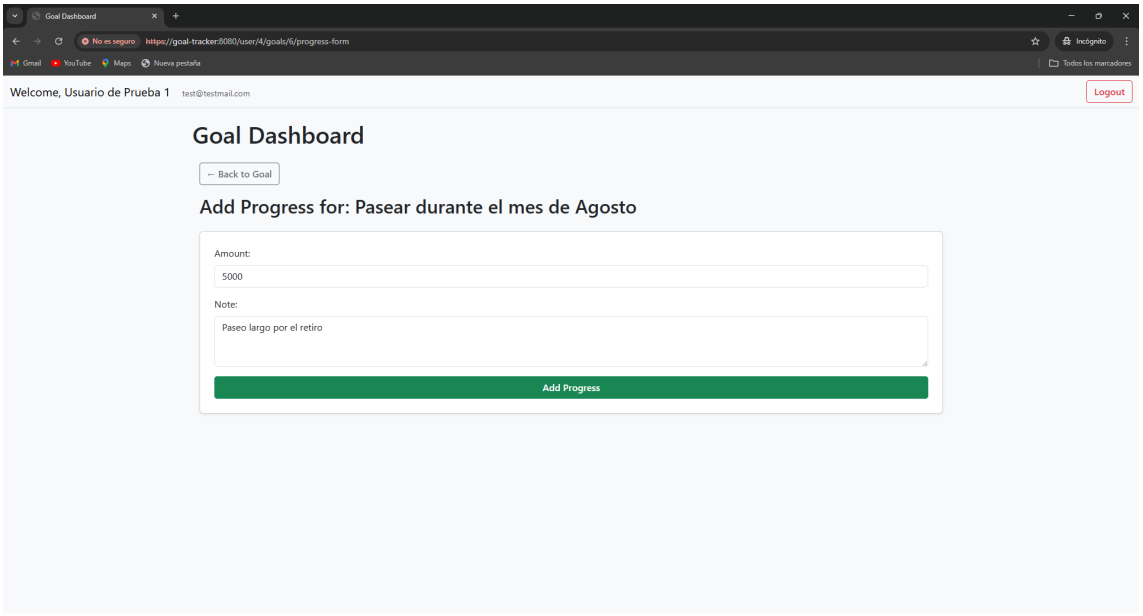


Figura 147: Crear un avance nuevo

Y una vez creado se redirige a la vista de la meta individual, donde también se aprecian los avances, que además muestran su fecha de creación:

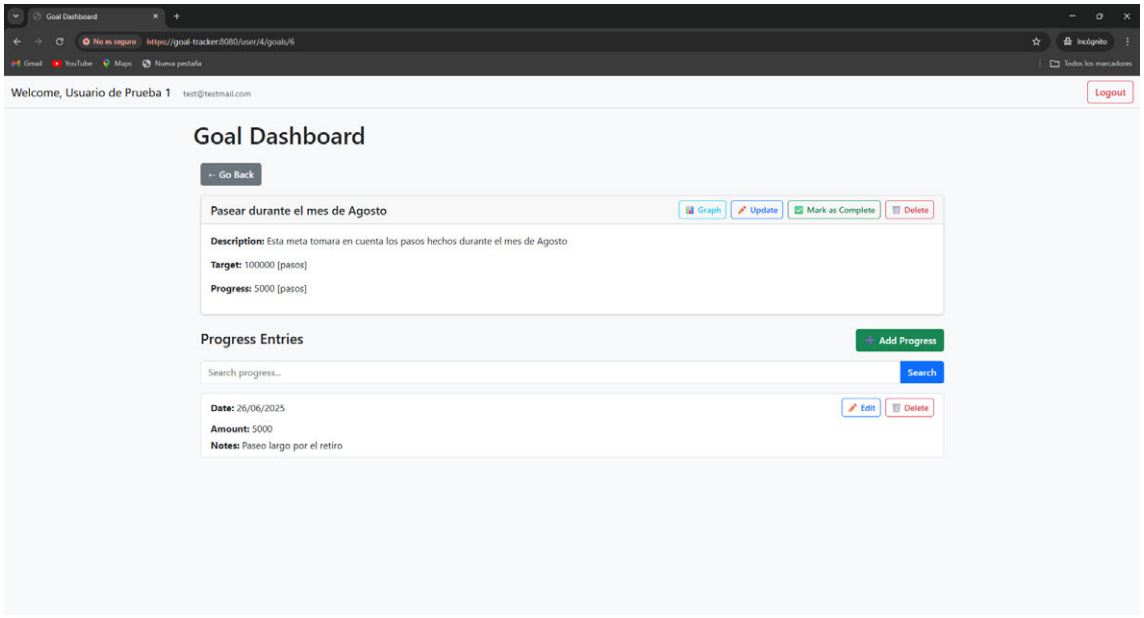


Figura 148: Creación de un avance exitoso

5.4.2. Actualizar progreso

Al pulsar el botón de `edit` de un avance, se carga un formulario con los datos originales que pueden ser cambiados. En este caso se va a cambiar los pasos hechos de 5000 a 6000:

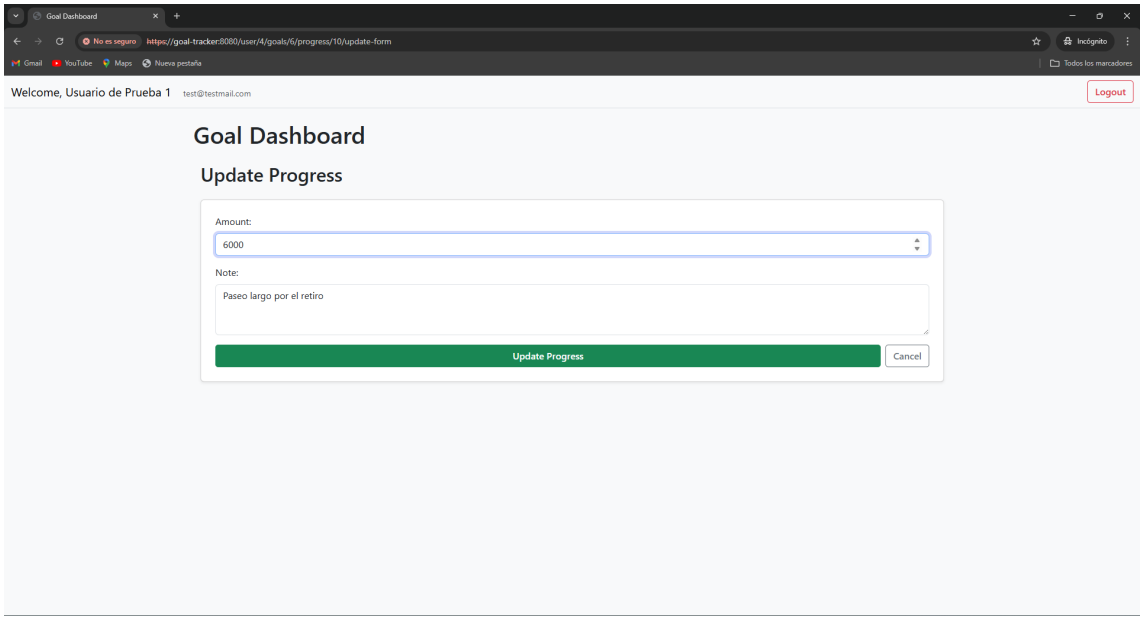


Figura 149: Actualizar un avance

Y redirige a la vista individual de la propia meta, donde se visualizan los nuevos cambios:

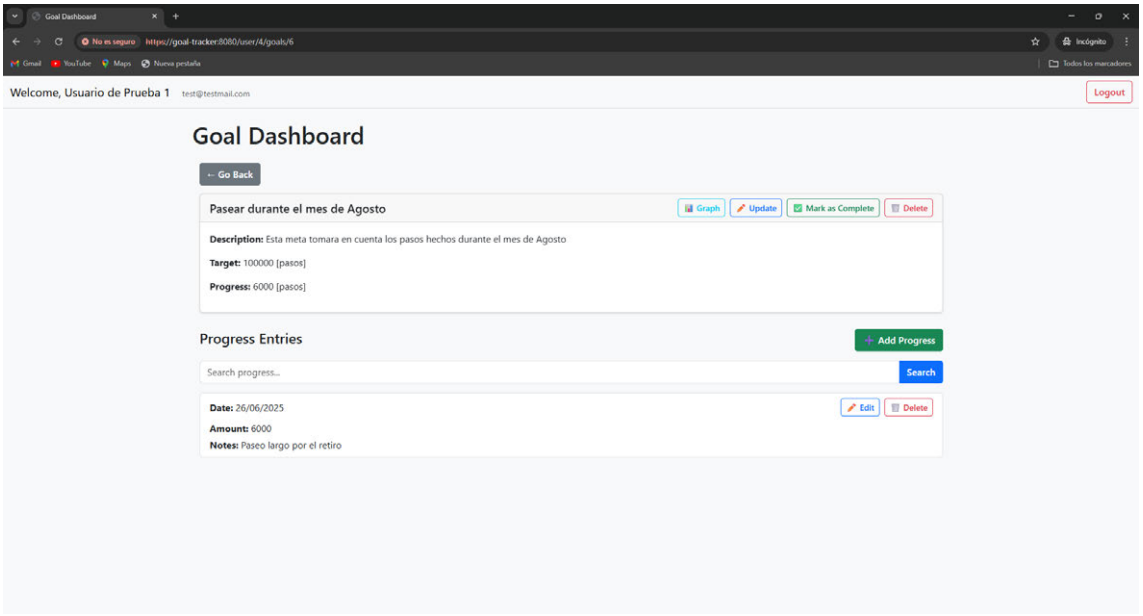


Figura 150: Actualización de un avance exitoso, redirige a la vista individual de la meta

5.4.3. Borrar Progreso

También se pueden borrar los avances:

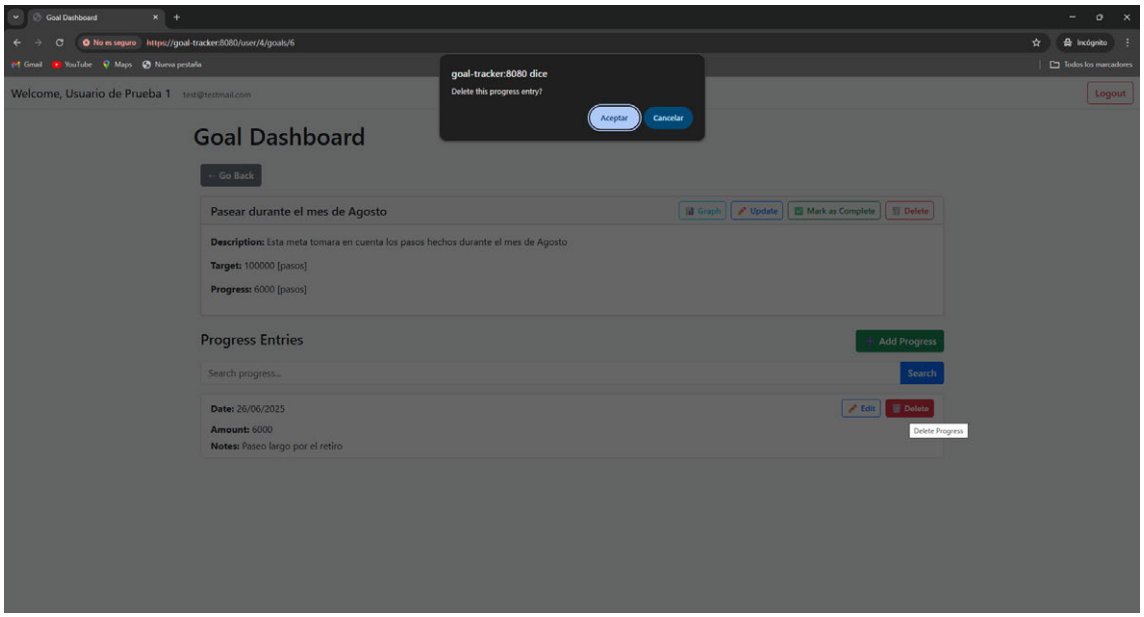


Figura 151: Borrar un avance

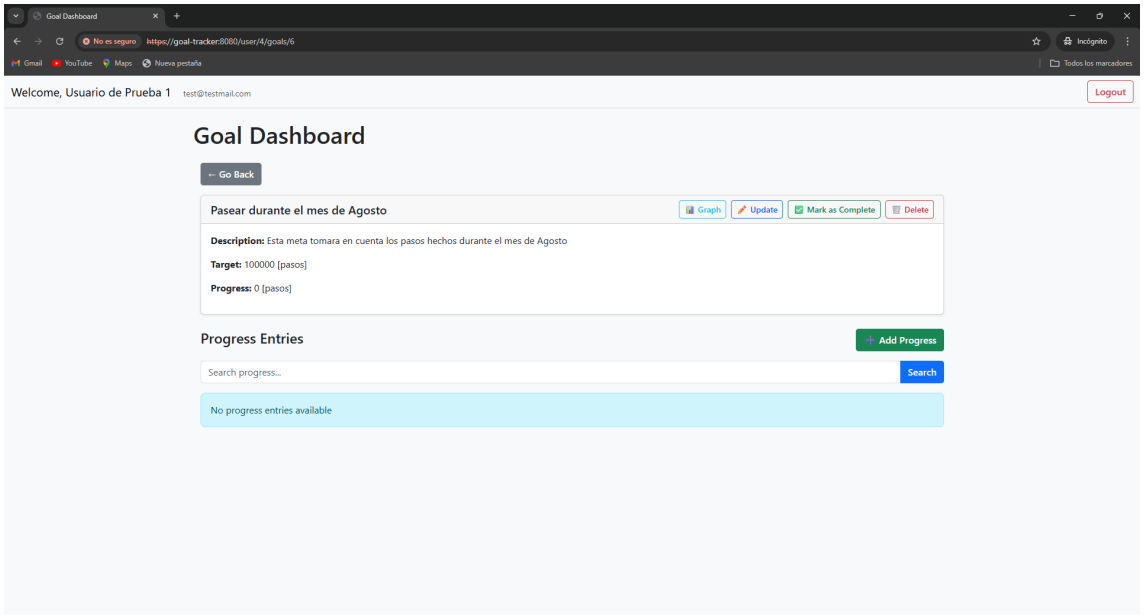


Figura 152: Avance borrado

5.5. Visualizar el gráfico

Usaremos de ejemplo otra meta que tiene varios avances registrados. Para poder acceder a esta vista, hay que pulsar el botón `graph` dentro de los detalles de una meta:

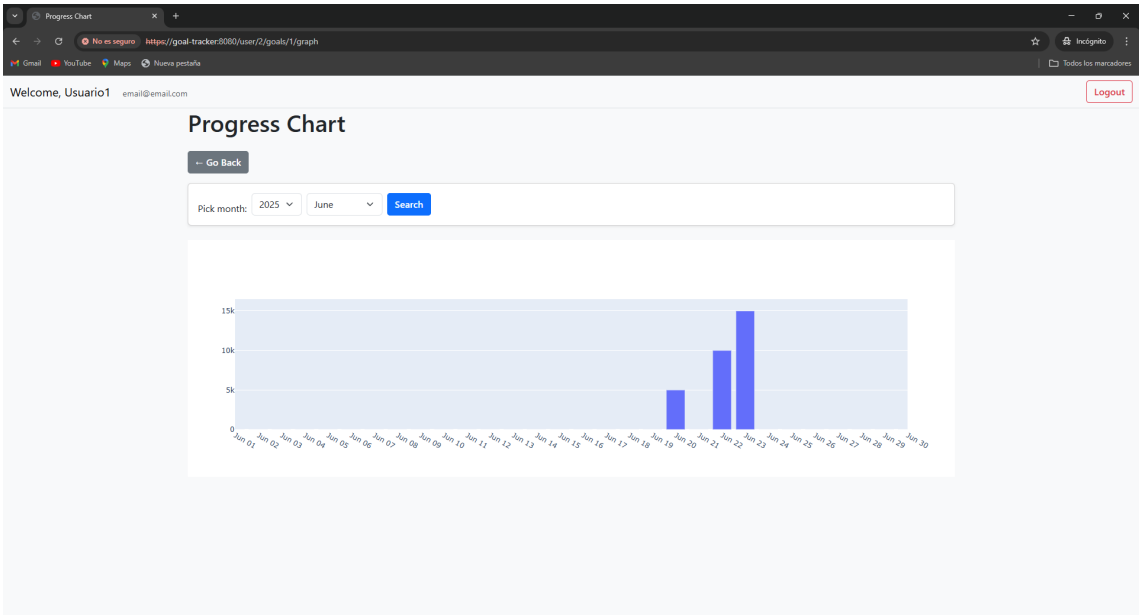


Figura 153: Vista de la gráfica

Se pueden buscar otros meses, pero todos estan registrados en el mes de Junio, por lo tanto se vera que en otros meses no hay datos:

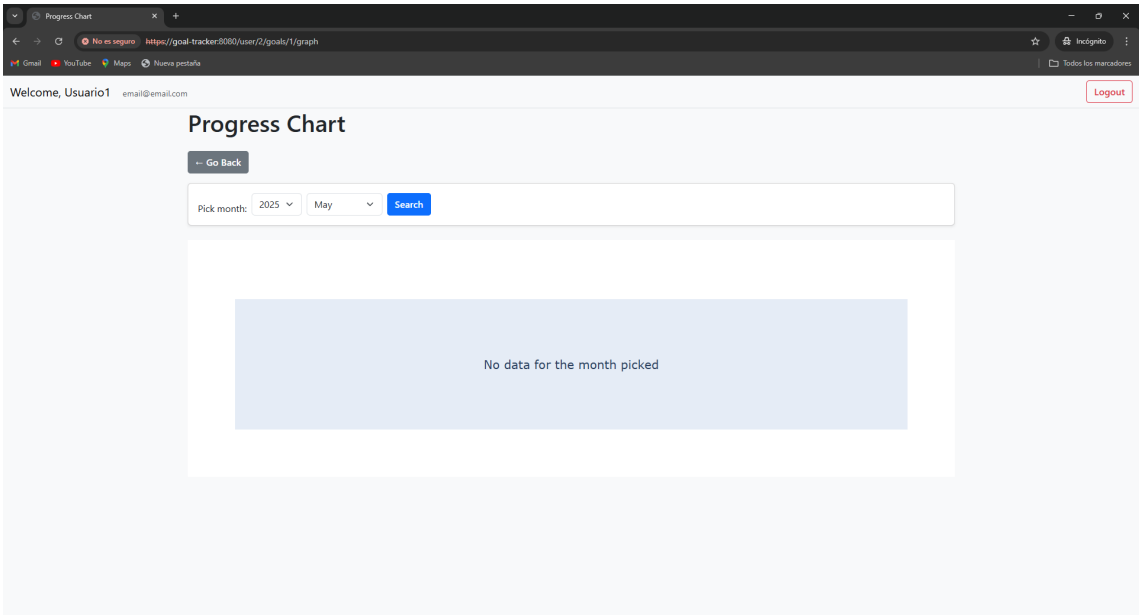


Figura 154: Búsqueda de un mes

También se puede ver la cantidad de progreso en un día específico poniendo el puntero del ratón sobre la barra de un día específico. Además aunque pueda haber varios avances registrados en un mismo día, en el gráfico se muestran sumados en el mismo día:

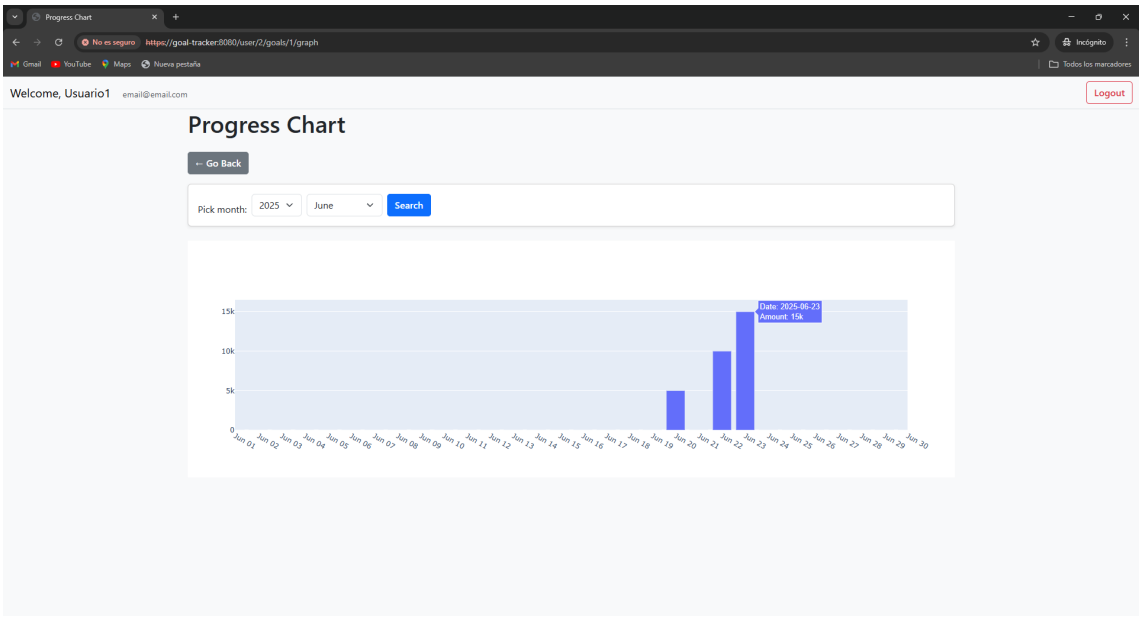


Figura 155: Información adicional al poner el puntero encima de una barra

Finalmente, si se pulsa sobre una de las barras, el navegador redirige a la vista donde se ven los progresos de la meta, filtrada por el día específico seleccionado:

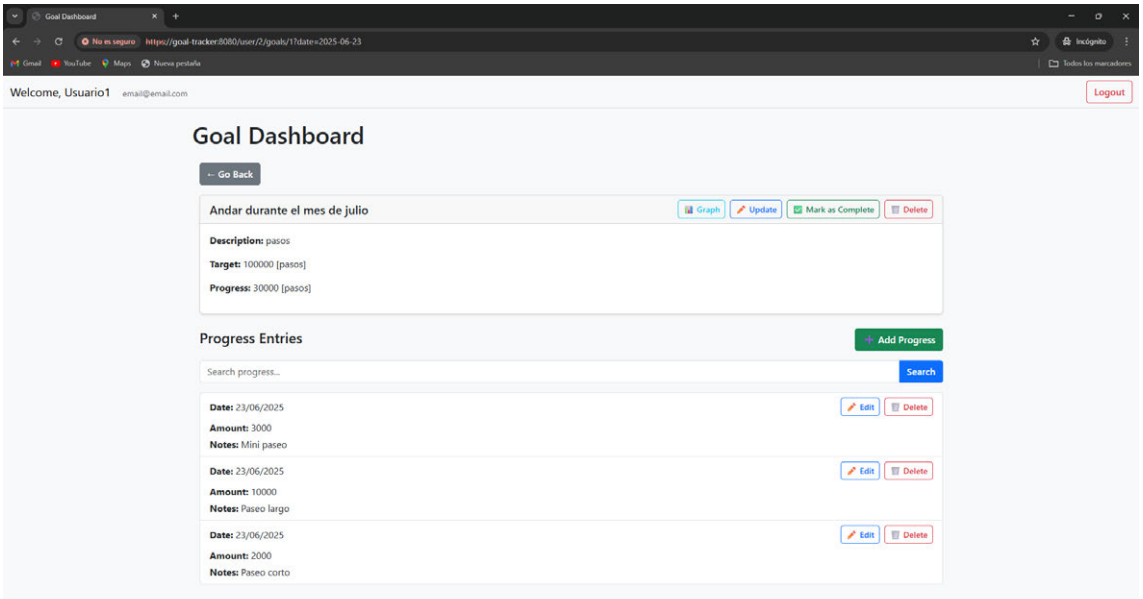


Figura 156: Resultado de presionar la barra

5.6. Exportar Datos

En la vista general donde se ven todas las metas, en la zona superior de la derecha, se aprecia un botón con tres puntos. Al presionar este botón, se muestran dos opciones, exportar como `CSV` o `PDF`.

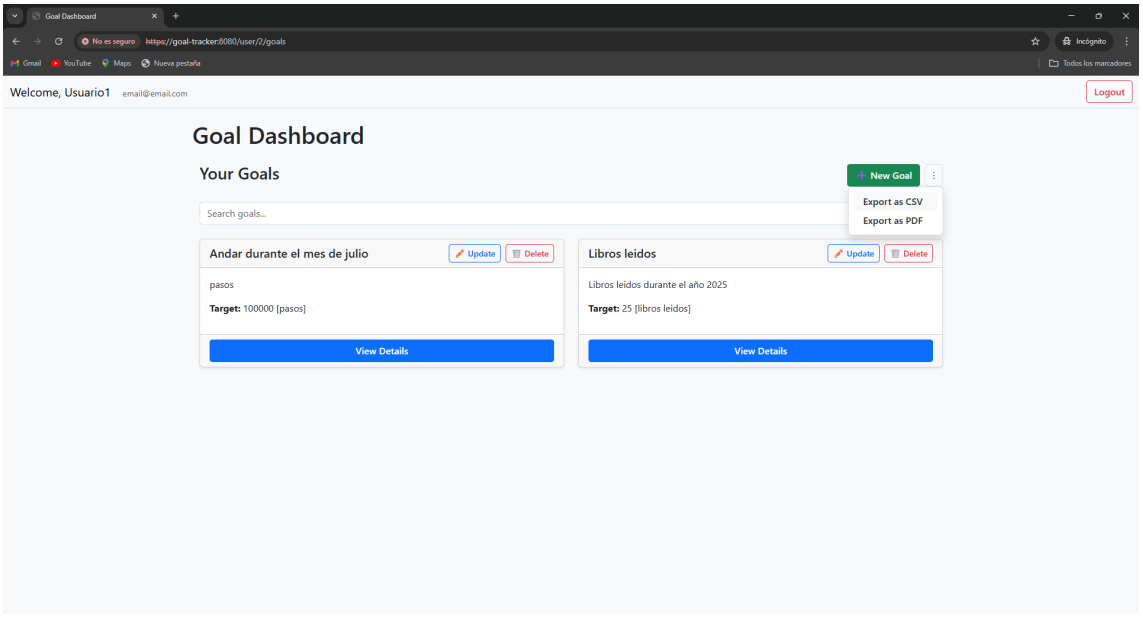


Figura 157: Menu de exportar datos

El resultado de exportar como `CSV`:

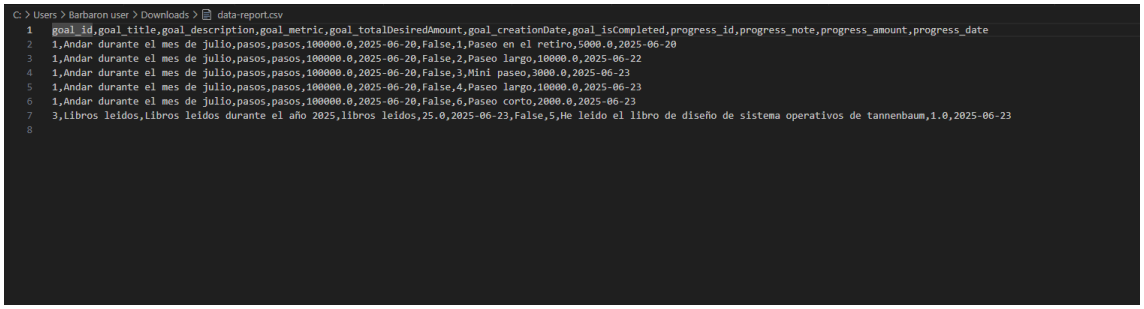
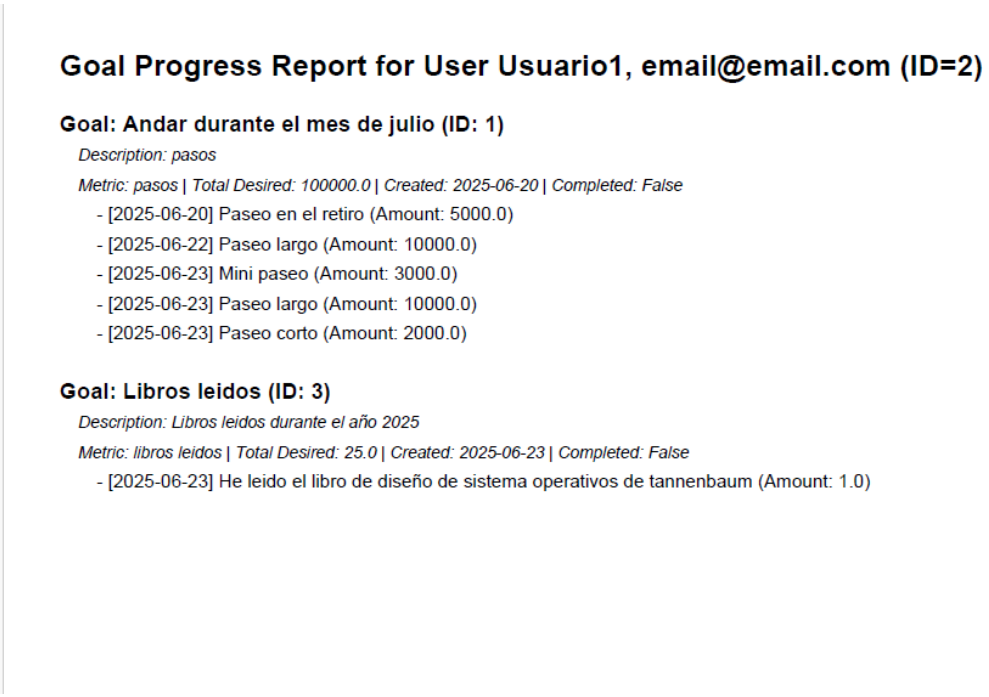


Figura 158: Resultado de exportar csv

El resultado de exportar como PDF:



Goal Progress Report for User Usuario1, email@email.com (ID=2)

Goal: Andar durante el mes de julio (ID: 1)

Description: pasos

Metric: pasos | Total Desired: 100000.0 | Created: 2025-06-20 | Completed: False

- [2025-06-20] Paseo en el retiro (Amount: 5000.0)
- [2025-06-22] Paseo largo (Amount: 10000.0)
- [2025-06-23] Mini paseo (Amount: 3000.0)
- [2025-06-23] Paseo largo (Amount: 10000.0)
- [2025-06-23] Paseo corto (Amount: 2000.0)

Goal: Libros leidos (ID: 3)

Description: Libros leidos durante el año 2025

Metric: libros leidos | Total Desired: 25.0 | Created: 2025-06-23 | Completed: False

- [2025-06-23] He leido el libro de diseño de sistema operativos de tannenbaum (Amount: 1.0)

Figura 159: Resultado de exportar pdf

5.7. Vista de administración

A continuación se inicia sesión con el administrador por defecto `admin@goal.tracker`. Su contraseña se define en un secreto de kubernetes. Al iniciar sesión, los administradores ven a todos los usuarios y además pueden ver si cada usuario es un administrador o está vetado de la aplicación. Además puede ver las metas de otros usuarios y gestionarlas:

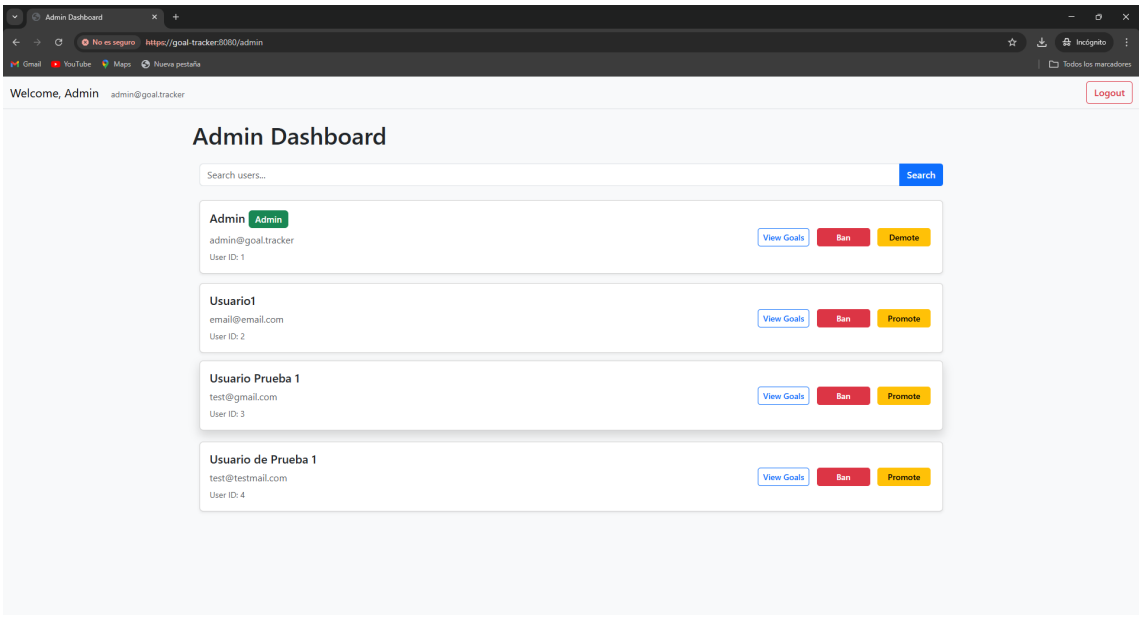


Figura 160: Vista de administrador

5.7.1. Vetar un usuario

Se veta un usuario:

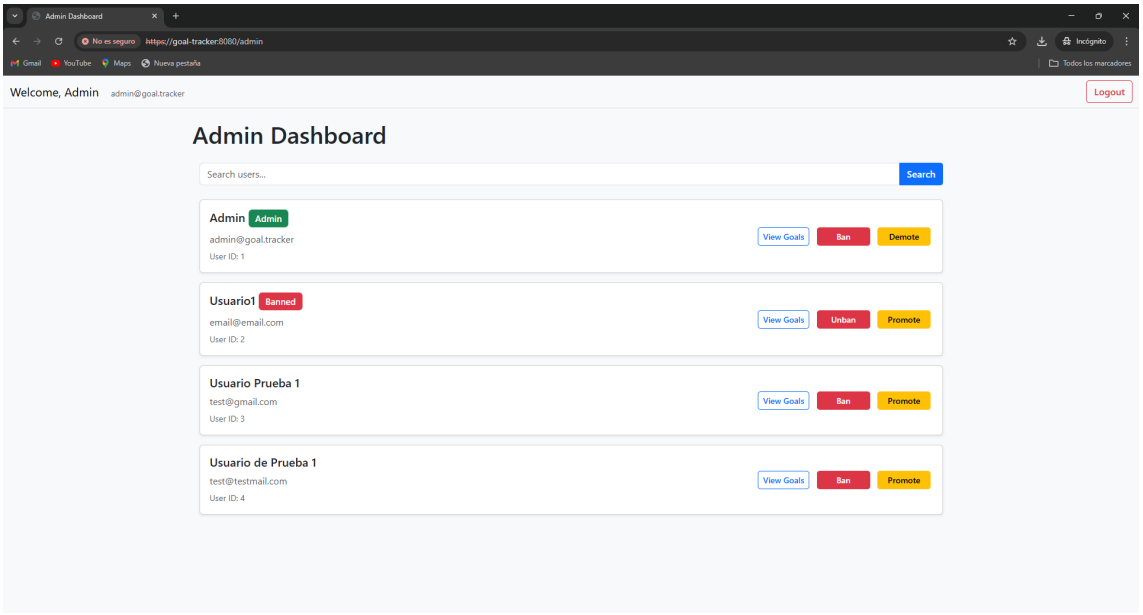


Figura 161: Usuario vetado

5.7.2. Promocionar usuario a administrador

Se promociona un usuario a nivel administrativo:

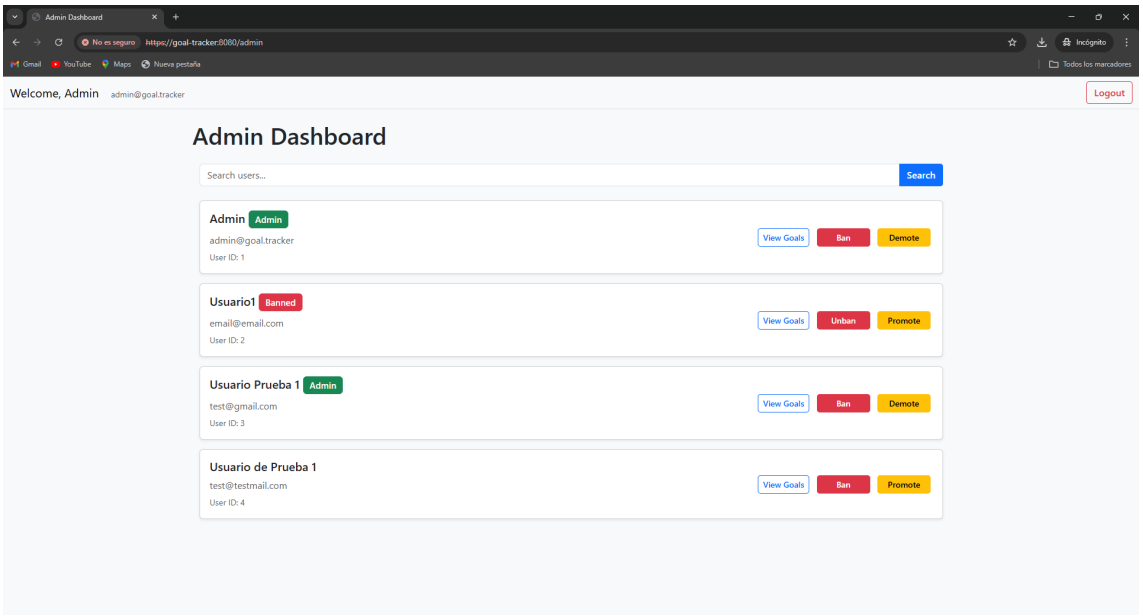


Figura 162: Nuevo administrador, ahora hay 2

5.8. Escalabilidad

Una de las principales ventajas de haber implementado la aplicación sobre un entorno orquestado con Kubernetes es la facilidad con la que se puede escalar horizontalmente. Esto permite aumentar o reducir la capacidad del sistema según la demanda, garantizando así mayor disponibilidad y tolerancia a fallos.

Para escalar un servicio específico, como por ejemplo el servidor `flask-frontend`, basta con ejecutar el siguiente comando:

```
kubectl scale deployment --replicas=2 flask-frontend -n prod
```

En este caso, el número de réplicas del frontend se incrementa a 2, lo que significa que se ejecutan dos pods idénticos del servidor Flask. Esto permite distribuir las solicitudes entrantes entre ambos pods, mejorando el rendimiento y la capacidad de respuesta del sistema.

La correcta distribución del tráfico entre las réplicas puede observarse a través de los registros del `Ingress`, como se muestra en la siguiente figura:

Figura 163: Logs del Ingress mostrando balanceo de carga entre pods

En los logs se aprecia cómo el `Ingress Controller` balancea las peticiones entre dos direcciones IP internas del clúster (`10.244.1.1` y `10.244.1.2`), correspondientes a los dos pods desplegados. Esto confirma el funcionamiento adecuado del balanceo de carga y demuestra la capacidad de escalado dinámico de la aplicación.

6. Conclusiones

Este proyecto ha cumplido con los requisitos principales, permitir que el usuario pueda gestionar sus metas y además que la aplicación tenga un diseño de microservicios.

El proyecto ha representado un proceso largo y exigente, que me ha permitido no solo consolidar conocimientos previamente adquiridos, sino también adquirir nuevas competencias técnicas esenciales para el desarrollo de aplicaciones modernas.

A lo largo de su desarrollo, he aprendido en profundidad conceptos clave como el funcionamiento del sistema de autenticación mediante JWT (JSON Web Tokens), lo que me ha permitido comprender mejor la gestión de la seguridad en aplicaciones web.

Además, quiero destacar el valor de las prácticas profesionales que he realizado, ya que han sido fundamentales para poder aplicar estos conocimientos en un entorno real. Gracias a ellas, he aprendido a construir un backend robusto y mantenible, así como a utilizar herramientas clave como Docker y Kubernetes para el despliegue y la orquestación de servicios.

En conjunto, este trabajo ha supuesto una experiencia muy enriquecedora tanto a nivel técnico como personal, que sin duda marca un antes y un después en mi formación como desarrollador.

7. Futuras ampliaciones

Aunque el proyecto ha alcanzado los objetivos propuestos, existen varias áreas susceptibles de mejora y ampliación que podrían abordarse en el futuro:

- **Mejora visual de la interfaz**: Especialmente en lo que respecta a los cuadros de diálogo y elementos interactivos, con el objetivo de proporcionar una experiencia de usuario más pulida y profesional.
- **Segmentación del backend**: Actualmente, el backend está diseñado como un único bloque monolítico. Una futura mejora podría consistir en dividirlo en servicios más específicos (por ejemplo, servicios de autenticación, gestión de usuarios, etc.) siguiendo una arquitectura más estricta de microservicios.
- **Despliegue en un entorno real**: Actualmente, la solución se ejecuta en Minikube, un entorno de desarrollo local. Como mejora, sería recomendable desplegar los servicios en un clúster real (por ejemplo, en Google Kubernetes Engine o AWS EKS), lo que permitiría evaluar aspectos como la escalabilidad, la tolerancia a fallos y el rendimiento bajo carga real.
- **Implementación de un sistema de caché**: Incorporar una capa de caché para el almacenamiento de recursos estáticos, como imágenes generadas o archivos descargables, mejoraría considerablemente los tiempos de respuesta y reduciría la carga sobre el backend.
- **Implementar OAuth 2.0:** Como mejora en el sistema de autenticación, se propone integrar OAuth 2.0 para permitir el inicio de sesión a través de proveedores externos (como Google, Facebook o GitHub). Esto facilitaría a los usuarios acceder a la aplicación sin necesidad de crear nuevas credenciales, mejoraría la seguridad mediante flujos de autorización estándar y aumentaría la escalabilidad y flexibilidad del sistema de autenticación, preparando la aplicación para integraciones futuras y entornos más complejos.

8. Impacto Legal, Social y Ético

8.1. Impacto Legal

El proyecto implica el tratamiento de datos personales vinculados a los objetivos individuales de los usuarios, lo cual obliga a considerar aspectos legales importantes. Aunque actualmente no se trata de una aplicación pública, si se escalara o distribuyera a terceros, debería cumplir con normativas como el Reglamento General de Protección de Datos (RGPD). Es especialmente relevante que el administrador de la aplicación tenga acceso a todas las metas de los usuarios, lo que plantea la necesidad de definir políticas claras de privacidad, justificación del acceso administrativo y mecanismos de auditoría.

8.2. Impacto Social

Desde un punto de vista social, la aplicación ofrece una herramienta útil para fomentar la organización personal, la autorreflexión y la consecución de metas. Puede ayudar a personas que buscan mejorar su productividad o adoptar hábitos saludables.

8.3. Impacto Ético

Ética y responsabilidad son aspectos claves en el diseño de esta solución. El acceso del administrador a todas las metas plantea un dilema ético sobre la privacidad del usuario. Aunque puede estar justificado por necesidades técnicas o de soporte, sería necesario limitar ese acceso o, al menos, registrar y auditar su uso. La aplicación debe garantizar que los usuarios estén informados de esta posibilidad. En el resto del diseño, se han seguido principios éticos como la no manipulación del comportamiento, la ausencia de publicidad o monetización invasiva, y la promoción de hábitos saludables.

9. Repositorio

El repositorio tiene alojado todo el código fuente, scripts auxiliares (que no estan explicados en el tfg debido a su utilidad exclusiva de desarrollo) y ficheros de despliegue. El repositorio no contiene explicación de como ejecutar el entorno pero con un conocimiento básico de kuberntes sumado a las explicaciones del propio tfg es posible desplegar el proyecto.

Enlace al repositorio: `https://github.com/CEmilovUPM/TFG/`

10. Referencias Bibliográficas

Referencias

[1] Spring.io, *Spring Boot Project*. Disponible en: `https://spring.io/projects/spring-boot`.

[2] Spring.io, *Spring Security Project*. Disponible en: `https://spring.io/projects/spring-security`.

[3] JWT.io, *Introduction to JSON Web Tokens*. Disponible en: `https://jwt.io/introduction`.

[4] Pallets Projects, *Flask Documentation*. Disponible en: `https://flask.palletsprojects.com`.

[5] Plotly Technologies Inc., *Plotly Python Graphing Library*. Disponible en: `https://plotly.com/python/`.

[6] Gunicorn Developers, *Gunicorn Documentation*. Disponible en: `https://gunicorn.org`.

[7] The Bootstrap Team, *Bootstrap Documentation*. Disponible en: `https://getbootstrap.com`.

[8] OWASP Foundation, *OWASP Validation Regex Repository*. Disponible en: `https://owasp.org/www-community/OWASP_Validation_Regex_Repository`.

[9] Minikube. *Run Kubernetes locally*. The Kubernetes Authors. Disponible en: `https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download`.