



Proyecto Final TFG

MemberFlow

**Sistema de Gestión Integral para escuelas de Artes
Marciales**

Autor: Dennis Eckerskorn

Curso: Desarrollo de Aplicaciones Multiplataforma (DAM)

Año: 2023 – 2025

Índice:

Tabla de contenido

Índice:	2
Introducción:	5
Objetivos:	6
Objetivo general:	6
Objetivos específicos:	6
Justificación:	7
Planificación:	8
Metodología empleada	9
Enfoque modular	9
Estructura modular del sistema:	10
Módulo MemberFlow-Data	10
1. Gestión de Usuarios (user_management)	10
2. Gestión de Clases (class_management)	12
3. Gestión Financiera (finance)	13
Repositorios y Servicios MemberFlow-Data:	15
Clase abstracta AbstractService:	15
Repositorios:	16
Servicios:	17
Carga de datos TestDataSeeder:	18
Perfil de desarrollo (dev):	19
Pruebas unitarias:	20
Módulo MemberFlow-API:	21
Estructura del proyecto:	21
Seguridad y Autenticación	22
Flujo de autenticación:	22
Configuración principal:	22
Roles y permisos	22
Validación y Manejo de Errores	23
Características:	23
Documentación automática (Swagger)	24
Controladores y Endpoints	28

Estructura y patrón seguido.....	28
DTOs y Mapeo.....	29
Funciones del DTO:.....	29
Integración con memberflow-data	31
Inyección de dependencias	31
Generación de facturas en PDF	32
Módulo MemberFlow-Frontend	33
Estructura del proyecto.....	33
Tecnologías utilizadas.....	34
Control de acceso y autenticación.....	35
Funcionalidades desarrolladas.....	37
Autenticación y control de acceso	37
Gestión de usuarios.....	37
Gestión de estudiantes.....	37
Gestión de profesores y administradores	38
Gestión del historial del estudiante	38
Gestión de notificaciones.....	38
Gestión de clases y entrenamientos.....	38
Gestión de asistencias	39
Gestión de membresías	39
Gestión de productos y tipos de IVA.....	39
Gestión de facturación.....	39
Gestión de pagos	40
Generación de facturas en PDF	40
Herramientas utilizadas.....	41
Contenerización con Docker	42
Estructura de archivos relevantes.....	42
Servicios definidos.....	42
Presupuesto estimado	43
Equipamiento e infraestructura:	43
Licencias y Herramientas	43
Horas de desarrollo	44
Estimación de viabilidad y precio de comercialización	45
Conclusiones	46

Viabilidad y proyección comercial.....	46
Futuras ampliaciones.....	47
Bibliografía.....	48

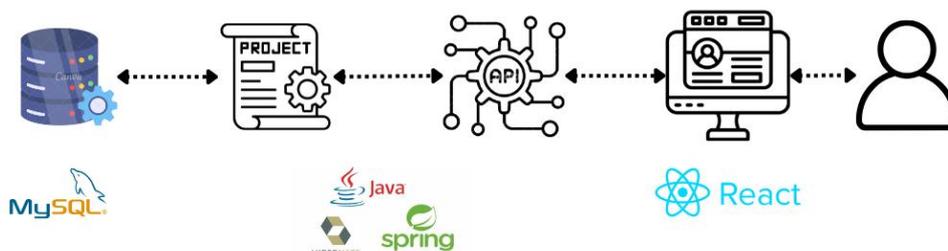
Introducció:

MemberFlow es un sistema de gestión integral diseñado para escuelas de artes marciales. Su objetivo principal es facilitar la administración de estudiantes, profesores, clases o entrenamientos, facturación y membresías.

El proyecto está estructurado en tres módulos bien diferenciados:

- **MemberFlow-Data:** Es el módulo encargado de establecer la conexión con la base de datos, gestionar las entidades del sistema y mapearlas con las tablas correspondientes. Además, incluye los servicios responsables de la lógica de negocio relacionada con la inserción, actualización, eliminación y consulta de datos. En otras palabras, implementa las operaciones CRUD fundamentales de la aplicación.
- **MemberFlow-API:** Este módulo expone la API del sistema, proporcionando los diferentes endpoints necesarios para la comunicación entre el frontend y la base de datos. Gestiona los controladores, la seguridad de acceso (autenticación y autorización) y la recepción/envío de datos hacia el módulo de persistencia.
- **MemberFlow-Frontend:** Es la interfaz visual del sistema, desarrollada para interactuar con los usuarios. Consume los endpoints de la API y permite una gestión intuitiva y eficiente de todas las funcionalidades del sistema a través de una interfaz gráfica.

MemberFlow



Objetivos:

Objetivo general:

Desarrollar un sistema web completo y funcional, denominado MemberFlow, orientado a la gestión integral de escuelas de artes marciales. El sistema debe permitir la administración eficiente de usuarios, clases, membresías, asistencias, pagos y facturación, proporcionando una herramienta práctica y moderna para academias. Como objetivo a largo plazo, se contempla la posibilidad de comercializar el producto una vez finalizado y validado su funcionamiento.

Objetivos específicos:

- Diseñar e implementar una arquitectura modular compuesta por tres partes: Backend (MemberFlow-Data), API REST (MemberFlow-API) y Frontend (MemberFlow-Frontend).
- Establecer una base de datos relacional que permita almacenar y gestionar información sobre usuarios, estudiantes, profesores, clases, asistencias, roles, permisos, membresías y pagos.
- Implementar un sistema de autenticación y autorización basado en JWT, que permita restringir el acceso a las funcionalidades según el rol del usuario (administrador, profesor o estudiante).
- Desarrollar formularios y vistas interactivas en el frontend que permitan crear, editar, listar y eliminar registros de forma intuitiva.
- Generar facturas en formato PDF y llevar un control de pagos asociados a cada alumno o membresía.
- Aplicar principios de seguridad como el cifrado de contraseñas y la validación de entradas del usuario.
- Realizar pruebas unitarias y de integración para verificar el correcto funcionamiento de los servicios del sistema.
- Documentar correctamente la API utilizando herramientas como Swagger/OpenAPI.
- Diseñar una interfaz de usuario moderna y responsive para facilitar el uso del sistema desde distintos dispositivos.
- Desplegar la aplicación localmente o en un entorno de producción simulado, preparando su posible publicación futura.

Justificació:

En la actualitat, muchas escuelas de artes marciales continúan utilizando métodos tradicionales o herramientas poco especializadas para gestionar sus actividades diarias, tales como hojas de cálculo, documentos manuales o aplicaciones genéricas que no se adaptan a sus necesidades específicas. Esta situación puede generar ineficiencias, errores en la gestión de pagos o membresías, y una experiencia poco fluida tanto para los gestores como para los alumnos.

MemberFlow nace con el objetivo de ofrecer una solución específica y completa para academias de artes marciales, permitiendo centralizar y automatizar la gestión de usuarios, clases, asistencia, membresías y facturación en un entorno accesible, moderno y seguro.

El desarrollo de este sistema no solo permite aplicar conocimientos adquiridos durante el ciclo formativo, como el diseño de bases de datos, la programación en Java, el uso de frameworks como Spring Boot y React, o la implementación de APIs REST, sino que también supone una oportunidad real de aportar valor al sector deportivo y educativo.

Además, el proyecto se ha planteado con una estructura escalable y modular, lo que facilita su mantenimiento, evolución futura y eventual comercialización como producto SaaS (Software as a Service) para academias que deseen mejorar su gestión interna.

Planificació:

El desarrollo del proyecto MemberFlow se ha planteado desde un enfoque modular, iniciando con la planificación y el diseño de la base de datos, elemento clave en la arquitectura del sistema. Esta planificación se ha organizado en torno a tres áreas funcionales principales, cada una con sus propias entidades, relaciones y servicios asociados:

- Gestión de Usuarios (User Management)
- Gestión de Clases (Class Management)
- Gestión Financiera (Finance)

El diseño de la base de datos se llevó a cabo mediante un diagrama entidad-relación (ER), en el que se definieron las entidades necesarias, sus claves primarias y foráneas, así como las relaciones entre las distintas tablas. Este modelo se implementó directamente en el módulo MemberFlow-Data, que constituye la capa de persistencia y lógica de negocio del sistema.

Para el mapeo objeto-relacional entre las entidades Java y las tablas de la base de datos, se utilizó Hibernate (ORM), empleando anotaciones para definir atributos, claves y relaciones entre entidades. Este módulo contiene toda la lógica de acceso a datos, pero no gestiona directamente la configuración del contexto de ejecución.

Cabe destacar que el archivo de configuración `application.properties` no se encuentra en `memberflow-data`, sino en el módulo `MemberFlow-API`, que es el encargado de iniciar la aplicación Spring Boot y establecer las configuraciones globales. Este archivo define, entre otros aspectos, la conexión con la base de datos, el dialecto de Hibernate, el comportamiento de creación de esquemas (`ddl-auto`), y otras propiedades relevantes del entorno de ejecución.

La clase `HibernateConfig`, presente en el módulo `memberflow-data`, complementa esta configuración permitiendo definir aspectos específicos del gestor de entidades y las fuentes de datos, en coordinación con el archivo `application.properties` del módulo `API`.

Metodología empleada

El desarrollo del sistema MemberFlow se ha llevado a cabo siguiendo una metodología ágil adaptada, inspirada en los principios del modelo Kanban, lo que ha permitido un enfoque flexible y visual del avance de tareas. Aunque no se utilizó una herramienta de gestión formal desde el inicio (como Jira o Trello), se estableció una planificación clara por bloques funcionales, lo que facilitó el seguimiento del progreso, la resolución de errores por capas y la integración gradual de funcionalidades.

Enfoque modular

El proyecto se organizó en torno a tres módulos principales, cada uno desarrollado de manera progresiva e independiente, favoreciendo la cohesión interna y la mantenibilidad del sistema:

MemberFlow-Data – Diseño de la base de datos y lógica de negocio

Se inició el desarrollo definiendo el modelo de datos mediante un diagrama entidad-relación (ER). A partir de este diseño, se implementaron las entidades JPA, relaciones entre tablas, restricciones y servicios. También se configuró el uso de Hibernate para el mapeo objeto-relacional y se organizaron los paquetes por dominios funcionales: gestión de usuarios, clases y finanzas.

MemberFlow-API – Desarrollo de la API REST y seguridad

Con la base de datos y los servicios definidos, se procedió a implementar la capa de exposición mediante controladores REST, DTOs y mapeadores. Además, se incorporó un sistema de autenticación basado en JWT (JSON Web Tokens) con control de acceso por roles (ADMIN, TEACHER, STUDENT) y configuración de seguridad mediante Spring Security.

MemberFlow-Frontend – Construcción de la interfaz visual

Finalmente, se desarrolló el frontend con React.js, organizando los componentes por dominios funcionales. Se implementaron formularios, paneles de usuario y listados conectados a la API REST mediante Axios. El sistema permite el login por rol, redirección dinámica, visualización de datos y administración completa desde el panel de administrador.

Este enfoque por módulos facilitó el desarrollo incremental y permitió detectar errores más fácilmente al probar cada capa por separado.

Estructura modular del sistema:

Módulo MemberFlow-Data

Este módulo constituye la capa de acceso a datos y lógica de negocio del sistema. Su función principal es gestionar las entidades del dominio, sus relaciones y los servicios que implementan las operaciones fundamentales (creación, lectura, actualización y eliminación).

A nivel de organización, el módulo está estructurado por dominios funcionales, facilitando la mantenibilidad y escalabilidad del sistema. Uno de estos dominios clave es el de Gestión de Usuarios, cuya arquitectura ha sido diseñada con especial atención a la reutilización y la claridad.

1. Gestión de Usuarios (user_management)

Este paquete se encarga de manejar todos los datos y operaciones relacionados con los diferentes tipos de usuarios: estudiantes (Student), profesores (Teacher) y administradores (Admin).

En lugar de utilizar herencia directa entre entidades, se ha optado por un enfoque basado en **composición**, lo que significa que cada entidad específica (Student, Teacher, Admin) contiene una instancia de la entidad User. Este diseño tiene varias ventajas:

- Mayor flexibilidad: Permite definir atributos particulares para cada tipo de usuario sin heredar comportamientos innecesarios.
- Reutilización de lógica: La entidad User agrupa todos los campos comunes (nombre, email, contraseña, etc.), evitando la duplicación de código.
- Identificación unificada: Cada entidad Student, Teacher y Admin comparte el mismo ID que su respectivo User, lo que simplifica la gestión y trazabilidad de los registros en la base de datos.
- Mejor control de relaciones: Facilita la integración con sistemas de autenticación, roles y permisos, ya que todos los accesos se gestionan desde el User.

Este diseño se refleja claramente en la estructura de clases, donde:

- User es la entidad base con la información compartida.
- Student, Teacher y Admin contienen una propiedad User user, que representa su identidad principal.
- Los servicios correspondientes se encargan de mantener la coherencia al guardar o modificar tanto el User como su entidad específica asociada.

User (Usuario):

Entidad base que representa la información común a todos los tipos de usuario del sistema. Incluye campos como nombre, email, contraseña (encriptada), estado de cuenta y fecha de registro. Es utilizada por las entidades Student, Teacher y Admin a través de composición.

Entidad Admin (Administrador):

Entidad asociada a User, utilizada para identificar a los usuarios con rol administrativo. Estos usuarios tienen acceso completo al sistema, incluyendo la gestión de usuarios, clases, facturación y configuración.

Entidad Student (Alumno):

Entidad específica que representa a un estudiante de la academia. Incluye datos adicionales como fecha de nacimiento, nivel de experiencia y relación con su historial de entrenamiento, membresías y asistencias.

Entidad Teacher (Profesores):

Entidad que representa a los instructores del sistema. Además de la información común del User, puede incluir información como especialidad o disciplinas impartidas. Está relacionado con los grupos de entrenamiento que dirige.

Role (Roles):

Define el rol asignado a cada usuario (ADMIN, TEACHER, STUDENT). Cada rol determina el conjunto de permisos que controlan qué acciones puede realizar un usuario dentro del sistema.

Permission (Permisos):

Especifica las acciones concretas que un usuario puede realizar. Está relacionado con uno o más roles y puede incluir acciones como: "gestionar estudiantes", "acceso total" o "ver datos propios".

Entidad Notification (Notificaciones):

Permite enviar mensajes internos a los usuarios. Se utiliza, por ejemplo, para notificar cambios en horarios, pagos pendientes o logros personales del alumno.

Entidad StudentHistory (Historial de estudiantes):

Registra eventos relevantes en la trayectoria del estudiante, como progresos, asistencia destacada, logros o participación en torneos. Facilita el seguimiento personalizado del alumno.

2. Gestión de Clases (class_management)

El paquete class_management se encarga de gestionar todos los aspectos relacionados con la organización de los entrenamientos, incluyendo la creación de grupos, la planificación de sesiones y el control de asistencias de los estudiantes. Esta parte del sistema está diseñada para facilitar una gestión estructurada y automatizada de las clases.

Las entidades principales que conforman este módulo son:

- **TrainingGroup (Grupo de entrenamiento):** Representa un grupo de entrenamiento dirigido por un profesor y compuesto por varios estudiantes. Cada grupo tiene asignado un nombre, un horario específico y un responsable (Teacher). Los grupos permiten estructurar la enseñanza en base a niveles, edades o categorías.
- **TrainingSession (Sesión de entrenamiento):** Se trata de una sesión concreta asociada a un grupo de entrenamiento. Cada vez que se crea un TrainingGroup, el sistema genera automáticamente una sesión inicial con la misma fecha y horario, lo que simplifica la planificación desde el backend y permite registrar asistencias desde el primer momento. El sistema está preparado para extender esta funcionalidad y generar sesiones recurrentes en el futuro.
- **Assistance (Asistencias):** Registra la asistencia de los estudiantes a cada sesión. Esto permite llevar un control detallado de la participación de los alumnos y generar métricas de seguimiento.
- **Membership (Membresías):** Relaciona a cada estudiante con su tipo de membresía activa, lo que permite validar su participación en entrenamientos según su estado de suscripción.

Desde el frontend, la creación de grupos se realiza mediante formularios que permiten asignar un profesor, un horario, y un nombre al grupo. Una vez creado, se puede visualizar automáticamente una agenda semanal, donde se muestran las sesiones organizadas por día y hora. Esta vista permite a los profesores o administradores:

- Visualizar rápidamente qué sesiones están planificadas para cada grupo.
- Acceder a los registros de asistencia de cada sesión.
- Añadir o modificar estudiantes en cada grupo.
- Consultar o cambiar el profesor asignado a un grupo específico.

Gracias a este diseño, la aplicación permite gestionar de forma eficiente tanto la estructura de clases como el seguimiento individualizado de los alumnos, asegurando una experiencia fluida para los responsables de la academia.

3. Gestión Financiera (finance)

El módulo de Gestión Financiera se encarga de administrar toda la información económica del sistema, incluyendo la facturación de productos o servicios, el control de pagos realizados por los estudiantes, y la gestión de impuestos aplicables. Este módulo permite llevar un seguimiento detallado de los ingresos de la academia, asegurando transparencia y control sobre las operaciones financieras.

Las principales entidades que conforman este módulo son:

Invoice (Factura):

Entidad principal que representa un documento de cobro emitido a un estudiante. Contiene información como la fecha de emisión, el estado de pago, el importe total, y la relación con el usuario correspondiente. Cada factura puede contener múltiples líneas (InvoiceLine) que detallan los productos o servicios incluidos.

InvoiceLine (Línea de Factura):

Cada factura se compone de una o más líneas que indican los conceptos facturados. Cada línea incluye el producto o servicio, la cantidad, el precio unitario, el tipo de IVA aplicable y el subtotal. Esta estructura permite generar facturas detalladas y transparentes para los alumnos.

Payment (Pago):

Entidad encargada de registrar los pagos realizados por los estudiantes. Cada pago está vinculado a una factura y puede representar un pago total o parcial. También se almacena la fecha del pago, el importe abonado y el método de pago utilizado.

Para definir los métodos de pago se utiliza un enumerado, que contempla opciones como:

- Efectivo
- Tarjeta
- Transferencia

Esto permite validar los tipos de pago desde el backend y mantener consistencia en los datos almacenados.

ProductService (Productos y/o Servicios):

Define los productos o servicios disponibles que pueden ser facturados a los alumnos. Esto incluye elementos como membresías, uniformes, seminarios, clases particulares u otros accesorios. Cada producto cuenta con un nombre, una descripción, un precio y un tipo de IVA asociado.

Repositorios y Servicios MemberFlow-Data:

El módulo MemberFlow-Data sigue una arquitectura basada en repositorios y servicios, cumpliendo con el principio de separación de responsabilidades. Como se ha explicado previamente, este módulo gestiona la lógica de negocio y la persistencia de datos, siendo la capa que conecta directamente con la base de datos.

Esta sección describe cómo se organizan los repositorios, cómo funcionan los servicios, la lógica común extraída en una clase abstracta, y cómo se aplican pruebas unitarias.

Clase abstracta AbstractService:

Para evitar la duplicación de lógica en los distintos servicios del sistema, se ha implementado una clase abstracta genérica denominada `AbstractService<T, ID>`, que centraliza los métodos comunes como:

- `save`: Guardar una entidad en la base de datos.
- `update`: Actualizar una entidad existente.
- `getEntityId`: Obtener el ID de una entidad (método protegido que se sobrescribe).
- `findById`: Recuperar una entidad por su ID.
- `findAll`: Listar todas las entidades.
- `deleteById`: Eliminar una entidad por ID.
- `exists`: Verificar si una entidad existe en la base de datos.

Esto permite que cada servicio concreto extienda esta clase y herede automáticamente esta funcionalidad, lo que favorece la reutilización y coherencia. Ejemplo de definición de la clase `AbstractService`:

```
@Service 5 usages ⚙ Dennis Eckerskom
public class StudentService extends AbstractService<Student, Integer> {

    private static final Logger logger = LoggerFactory.getLogger(StudentService.class); 35 usages
    private final StudentRepository studentRepository; 4 usages
    private final UserRepository userRepository; 2 usages
    private final AssistanceRepository assistanceRepository; 2 usages

    /**
     * Constructor for StudentService.
     *
     * @param studentRepository the student repository
     */
    public StudentService(StudentRepository studentRepository, UserRepository userRepository, AssistanceRepository assistanceRepository) {
        super(studentRepository);
        this.studentRepository = studentRepository;
        this.userRepository = userRepository;
        this.assistanceRepository = assistanceRepository;
    }
}
```

Repositorios:

Cada entidad principal del sistema dispone de su correspondiente interfaz repositorio, la cual extiende JpaRepository. Esto permite aprovechar los métodos genéricos proporcionados por Spring Data JPA, como findAll, findById, deleteById, entre otros.

Por ejemplo, el repositorio de Student:

```
public interface StudentRepository extends JpaRepository<Student, Integer> {  
    boolean existsByDni(String dni); 3 usages @ Dennis Eckerskorn  
    Student findByDni(String dni); 2 usages @ Dennis Eckerskorn  
    boolean existsByUserEmail(String email); no usages @ Dennis Eckerskorn  
    Student findByUserEmail(String email); no usages @ Dennis Eckerskorn  
}
```

Gracias a este enfoque, se pueden realizar búsquedas comunes sin necesidad de definir consultas SQL explícitas. Este patrón se repite en todas las entidades del sistema (UserRepository, InvoiceRepository, PaymentRepository, etc.), lo cual simplifica el desarrollo y reduce la cantidad de código repetido.

Servicios:

Los servicios implementan la lógica de negocio y actúan como intermediarios entre los controladores (en memberflow-api) y los repositorios (en memberflow-data). Se encargan de:

- Validar la información antes de guardarla.
- Gestionar relaciones entre entidades (por ejemplo, guardar un User antes de un Student).
- Aplicar reglas específicas del dominio.

La inyección de dependencias de Spring facilita el uso de los repositorios dentro de los servicios mediante constructores o anotaciones como @Autowired.

Registros de eventos con Logger:

Para facilitar la depuración y el seguimiento del comportamiento del sistema durante su ejecución, se ha utilizado la clase Logger de org.slf4j. Esto permite generar mensajes personalizados, indicando el flujo de ejecución, operaciones exitosas o errores.

Un ejemplo de la implementación sería:

```
@Service 5 usages ⚙️ Dennis Eckerskorn
public class StudentService extends AbstractService<Student, Integer> {

    private static final Logger logger = LoggerFactory.getLogger(StudentService.class); 35 usages
    private final StudentRepository studentRepository; 4 usages
    private final UserRepository userRepository; 2 usages
    private final AssistanceRepository assistanceRepository; 2 usages

    /**
     * Constructor for StudentService.
     *
     * @param studentRepository the student repository
     */
    public StudentService(StudentRepository studentRepository, UserRepository userRepository, AssistanceRepository assistanceRepository) {
        super(studentRepository);
        this.studentRepository = studentRepository;
        this.userRepository = userRepository;
        this.assistanceRepository = assistanceRepository;
    }

    @Override ⚙️ Dennis Eckerskorn
    public Student save(Student entity) throws DuplicateEntityException, InvalidDataException {
        logger.info("Saving student: {}", entity);
        if (entity == null || entity.getUser() == null || entity.getDni() == null || entity.getBirthdate() == null) {
            logger.error("Student or DNI cannot be null");
            throw new InvalidDataException("Student or DNI cannot be null");
        }
        logger.info("Student saved: {}", entity);
        userRepository.save(entity.getUser());
        return super.save(entity);
    }
}
```

El uso de logs mejora el mantenimiento del sistema y facilita la localización de errores durante las pruebas o en producción.

Carga de datos TestDataSeeder:

Durante el desarrollo, es útil contar con datos de prueba automáticos para verificar funcionalidades. Para ello, se ha implementado la clase TestDataSeeder, que se ejecuta al iniciar la aplicación en el perfil de desarrollo (dev).

Su función es crear registros iniciales de:

- Usuarios con diferentes roles (admin, profesor, estudiante).
- Roles y permisos asociados.
- Grupos de entrenamiento y sesiones.
- Asistencias y membresías.
- Facturas, productos y pagos de ejemplo.

Ejemplo:

```
@Component no usages & Dennis Eckerskom
@Profile("dev")
public class TestDataSeeder implements CommandLineRunner {

    @Override & Dennis Eckerskom
    public void run(String... args) throws Exception {
        // Roles
        Role studentRole = new Role();
        studentRole.setName("ROLE_STUDENT");
        this.roleService.save(studentRole);

        Role teacherRole = new Role();
        teacherRole.setName("ROLE_TEACHER");
        this.roleService.save(teacherRole);

        Role adminRole = new Role();
        adminRole.setName("ROLE_ADMIN");
        this.roleService.save(adminRole);

        // Permisos
        for (PermissionValues value : PermissionValues.values()) {
            Permission p = new Permission();
            p.setPermissionName(value);
            this.permissionService.save(p);
        }

        // Obtener permisos de la base de datos
        Permission fullAccess = this.permissionService.findPermissionByName(PermissionValues.FULL_ACCESS);
        Permission manageUsers = this.permissionService.findPermissionByName(PermissionValues.MANAGE_STUDENTS);
        Permission viewOwnData = this.permissionService.findPermissionByName(PermissionValues.VIEW_OWN_DATA);

        // Asignar permisos a roles
        studentRole.addPermission(viewOwnData);
        teacherRole.addPermission(manageUsers);
        teacherRole.addPermission(viewOwnData);
        adminRole.addPermission(fullAccess);

        // Guardar roles actualizados
        this.roleService.update(studentRole);
        this.roleService.update(teacherRole);
        this.roleService.update(adminRole);
    }
}
```

Perfil de desarrollo (dev):

Para controlar el comportamiento del sistema en distintos entornos (desarrollo, pruebas o producción), se ha definido un perfil específico denominado dev. Este perfil permite trabajar con una base de datos temporal y precargada con datos de prueba mediante la clase `TestDataSeeder`.

La activación del perfil se realiza a través del archivo `application.properties`, añadiendo las siguientes propiedades:

- `spring.profiles.active=dev`: Activa el perfil de desarrollo.
- `spring.jpa.hibernate.ddl-auto=create`: Indica que Hibernate debe generar automáticamente el esquema de la base de datos al iniciar la aplicación.

Con esta configuración, al iniciar el sistema:

- Se genera el esquema completo de la base de datos a partir de las entidades del proyecto.
- Se ejecuta automáticamente el `TestDataSeeder`, que inserta datos de prueba como usuarios, roles, estudiantes, profesores, membresías, facturas, etc.

Una vez la base de datos ha sido correctamente inicializada con los datos deseados, se recomienda cambiar la propiedad `ddl-auto` a `validate`, para evitar sobrescribir datos accidentalmente:

- `spring.jpa.hibernate.ddl-auto=validate`

Esto permite seguir utilizando los datos precargados, pero asegurando que el esquema generado por Hibernate coincida exactamente con el existente en la base de datos.

Importante: Tras finalizar la fase de desarrollo o prueba, se recomienda desactivar o comentar la propiedad `spring.profiles.active=dev` para evitar que el sistema vuelva a recrear o modificar la base de datos de forma no deseada en entornos productivos.

Pruebas unitarias:

El módulo incluye pruebas unitarias con JUnit 5 y Mockito para garantizar la validez de los servicios y las relaciones entre entidades. Las pruebas comprueban el correcto funcionamiento de operaciones como:

- Creación y recuperación de entidades.
- Validación de relaciones (User - Student, Invoice - InvoiceLine, etc.).
- Regla de negocio personalizada.

Ejemplo de prueba para StudentService:

```
@BeforeEach @Dennis Eckerskom
void setUp() throws Exception {
    MockitoAnnotations.openMocks( testClass: this);

    student = new Student();
    student.setId(1);
    student.setDni("12345678A");
    student.setBirthdate(LocalDate.of( year: 2000, month: 1, dayOfMonth: 1));
    student.setUser(new User());
    student.setHistories(new HashSet<>());
    student.setAssistances(new HashSet<>());
    student.setTrainingGroups(new HashSet<>());

    membership = new Membership(); membership.setId(1);
    history = new StudentHistory(); history.setId(1);
    assistance = new Assistance(); assistance.setId(10);
    group = new TrainingGroup(); group.setId(99); group.setStudents(new HashSet<>());

    // Inyectar entityManager usando reflexión
    Field emField = StudentService.class.getSuperclass().getDeclaredField( name: "entityManager");
    emField.setAccessible(true);
    emField.set(studentService, entityManager);
}

@Test @Dennis Eckerskom
void save_ValidStudent_ShouldSucceed() {
    when(userRepository.save(any())).thenReturn(student.getUser());
    when(studentRepository.save(any())).thenReturn(student);

    Student saved = studentService.save(student);
    assertEquals( expected: "12345678A", saved.getDni());
}
```

Cada servicio del sistema tiene su propia clase de prueba, lo que permite asegurar que el sistema responde correctamente ante distintos escenarios.

Módulo MemberFlow-API:

El módulo MemberFlow-API representa la capa intermedia entre el frontend del sistema y la lógica de negocio implementada en memberflow-data. Su objetivo principal es exponer dicha funcionalidad a través de una API REST, segura, estructurada y bien documentada.

Este módulo actúa como puente de comunicación, permitiendo que los clientes (principalmente la interfaz web) interactúen con los servicios del sistema mediante peticiones HTTP. Además, incorpora mecanismos de autenticación, control de acceso, validación de datos, documentación automática y gestión global de excepciones.

Estructura del proyecto:

El proyecto memberflow-api está organizado en paquetes según su responsabilidad funcional, lo que facilita su mantenibilidad y escalabilidad. Los principales paquetes son:

- **controller:** Contiene todos los controladores REST para entidades como User, Student, Teacher, Admin, Membership, Invoice, etc. Cada controlador mapea peticiones HTTP (GET, POST, PUT, DELETE) a las operaciones del servicio correspondiente.
- **dto:** Agrupa las clases DTO (Data Transfer Objects), que encapsulan la información intercambiada entre el cliente y la API, ocultando detalles internos de las entidades del modelo.
- **mapper:** Cada DTO incluye su propio mapper, encargadas de transformar entidades a DTOs y viceversa.
- **security:** Gestiona la configuración de seguridad del sistema, incluyendo autenticación, autorización, filtros JWT y definición de roles.
- **exception:** Centraliza el manejo de errores mediante excepciones personalizadas (EntityNotFoundException, ValidationException, etc.) y un GlobalExceptionHandler.
- **config:** Incluye la configuración global del sistema, como Swagger/OpenAPI, CORS, o propiedades personalizadas.

Seguridad y Autenticación

La seguridad del sistema está basada en Spring Security con JWT (JSON Web Tokens). Este mecanismo permite autenticar usuarios, generar un token firmado, y validar cada petición protegida según los roles y permisos asignados.

Flujo de autenticación:

- El usuario realiza login mediante un endpoint `/auth/login` enviando sus credenciales.
- Si las credenciales son válidas, se genera un token JWT que se devuelve al cliente.
- El cliente almacena el token (por ejemplo, en `localStorage`) y lo incluye en cada petición mediante el encabezado `Authorization: Bearer <token>`.
- Spring Security, mediante filtros personalizados, intercepta cada petición, valida el token, y permite o bloquea el acceso según el rol del usuario.

Configuración principal:

La clase `SecurityConfig` se encarga de:

- Definir las reglas de acceso según el rol (`ADMIN`, `TEACHER`, `STUDENT`).
- Permitir acceso libre a ciertos endpoints (`/auth/**`, `/swagger-ui/**`, etc.).
- Registrar el filtro `JwtAuthenticationFilter`.
- Configurar `CORS` y el `AuthenticationProvider`.

Roles y permisos

Cada usuario en el sistema está vinculado a un rol (`Role`) y este, a su vez, puede tener una lista de permisos (`Permission`). Esta jerarquía permite definir tanto acceso general por rol como restricciones específicas.

Por ejemplo:

- `ADMIN_ROLE`: Accede a todos los endpoints.
- `TEACHER_ROLE`: Gestiona estudiantes y clases.
- `STUDENT_ROLE`: Solo puede acceder a sus propios datos.

Validación y Manejo de Errores

Antes de procesar cualquier petición, los DTOs se validan automáticamente gracias a las anotaciones de `javax.validation` (`@NotNull`, `@Email`, `@Size`, etc.). Si la validación falla, el `GlobalExceptionHandler` captura la excepción y devuelve una respuesta clara con el código de error y el mensaje correspondiente.

Además, se han definido excepciones personalizadas para manejar casos específicos, como cuando no se encuentra una entidad, o cuando se intenta realizar una operación no permitida.

Características:

- Usa `@ControllerAdvice` y `@ExceptionHandler`.
- Captura excepciones comunes (`EntityNotFoundException`, `MethodArgumentNotValidException`, etc.).
- Devuelve respuestas con estructura uniforme (mensaje, código, timestamp).

Fragmento de la clase `GlobalExceptionHandler`:

```
@ControllerAdvice 1 usage & Dennis Eckerskom
public class GlobalExceptionHandler {

    private static final Logger logger = LoggerFactory.getLogger(GlobalExceptionHandler.class); 1 usage

    // * Custom application exceptions

    @ExceptionHandler(EntityNotFoundException.class) no usages & Dennis Eckerskom
    public ResponseEntity<Object> handleEntityNotFound(EntityNotFoundException ex, WebRequest request) {
        return buildErrorResponse( message: "Entity not found: " + ex.getMessage(), HttpStatus.NOT_FOUND);
    }

    @ExceptionHandler(DuplicateEntityException.class) no usages & Dennis Eckerskom
    public ResponseEntity<Object> handleDuplicateEntity(DuplicateEntityException ex, WebRequest request) {
        return buildErrorResponse( message: "Duplicate entity: " + ex.getMessage(), HttpStatus.CONFLICT);
    }

    @ExceptionHandler(BadRequestException.class) no usages & Dennis Eckerskom
    public ResponseEntity<Object> handleBadRequest(BadRequestException ex, WebRequest request) {
        return buildErrorResponse( message: "Bad request: " + ex.getMessage(), HttpStatus.BAD_REQUEST);
    }

    @ExceptionHandler(InvalidDataException.class) no usages & Dennis Eckerskom
    public ResponseEntity<Object> handleInvalidData(InvalidDataException ex, WebRequest request) {
        return buildErrorResponse( message: "Invalid data: " + ex.getMessage(), HttpStatus.UNPROCESSABLE_ENTITY);
    }

    @ExceptionHandler(AuthenticationException.class) no usages & Dennis Eckerskom
    public ResponseEntity<Object> handleAuthentication(AuthenticationException ex, WebRequest request) {
        return buildErrorResponse( message: "Unauthorized: " + ex.getMessage(), HttpStatus.UNAUTHORIZED);
    }
}
```

Esto mejora la experiencia del desarrollador y facilita el uso correcto de la API desde el frontend.

Documentación automática (Swagger)

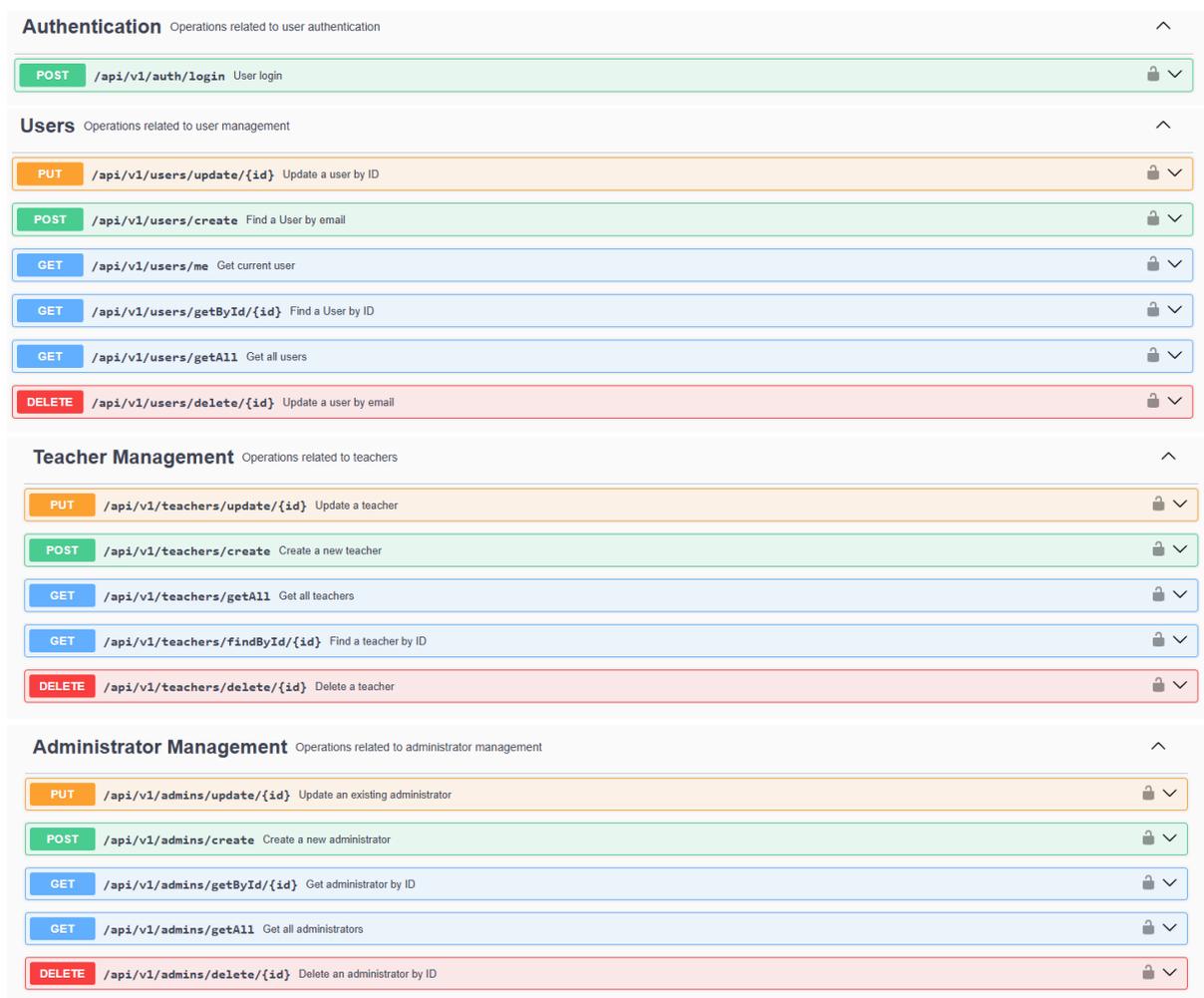
Para facilitar el desarrollo y las pruebas de la API, se ha integrado Swagger/OpenAPI mediante la librería springdoc-openapi. Esto genera automáticamente una documentación interactiva en la ruta:

- `/swagger-ui/index.html`

Desde esta interfaz, cualquier usuario autenticado puede:

- Consultar todos los endpoints disponibles.
- Ver ejemplos de peticiones/respuestas.
- Probar la API directamente desde el navegador.

Algunas capturas de la API documentada con Swagger:



The screenshot displays the Swagger UI interface, organized into four main sections, each with a list of API endpoints:

- Authentication** (Operations related to user authentication):
 - POST `/api/v1/auth/login` User login
- Users** (Operations related to user management):
 - PUT `/api/v1/users/update/{id}` Update a user by ID
 - POST `/api/v1/users/create` Find a User by email
 - GET `/api/v1/users/me` Get current user
 - GET `/api/v1/users/getById/{id}` Find a User by ID
 - GET `/api/v1/users/getAll` Get all users
 - DELETE `/api/v1/users/delete/{id}` Update a user by email
- Teacher Management** (Operations related to teachers):
 - PUT `/api/v1/teachers/update/{id}` Update a teacher
 - POST `/api/v1/teachers/create` Create a new teacher
 - GET `/api/v1/teachers/getAll` Get all teachers
 - GET `/api/v1/teachers/findById/{id}` Find a teacher by ID
 - DELETE `/api/v1/teachers/delete/{id}` Delete a teacher
- Administrator Management** (Operations related to administrator management):
 - PUT `/api/v1/admins/update/{id}` Update an existing administrator
 - POST `/api/v1/admins/create` Create a new administrator
 - GET `/api/v1/admins/getById/{id}` Get administrator by ID
 - GET `/api/v1/admins/getAll` Get all administrators
 - DELETE `/api/v1/admins/delete/{id}` Delete an administrator by ID

Student Management Operations related to student management

PUT	<code>/api/v1/students/updateMembership/{studentId}</code> Update a student's membership	🔒	⌵
PUT	<code>/api/v1/students/update/{id}</code> Update an existing Student	🔒	⌵
POST	<code>/api/v1/students/register</code> Register a new Student	🔒	⌵
POST	<code>/api/v1/students/create</code> Create a new Student	🔒	⌵
POST	<code>/api/v1/students/addStudentHistory/{id}</code> Add a Student History	🔒	⌵
POST	<code>/api/v1/students/addMembershipToStudent/{id}</code> Add a Membership to a Student	🔒	⌵
POST	<code>/api/v1/students/addGroupToStudent/{id}</code> Add a Training Group to a Student	🔒	⌵
POST	<code>/api/v1/students/addAssistanceToStudent/{id}</code> Add an Assistance to a Student	🔒	⌵
GET	<code>/api/v1/students/getAll</code> Get all Students	🔒	⌵
GET	<code>/api/v1/students/findById/{id}</code> Find a Student by ID	🔒	⌵
DELETE	<code>/api/v1/students/removeStudentHistory/{id}</code> Remove a Student History	🔒	⌵
DELETE	<code>/api/v1/students/removeMembershipFromStudent/{id}</code> Remove a Membership from a Student	🔒	⌵
DELETE	<code>/api/v1/students/deleteAssistanceFromStudent/{id}</code> Remove an Assistance from a Student	🔒	⌵
DELETE	<code>/api/v1/students/delete/{id}</code> Delete a Student	🔒	⌵

Role Management Operations related to roles

PUT	<code>/api/v1/roles/update/{id}</code> Update a role	🔒	⌵
POST	<code>/api/v1/roles/permissions/add/{roleId}/{permissionId}</code> Add permission to role	🔒	⌵
POST	<code>/api/v1/roles/create</code> Create a new role	🔒	⌵
GET	<code>/api/v1/roles/getById/{id}</code> Get role by ID	🔒	⌵
GET	<code>/api/v1/roles/getAll</code> Get all roles	🔒	⌵
DELETE	<code>/api/v1/roles/permissions/remove/{permissionId}/{roleId}</code> Remove permission from role	🔒	⌵
DELETE	<code>/api/v1/roles/delete/{id}</code> Delete a role	🔒	⌵

Permission Management Operations related to permissions

PUT	<code>/api/v1/permissions/update/{id}</code> Update an existing permission	🔒	⌵
POST	<code>/api/v1/permissions/create</code> Create a new permission	🔒	⌵
GET	<code>/api/v1/permissions/getById/{id}</code> Get permission by ID	🔒	⌵
GET	<code>/api/v1/permissions/getAll</code> Get all permissions	🔒	⌵
DELETE	<code>/api/v1/permissions/delete/{id}</code> Delete a permission	🔒	⌵

Training Group Management Operations related to training group management

PUT	/api/v1/training-groups/update/{id}	Update an existing training group	🔒
PUT	/api/v1/training-groups/remove-student	Remove a student from a group	🔒
PUT	/api/v1/training-groups/assign-student	Assign a student to a group	🔒
POST	/api/v1/training-groups/create	Create a training group with a teacher	🔒
GET	/api/v1/training-groups/getAll	Get all training groups	🔒
GET	/api/v1/training-groups/findById/{id}	Find a training group by ID	🔒
DELETE	/api/v1/training-groups/delete/{id}	Delete a training group by ID	🔒

Notification Management Operations related to notifications

PUT	/api/v1/notifications/update/{id}	Update a notification by ID	🔒
POST	/api/v1/notifications/create	Create a new notification	🔒
GET	/api/v1/notifications/getById/{id}	Get a notification by ID	🔒
GET	/api/v1/notifications/getAll	Get all notifications	🔒
DELETE	/api/v1/notifications/delete/{id}	Delete a notification by ID	🔒

Student History Management Operations related to student history

PUT	/api/v1/student-history/update/{id}	Update an existing student history	🔒
POST	/api/v1/student-history/create	Create a new student history	🔒
GET	/api/v1/student-history/getAll	Get all student histories	🔒
GET	/api/v1/student-history/findById/{id}	find student history by ID	🔒
DELETE	/api/v1/student-history/delete/{id}	Delete a student history	🔒

Memberships Operations related to membership management

PUT	/api/v1/memberships/update/{id}	Update a membership	🔒
POST	/api/v1/memberships/create	Create a new membership	🔒
GET	/api/v1/memberships/getById/{id}	Get a membership by ID	🔒
GET	/api/v1/memberships/getAll	Get all memberships	🔒
DELETE	/api/v1/memberships/delete/{id}	Delete a membership	🔒

Assistance Management Operations related to assistance management

PUT	/api/v1/assistances/update	Update an existing assistance record	🔒
POST	/api/v1/assistances/create	Create a new assistance record	🔒
GET	/api/v1/assistances/getById/{id}	Get assistance records by student ID	🔒
GET	/api/v1/assistances/getAll	Get all assistance records	🔒
DELETE	/api/v1/assistances/delete/{id}	Delete an assistance record by ID	🔒

Invoices Operations related to invoice management

PUT	/api/v1/invoices/update	Update an existing invoice	🔒
PUT	/api/v1/invoices/recalculateTotalOfInvoiceById/{invoiceId}	Recalculate the total of an invoice	🔒
POST	/api/v1/invoices/create	Create a new invoice	🔒
POST	/api/v1/invoices/createInvoiceWithLines	Create a new invoice with lines	🔒
POST	/api/v1/invoices/addLinesByInvoiceId/{invoiceId}	Add a line to an invoice by invoice ID	🔒
GET	/api/v1/invoices/getById/{id}	Get invoice by ID	🔒
GET	/api/v1/invoices/getAll	Get all invoices	🔒
GET	/api/v1/invoices/getAllInvoicesByUserId/{userId}	Get all invoices by user ID	🔒
GET	/api/v1/invoices/generatePDFById/{id}	Generar y descargar factura en PDF	🔒
DELETE	/api/v1/invoices/removeLineFromInvoiceById/{invoiceId}	Remove a line from an invoice by IDs	🔒
DELETE	/api/v1/invoices/deleteById/{id}	Delete invoice by ID	🔒
DELETE	/api/v1/invoices/clearAllLinesFromInvoiceById/{invoiceId}	Clear all invoice lines from an invoice	🔒

Invoice Lines Operations related to invoice lines

PUT	/api/v1/invoice-lines/update	Update an existing invoice line	🔒
POST	/api/v1/invoice-lines/create	Create a new invoice line	🔒
GET	/api/v1/invoice-lines/getById/{id}	Get invoice line by ID	🔒
GET	/api/v1/invoice-lines/getAll	Get all invoice lines	🔒
DELETE	/api/v1/invoice-lines/deleteById/{id}	Delete invoice line by ID	🔒

Payments Operations related to payment management

PUT	/api/v1/payments/update	Update an existing payment and adjust invoice status	🔒
POST	/api/v1/payments/create	Create a new payment and mark invoice as PAID	🔒
GET	/api/v1/payments/getById/{id}	Get payment by ID	🔒
GET	/api/v1/payments/getAll	Get all payments	🔒
GET	/api/v1/payments/getAllByUserId/{userId}	Get all payments by user ID	🔒
DELETE	/api/v1/payments/deleteById/{id}	Remove a payment and set invoice status to NOT_PAID	🔒

Product Services Operations related to product/service management

PUT	/api/v1/products-services/update	Update an existing product/service	🔒
POST	/api/v1/products-services/create	Create a new product/service	🔒
GET	/api/v1/products-services/getById/{id}	Get product/service by ID	🔒
GET	/api/v1/products-services/getAll	Get all products/services	🔒
DELETE	/api/v1/products-services/deleteById/{id}	Delete product/service by ID	🔒

IVA Types Operations related to IVA type management

PUT	/api/v1/iva-types/update	Update an existing IVA type	🔒
POST	/api/v1/iva-types/create	Create a new IVA type	🔒
GET	/api/v1/iva-types/getById/{id}	Get IVA type by ID	🔒
GET	/api/v1/iva-types/getAll	Get all IVA types	🔒
DELETE	/api/v1/iva-types/deleteById/{id}	Delete IVA type by ID (if not in use)	🔒

Controladores y Endpoints

Los controladores son responsables de exponer los servicios del sistema a través de endpoints REST, permitiendo al frontend (y potencialmente a otros clientes) interactuar con los datos de forma estructurada.

Estructura y patrón seguido

Cada entidad principal tiene su propio controlador ubicado en el paquete controller. El patrón seguido es:

- `@RestController` para marcar la clase como controlador REST.
- `@RequestMapping("/api/v1/entidad")` para definir el prefijo de ruta.
- Métodos con anotaciones como `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping`.
- Uso de DTOs como entrada y salida de datos.

Ejemplo StudentController:

```
@RestController no usages Dennis Eckerskorn
@RequestMapping("/api/v1/students")
@Tag(name = "Student Management", description = "Operations related to student management")
public class StudentController {

    private final StudentService studentService; 23 usages
    private final RoleService roleService; 3 usages
    private final UserService userService; 2 usages
    private final MembershipService membershipService; 3 usages
    private final TrainingSessionService trainingSessionService; 2 usages

    @Autowired 3 usages
    private PasswordEncoder passwordEncoder;
```

Este patrón se replica en todos los controladores (StudentController, InvoiceController, etc.), asegurando coherencia y facilidad de mantenimiento.

DTOs y Mapeo

Para evitar exponer directamente las entidades JPA (lo cual implicaría riesgos de seguridad, acoplamiento innecesario y problemas de serialización), se utilizan DTOs (Data Transfer Objects). Cada DTO contiene métodos fromEntity() y toEntity() que permiten convertir datos en ambos sentidos.

Funciones del DTO:

- Encapsulan solo los datos necesarios para una operación específica.
- Ocultan detalles internos de las entidades (como claves foráneas, relaciones complejas).
- Permiten aplicar validaciones (@NotNull, @Email, etc.).
- Ayudan a transformar la estructura de entrada/salida para adaptarse al frontend.

Ejemplo StudentDTO:

```
public class StudentDTO { 17 usages ⓘ Dennis Eckerskom

    private Integer id; 4 usages
    private UserDTO user; 5 usages
    private String dni; 4 usages
    private LocalDate birthdate; 4 usages
    private String belt; 4 usages
    private String progress; 4 usages
    private String medicalReport; 4 usages
    private String parentName; 4 usages
    private Integer membershipId; 3 usages

    private List<AssistanceDTO> assistances; 3 usages
    private List<TrainingGroupDTO> trainingGroups; 3 usages
    private List<TrainingSessionDTO> trainingSessions; 3 usages

    private MembershipDTO membership; 3 usages

    public StudentDTO() { no usages ⓘ Dennis Eckerskom
    }
```

fromEntity():

```
public static StudentDTO fromEntity(Student student) { @ Dennis Eckerskorn
    if (student == null || student.getUser() == null) {
        return null;
    }

    List<AssistanceDTO> assistanceDTOs = student.getAssistances().stream() Stream<Assistance>
        .map(AssistanceDTO::fromEntity) Stream<AssistanceDTO>
        .collect(Collectors.toList());

    List<TrainingGroupDTO> groupDTOs = student.getTrainingGroups().stream() Stream<TrainingGroup>
        .map(TrainingGroupDTO::fromEntity) Stream<TrainingGroupDTO>
        .collect(Collectors.toList());

    Set<TrainingSession> allSessions = student.getAssistances().stream() Stream<Assistance>
        .map(Assistance::getTrainingSession) Stream<TrainingSession>
        .collect(Collectors.toSet());

    List<TrainingSessionDTO> sessionDTOs = allSessions.stream() Stream<TrainingSession>
        .map(TrainingSessionDTO::fromEntity) Stream<TrainingSessionDTO>
        .collect(Collectors.toList());

    return new StudentDTO(
        student.getId(),
        UserDTO.fromEntity(student.getUser()),
        student.getDni(),
        student.getBirthdate(),
        student.getBelt(),
        student.getProgress(),
        student.getMedicalReport(),
        student.getParentName(),
        student.getMembership() != null ? student.getMembership().getId() : null,
        assistanceDTOs,
        groupDTOs,
        sessionDTOs
    ).withMembership(student.getMembership() != null ? MembershipDTO.fromEntity(student.getMembership()) : null);
}
```

toEntity():

```
public Student toEntity() { @ Dennis Eckerskorn
    Student student = new Student();
    student.setId(this.id);
    if (this.user != null) {
        student.setUser(this.user.toEntity());
    }
    student.setDni(this.dni);
    student.setBirthdate(this.birthdate);
    student.setBelt(this.belt);
    student.setProgress(this.progress);
    student.setMedicalReport(this.medicalReport);
    student.setParentName(this.parentName);
    return student;
}
```

Este enfoque desacopla completamente la capa de presentación (API) de la lógica de persistencia (memberflow-data).

Integración con memberflow-data

memberflow-api depende del módulo memberflow-data para acceder a las entidades y servicios que contienen la lógica de negocio y persistencia.

Inyección de dependencias

Gracias a Spring Boot y al uso de `@ComponentScan`, `@Service`, y `@Autowired` o `@RequiredArgsConstructor`, se inyectan directamente los servicios definidos en memberflow-data.

Ejemplo:

```
private final UserService userService; 8 usages
private final RoleService roleService; 2 usages
private final JwtUtil jwtUtil; 2 usages

@Autowired 2 usages
private PasswordEncoder passwordEncoder;

public UserController(UserService userService, RoleService roleService, JwtUtil jwtUtil) {
    this.userService = userService;
    this.roleService = roleService;
    this.jwtUtil = jwtUtil;
}
```

De esta forma, el módulo API actúa como una capa superior que no necesita conocer los detalles internos de implementación.

Generación de facturas en PDF

Una de las funcionalidades destacadas del sistema es la capacidad de generar facturas en formato PDF desde los datos almacenados en la base de datos.

Proceso:

- Se accede al endpoint `/api/v1/generatePDFById/{id}`
- Se recuperan los datos de la factura (cliente, líneas, totales, IVA...).
- Se genera un documento PDF con estructura profesional.
- El archivo PDF se devuelve como respuesta con tipo `application/pdf`.

Endpoint:

```
@GetMapping("/generatePDFById/{id}") no usages  ⚡ Dennis Eckerskorn
@Operation(summary = "Generate PDF for invoice by ID")
public ResponseEntity<byte[]> downloadInvoicePdf(@PathVariable Integer id) throws EntityNotFoundException {
    Invoice invoice = invoiceService.findById(id);

    byte[] pdf = invoicePdfGenerator.generateInvoicePdf(invoice);

    return ResponseEntity.ok()
        .header(HttpHeaders.CONTENT_DISPOSITION, ...headerValues: "attachment; filename=factura_" + id + ".pdf")
        .contentType(MediaType.APPLICATION_PDF)
        .body(pdf);
}
```

Implementación

Se ha utilizado una librería, iText, para construir el documento. El diseño incluye:

- Encabezado con datos del emisor.
- Detalle de productos o servicios.
- Totales con desglose de IVA.
- Fecha y firma.

Esto permite a la academia emitir facturas válidas y descargables para sus alumnos, profesionalizando la gestión financiera.

Módulo MemberFlow-Frontend

El módulo memberflow-frontend constituye la interfaz visual del sistema, desarrollada como una aplicación web dinámica mediante React.js. Su principal objetivo es permitir a los usuarios (administradores, profesores y estudiantes) interactuar con las funcionalidades del sistema de manera intuitiva, clara y adaptada a su rol.

Este módulo consume los endpoints expuestos por el backend (memberflow-api) a través de peticiones HTTP REST. Las respuestas se procesan y presentan mediante componentes React reutilizables, organizados según las distintas áreas del sistema.

Estructura del proyecto

El frontend está organizado en diferentes carpetas, cada una con una responsabilidad específica:

- api/
Contiene la configuración de Axios (axiosConfig.js), utilizada para establecer la comunicación con la API, incluyendo la gestión del token JWT para autenticación.
- components/
Estructura los componentes reutilizables del sistema:
- forms/: Formularios como LoginForm, UsercreateForm, NotificationForm, etc.
- layout/: Componentes visuales comunes como Sidebar, Topbar, ContentArea, que definen la estructura general de la aplicación.
- lists/: Componentes para visualizar datos en forma de lista (UserList, NotificationList, etc.).
- styles/: Archivos CSS organizados por componente.
- pages/
Define las páginas por rol (admin/, teacher/, student/) y páginas comunes como LoginPage, RedirectByRole, ProfilePage.
- utils/
Incluye funciones auxiliares como jwtHelper.js, que permite decodificar el token JWT y extraer información como el nombre del usuario y su rol.

Archivos raíz

- App.js: Define las rutas de la aplicación y el layout general.
- index.js: Punto de entrada de la app React.
- App.css, index.css: Estilos globales.

Tecnologies utilitzades

El desenvolupament del mòdul memberflow-frontend se ha basat en tecnologies modernes del ecosistema JavaScript, amb l'objectiu de construir una aplicació web de tipus SPA (Single Page Application) eficient, modular i mantenible. A continuació, se detallen les principals eines emprades:

React.js

Biblioteca principal per a la construcció de la interfície d'usuari. Permet crear components reutilitzables i gestionar l'estat de forma reactiva mitjançant hooks com useState i useEffect. Toda la estructura visual del sistema està organitzada a través de components.

Axios

Client HTTP utilitzat per a la comunicació amb el backend (memberflow-api). Se configura mitjançant una instància personalitzada en axiosConfig.js, que defineix una baseURL i un interceptor que afegeix automàticament el token JWT a cada petició autenticada.

React Router DOM

Libreria encarregada del enrutament dins de l'aplicació. Permet definir rutes per rol (/admin, /teacher, /student) i redirigir dinàmicament al usuari segons el seu perfil, mitjançant el component RedirectByRole.jsx.

JWT (JSON Web Tokens)

Sistema d'autenticació utilitzat per a controlar l'accés als recursos. Al iniciar sessió, el backend retorna un token JWT que es guarda a localStorage i es inclou automàticament en totes les peticions protegides. A més, el rol de l'usuari es extreu del token per personalitzar la navegació.

LocalStorage

Utilitzat per guardar de forma persistent el token JWT i el rol de l'usuari. Això permet mantenir la sessió iniciada després de recarregar la pàgina i restringir dinàmicament l'accés a funcionalitats específiques segons el tipus d'usuari.

CSS modularitzat

El sistema d'estils es gestiona mitjançant fitxers .css individuals associats a cada component. Això permet mantenir una estructura ordenada i facilita el manteniment visual.

Control de acceso y autenticación

El sistema de autenticación de MemberFlow está basado en tokens JWT y se implementa de forma segura desde el frontend con React.

Al iniciar sesión, se siguen los siguientes pasos:

- **Solicitud de autenticación:**
El componente LoginForm.jsx gestiona el formulario de acceso. Al enviarlo, realiza una petición POST al endpoint /auth/login, enviando el correo electrónico y la contraseña del usuario.
- **Recepción y almacenamiento del token:**
Si las credenciales son válidas, el servidor devuelve un token JWT que se guarda en el localStorage del navegador. Este token se utilizará para autenticar futuras peticiones.
- **Obtención del rol del usuario:**
Una vez autenticado, se realiza una segunda petición al endpoint /users/me, que devuelve la información del usuario, incluyendo su roleName. Este rol también se guarda en localStorage y es utilizado para redirigir al usuario a su panel correspondiente.
- **Configuración global de Axios:**
En el archivo axiosConfig.js, se configura una instancia de Axios que inyecta automáticamente el token JWT en cada petición HTTP mediante un interceptor. Esto asegura que todas las peticiones a la API estén autenticadas si el usuario ha iniciado sesión.
- **Manejo de errores:**
Si las credenciales son incorrectas, se muestra un mensaje de error personalizado en el formulario (Correo electrónico o Contraseña son incorrectos).

Este flujo garantiza que solo los usuarios autenticados accedan a las funcionalidades protegidas, y que cada uno sea redirigido según su rol (ADMIN, TEACHER, STUDENT).

Fragmento de api/axiosConfig.js:

```
import axios from 'axios'

const api = axios.create({
  baseURL: 'http://localhost:8080/api/v1',
  headers: {
    'Content-Type': 'application/json',
  },
});

api.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('token');
    if (token) {
      config.headers['Authorization'] = `Bearer ${token}`;
    }
    return config;
  },
  (error) => Promise.reject(error)
);

export default api;
```

Fragmento de LoginForm:

```
import React, { useState } from "react";
import api from "../../api/axiosConfig";
import "../styles/Login.css";

const LoginForm = ({ onLoginSuccess }) => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleLogin = async (e) => {
    e.preventDefault();
    setError('');
    try {
      const res = await api.post('/auth/login', { email, password });
      const token = res.data.token;
      localStorage.setItem('token', token);
      console.log("Token guardado:", token);

      const profileRes = await api.get('/users/me');
      const roleName = profileRes.data.roleName;
      console.log("Rol del usuario:", roleName);
      localStorage.setItem('roleName', roleName);

      onLoginSuccess();
    } catch (err) {
      console.error("Error al iniciar sesión:", err);
      setError('Correo electrónico o Contraseña son incorrectos');
    }
  };
};
```

Funcionalidades desarrolladas

El módulo memberflow-frontend proporciona una interfaz moderna, modular y adaptada al rol del usuario. Actualmente, el panel de administrador incluye acceso completo a todas las áreas funcionales del sistema: gestión de usuarios, estudiantes, profesores, clases, asistencias, historial del alumno, membresías, productos, facturación, pagos, tipos de IVA y notificaciones.

A continuación, se detallan las funcionalidades ya implementadas:

Autenticación y control de acceso

- Formulario de login con validación (LoginForm.jsx).
- Envío de credenciales al endpoint /auth/login y almacenamiento del token JWT en localStorage.
- Obtención del rol del usuario mediante /users/me y redirección automática con RedirectByRole.jsx.
- Visualización dinámica de opciones en el menú lateral según el rol (SidebarAdmin.jsx, SidebarTeacher.jsx, SidebarStudent.jsx).
- Protección de rutas sensibles y control de acceso a vistas.

Gestión de usuarios

- Creación de usuarios con distintos roles (ADMIN, TEACHER, STUDENT).
- Edición y eliminación de usuarios registrados.
- Asociación automática entre User y sus entidades compuestas (Admin, Teacher, Student).
- Visualización en lista de todos los usuarios.

Componentes destacados:

UserCreateForm.jsx, UserForm.jsx, UserList.jsx

Gestión de estudiantes

- Registro de nuevos estudiantes y edición de los existentes.
- Eliminación de estudiantes con confirmación segura.
- Asociación a grupos de entrenamiento desde formularios visuales.
- Acceso al historial del estudiante.
- Gestión de asistencias, membresías y relación con facturación.

Componentes destacados:

StudentForm.jsx, TrainingGroupsStudentManager.jsx, StudentHistoryList.jsx, AssistanceList.jsx, MembershipList.jsx

Gestión de profesores y administradores

- Registro de profesores (Teacher) y administradores (Admin) mediante formularios personalizados.
- Listado de profesores y administradores con opciones de edición.
- Relación clara con la entidad User.

Componentes destacados:

TeacherForm.jsx, AdminForm.jsx, TeacherList.jsx, AdminList.jsx

Gestión del historial del estudiante

- Creación de entradas en el historial académico del alumno.
- Registro de logros, progresos, participaciones destacadas, etc.
- Listado completo de eventos del historial.

Componentes destacados:

StudentHistoryCreateForm.jsx, StudentHistoryList.jsx

Gestión de notificaciones

- Envío de notificaciones a usuarios individuales desde un formulario simple.
- Listado de notificaciones enviadas con detalles por usuario.

Componentes destacados:

NotificationCreateForm.jsx, NotificationList.jsx

Gestión de clases y entrenamientos

- Creación y edición de grupos de entrenamiento (TrainingGroup) con nombre, horario y profesor asignado.
- Listado de grupos con opciones de edición y eliminación.
- Asociación de estudiantes a grupos mediante un formulario interactivo.
- Generación automática de sesiones de entrenamiento al crear un grupo.
- Visualización de un horario semanal general (ViewTimetable.jsx).

Componentes destacados:

*TrainingGroupFrom.jsx, TrainingGroupList.jsx,
TrainingGroupsStudentManager.jsx, TrainingSessionList.jsx, ViewTimetable.jsx*

Gestión de asistencias

- Registro manual de asistencias por sesión y estudiante.
- Visualización completa de la asistencia por grupo o alumno.

Componentes destacados:

AssistanceForm.jsx, AssistanceList.jsx

Gestión de membresías

- Creación y edición de tipos de membresía.
- Listado completo de membresías asignadas a estudiantes.
- Cambio de membresía activo desde el listado mediante menú desplegable.
- Validación de fechas y estado de la membresía desde el backend.

Componentes destacados:

MembershipCreateForm.jsx, MembershipList.jsx, MembershipSelector.jsx

Gestión de productos y tipos de IVA

- Alta y edición de productos facturables (ProductService).
- Asociación de cada producto con su tipo de IVA correspondiente.
- Visualización del listado de productos con filtros y acciones.
- Gestión completa de los tipos de IVA desde una interfaz visual.

Componentes destacados:

ProductForm.jsx, ProductList.jsx, IVATypeManager.jsx

Gestión de facturación

- Creación de facturas (Invoice) con múltiples líneas de producto.
- Cálculo automático de subtotales, impuestos y total general.
- Asociación directa de la factura a un estudiante existente.
- Listado completo de facturas con su estado (pagada o pendiente).
- Visualización detallada de los pagos asociados.

Componentes destacados:

InvoiceForm.jsx, InvoiceList.jsx, InvoiceLineItem.jsx, InvoiceLineTable.jsx

Gestión de pagos

- Registro de pagos asociados a una factura concreta.
- Selección del método de pago mediante enumerado (efectivo, tarjeta, transferencia...).
- Recalculo automático del estado de la factura al registrar el pago.

Componentes destacados:

PaymentForm.jsx, PaymentList.jsx

Generación de facturas en PDF

- Una vez creada la factura, se ofrece la opción de generar un PDF.
- Se realiza una petición al endpoint:
`/invoices/generatePDFById/{createdInvoice.id}`.
- El archivo generado se abre automáticamente en una nueva pestaña con `window.open`.

Esta implementación en el frontend permite al administrador operar de forma centralizada sobre todo el sistema. Además, la arquitectura basada en componentes modulares facilita la integración de futuras mejoras y la habilitación progresiva de funcionalidades para los roles de profesor y estudiante.

Herramientas utilizadas

Durante el desarrollo del sistema MemberFlow se han utilizado las siguientes herramientas y entornos de apoyo:

- **GitHub:** Gestión del control de versiones, documentación de avances, y seguimiento del proyecto.
- **Visual Studio Code / IntelliJ IDEA:** Entornos de desarrollo para el frontend y backend, respectivamente.
- **Postman:** Utilizado para realizar pruebas de los endpoints REST expuestos por la API.
- Navegador con herramientas de desarrollo (DevTools): Para validar el comportamiento visual y funcional del frontend.
- **Docker:** Herramienta de contenerización utilizada para desplegar el sistema de forma unificada, facilitando su ejecución en cualquier entorno.

Contenerización con Docker

Con el objetivo de facilitar la distribución, despliegue y pruebas del sistema en diferentes entornos, se ha realizado la Contenerización completa de MemberFlow utilizando Docker.

Estructura de archivos relevantes

En el directorio memberflow-docker/docker se encuentran los siguientes elementos:

- **docker-compose.yml:** Define los servicios del sistema (backend, frontend, base de datos MySQL, y servidor Nginx).
- **Dockerfile-api:** Imagen personalizada para el backend (memberflow-api).
- **Dockerfile-front:** Imagen personalizada para el frontend (memberflow-frontend).
- **nginx.conf:** Archivo de configuración del servidor web Nginx, utilizado como proxy para servir el frontend y enrutar correctamente.
- **init.sql / backup.sql:** Scripts SQL para inicializar la base de datos y cargar datos de prueba si es necesario.
- **mysql-data/:** Carpeta persistente para almacenar los datos del contenedor de MySQL.

Servicios definidos

- **backend:** Se construye a partir del Dockerfile-api, expone la API en el puerto 8080.
- **frontend:** Compila la aplicación React y la sirve mediante Nginx en el puerto 80.
- **mysql:** Servicio de base de datos con volumen persistente y configuración automática desde init.sql.
- **nginx:** Servidor intermedio que sirve los archivos estáticos del frontend y enruta correctamente las peticiones.

Presupuesto estimado

A continuación, se presenta una estimación aproximada del coste de desarrollo del sistema MemberFlow, considerando el uso de equipamiento, herramientas, licencias y tiempo de trabajo, con valores ficticios orientados a simular un entorno realista de desarrollo profesional.

Equipamiento e infraestructura:

Concepto	Detalles	Coste estimado (€)
Ordenador de sobremesa	Procesador i7-10700K, 32GB RAM, SSD 1TB, Windows 11 Pro	1200€
Monitor externo	Monitor Gigabyte 34WQC	500€
Conexión a Internet	Tarifa mensual durante el desarrollo	25€ x 4 = 100€
Ratón y Teclado	Periféricos gama media	75€
Total: 1875€		

Licencias y Herramientas

Durante el desarrollo se han utilizado versiones gratuitas o educativas, sin coste real. Sin embargo, se ha estimado el precio de las versiones profesionales como referencia:

Herramienta / Tecnología	Tipo de licencia / Uso	Coste estimado (€)
IntelliJ IDEA	Licencia educativa/anual (uso simulado)	100€
Docker Desktop	Gratuito para uso individual	0€
Postman	Gratuito	0€
GitHub Pro	Licencia mensual (simulada)	50€
VS Code	Gratuito	0€
OpenAI ChatGPT	Licencia Plus	24€ x 4 = 96€
Total: 246€		

Horas de desarrollo

Se estima que el desarrollo se ha realizado entre marzo y junio, con una media de 4 horas diarias, 6 días por semana durante 14 semanas. Esto supone un total aproximado de:

4h/día x 6 días/semana x 14 semanas = 336 horas

Tomando como referencia un valor estimado de 10€/hora para valorar el tiempo invertido en el proyecto:

Tarea	Horas estimadas	Importe (€)
Diseño de la base de datos y lógica	70h	700€
Desarrollo de API REST + seguridad	80h	800€
Desarrollo del frontend en React	100h	1000€
Docker y despliegue	30h	300€
Documentación y pruebas	56h	560€
Total, horas estimadas: 336h x 10€/hora = 3360€		

Coste total del proyecto:

Categoría	Importe (€)
Equipamiento e infraestructura	1875€
Licencias y herramientas	246€
Horas de desarrollo	3360€
Total: 5433€	

Todos los valores indicados son simulaciones orientativas para fines académicos y no representan facturación real.

Estimación de viabilidad y precio de comercialización

Dado el enfoque profesional y completo del sistema MemberFlow, resulta viable contemplar su comercialización, tanto en formato licencia cerrada como en modalidad SaaS (Software as a Service). A continuación, se detallan estimaciones orientativas de precios y un análisis de la rentabilidad mínima requerida para justificar su mantenimiento y soporte técnico.

Opción 1: Venta por licencia (modelo cerrado)

En este modelo, el sistema se vende a academias como una solución "on-premise", que pueden instalar en sus propios servidores o equipos locales. El precio se estima según funcionalidades y nivel de personalización.

Licencia	Precio estimado (€)
Versión básica (hasta 50 alumnos)	600€
Versión avanzada (hasta 200 alumnos)	1200€
Versión premium (sin límite + soporte)	1800€

Suponiendo un coste de desarrollo de **5443€** (presupuesto anterior), bastaría con vender:

- 3 licencias premium para recuperar la inversión inicial.
- A partir de la 4ª venta. El sistema sería rentable y comenzará a generar beneficios.

Opción 2: SaaS (Software como Servicio)

Otra opción viable es ofrecer MemberFlow como un servicio online, mediante suscripción mensual. Este modelo elimina la necesidad de instalación por parte del cliente y permite ingresos recurrentes.

Plan SaaS	Precio mensual (€)
Básico (hasta 50 usuarios)	25 € / mes
Pro (hasta 200 usuarios)	45€ / mes
Empresa (sin límite + soporte)	70€ / mes

Costes de mantenimiento estimados:

- Alojamiento en servidor (ej. VPS / Docker host): aproximadamente 20 €/mes
- Dominio + backups: aproximadamente 5 €/mes
- Tiempo de soporte mínimo mensual (10 h × 10 €/h): 100 €/mes

Total, mantenimiento mensual mínimo: = **125 €**

Punto de equilibrio:

Para cubrir estos costes sin pérdidas, se necesitarían, por ejemplo:

- 5 clientes en plan Pro (5 × 45 € = 225 €/mes)
- 2 clientes en plan Empresa (2 × 70 € = 140 €/mes)

Esto hace que el sistema pueda ser rentable a partir de 3-5 clientes activos mensuales, lo que lo convierte en una solución viable incluso en fase temprana.

Conclusiones

El desarrollo del sistema MemberFlow ha permitido implementar una solución integral para la gestión de academias de artes marciales. A través de un enfoque modular, escalable y progresivo, se han construido tres componentes principales:

- Un **backend robusto** (memberflow-data + memberflow-api), con seguridad basada en roles, lógica de negocio clara y una API REST completa.
- Un **frontend moderno** (memberflow-frontend), desarrollado en React, con una interfaz dinámica, adaptada al tipo de usuario y conectada en tiempo real con los servicios del backend.
- Una **arquitectura de despliegue contenerizada**, mediante Docker y Nginx, que permite ejecutar el sistema en cualquier entorno de forma sencilla y portátil.

El sistema permite gestionar de manera centralizada usuarios, clases, membresías, pagos, productos, facturación, asistencia y notificaciones, garantizando una experiencia unificada para administradores, profesores y estudiantes.

Además, la arquitectura basada en capas y tecnologías ampliamente adoptadas asegura su mantenibilidad a largo plazo, así como su capacidad de evolución frente a nuevas funcionalidades o necesidades del cliente.

Viabilidad y proyección comercial

El sistema MemberFlow puede ser comercializado tanto mediante licencia única (modelo cerrado) como en formato SaaS (Software como Servicio). Ambos modelos resultan económicamente viables, con una inversión inicial moderada y un punto de equilibrio alcanzable con pocos clientes.

En el modelo de licencia, bastarían 3 ventas premium para recuperar los costes de desarrollo.

En el modelo SaaS, la rentabilidad mensual se puede alcanzar con 3 a 5 academias activas, lo que lo convierte en una solución escalable incluso para un primer lanzamiento.

Estas proyecciones, junto con la solidez técnica del sistema, posicionan a MemberFlow como una plataforma con alto potencial de adopción comercial, especialmente en el sector deportivo y educativo especializado.

Futuras ampliaciones

A pesar del estado avanzado del sistema, existen diversas funcionalidades y mejoras previstas para fases posteriores, orientadas a perfeccionar la experiencia de uso, la gestión de negocio y el cumplimiento legal. Las líneas de trabajo contempladas incluyen:

- **Activación completa del panel para profesores y estudiantes**, con acceso personalizado a sus funcionalidades y datos según el rol.
- **Traducción multilinguaje del frontend** (español, inglés, alemán) para facilitar su adopción en academias internacionales o con alumnos de diferentes nacionalidades.
- **Integración con pasarelas de pago como Stripe o PayPal**, que permitirá automatizar el cobro de mensualidades, matrículas o productos directamente desde el sistema.
- **Mejora de la visualización de horarios y asistencias**, mediante calendarios interactivos, semanales o mensuales, adaptados tanto a móviles como escritorio.
- **Generación de informes y estadísticas personalizadas**, con indicadores clave como asistencias por grupo, ingresos mensuales, número de alumnos activos o histórico de pagos.
- **Automatización de copias de seguridad** desde el entorno Docker, incluyendo scripts programados para el volcado y exportación periódica de la base de datos.
- **Cumplimiento del Reglamento General de Protección de Datos (RGPD)**, mediante la implementación de:
 - Consentimiento explícito para el tratamiento de datos personales.
 - Política de privacidad visible y aceptada por los usuarios al registrarse.
 - Acceso y modificación de datos por parte del usuario según sus derechos.
 - Encriptación segura de contraseñas y almacenamiento limitado de información sensible.
- **Emisión de facturas conforme a la normativa fiscal vigente**, adaptando los documentos generados a los requisitos legales:
 - Inclusión de CIF/NIF, razón social, fecha de emisión y desglose de IVA.
 - Registro de facturas en orden correlativo.
 - Soporte para firma digital y exportación en formato PDF válido para contabilidad.

Bibliografia

El desarrollo del proyecto se ha apoyado principalmente en documentación técnica oficial y recursos asistidos mediante inteligencia artificial. Algunas fuentes relevantes incluyen:

- Spring Boot Documentation: <https://docs.spring.io/spring-boot/index.html>
- React Documentation: <https://react.dev/>
- Docker Docs: <https://docs.docker.com/>
- NGINX Configuration Guide:
https://nginx.org/en/docs/beginners_guide.html

ChatGPT (OpenAI): Asistencia en redacción técnica, refactorización de código, generación de ejemplos y solución de dudas durante el desarrollo.