Informe de Desarrollo Full-Stack para SolidarianID

Desarrollo Full-Stack

Máster Universitario en Ingeniería del Software

Autores:

Hernán Salambay Roldán
Alejandro Montoya Toro
Pedro Nicolás Gomariz
Aurora Hervás López
Dongyue Yu

15 de Enero de 2025

Facultad de Informática
UMU

Contenidos

1 Introducción

Este documento recoge la memoria del proyecto desarrollado, SolidarianID, una plataforma destinada a la gestión de comunidades solidarias, causas y acciones sociales. Durante este cuatrimestre, el enfoque principal ha sido el desarrollo del backend, utilizando Node.js, TypeScript y el framework NestJS. Se han aplicado los principios de la arquitectura limpia, Domain-Driven Design y una estructura de microservicios. En el frontend, se ha creado una aplicación web para el administrador de la plataforma, utilizando la arquitectura MVC.

SolidarianID constituye el centro del proyecto común en el que se centra el máster. Alrededor de él orbitan aspectos clave como la gestión de proyectos, el control de calidad, la arquitectura de datos y las prácticas de desarrollo CI/CD, que se mencionan brevemente en este documento, ya que cada una de ellas cuenta con su propia documentación específica. Además, el desarrollo colaborativo del proyecto ha estado centralizado mediante el uso de GitHub y las herramientas proporcionadas en otras asignaturas, lo que nos ha mejorado la experiencia de desarrollo.

A continuación, se detallan los requisitos técnicos y funcionales, las decisiones arquitectónicas tomadas, los mecanismos de seguridad implementados (autenticación y autorización), y el desarrollo de la API y las interfaces gráficas.

2 Arquitectura de la aplicación

La arquitectura de nuestra aplicación SolidarianID está basada en microservicios, lo que nos permite una alta escalabilidad y mantenibilidad, facilitando la integración y el despliegue independiente de cada componente. Para la comunicación entre los distintos componentes se han planteado dos enfoques: la interna entre microservicios utiliza el bus de eventos de NestJS, mientras que la interacción entre diferentes "bounded contexts" —áreas del dominio con lógica y modelos propios— se realiza mediante Kafka, garantizando una comunicación asíncrona, persistente y eficiente.

La plataforma se organiza en cuatro microservicios principales:

- **Pasarela de API:** gestiona las solicitudes externas y las dirige a los microservicios correspondientes.
- **Usuarios:** maneja la gestión de los perfiles de usuario, incluyendo autenticación, autorización, notificaciones e historial.
- **Comunidades:** administra la creación y gestión de comunidades, causas y acciones.
- **Estadísticas:** procesa datos para generar informes y visualizar métricas.

Cada microservicio cuenta con su propia base de datos, siguiendo el principio de independencia de datos, lo que nos asegura coherencia y modularidad. Además, el desarrollo de la plataforma ha seguido el enfoque "*monorepo*" ofrecido por NestJS, lo que facilita la gestión centralizada de los microservicios y librerías compartidas.

El frontend, desarrollado con NestJS bajo el patrón MVC, complementa la arquitectura con una interfaz dinámica. Donde utilizamos Handlebars para las vistas y Tailwind CSS para cumplir con el diseño responsivo. Su estructura planteada incluye:

- **Vistas:** páginas como "*home*" y "*login*", y secciones de administración.
- **Módulos:** divididos en "*reports*", "*statistics*" y "*validations*" para funcionalidades clave.

- **Controladores y servicios:** lógica de negocio y facilitación de la interacción con el backend.

En la siguiente figura se puede observar la estructura general de la arquitectura:

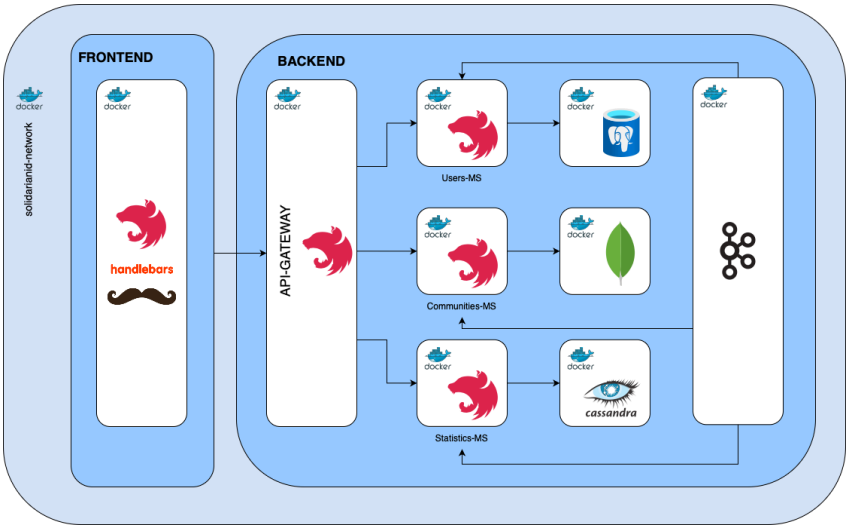

Figura 1: Arquitectura de la plataforma SolidarianID.

La conexión entre el frontend y el backend se realiza con Axios para una comunicación síncrona eficaz y sencilla. Mientras que, como se ha comentado anteriormente, el bus de eventos interno y Kafka sincronizan los microservicios de forma asíncrona, garantizando respuestas rápidas a los cambios en el dominio y la distribución eficiente de eventos.

Los apartados siguientes describen con mayor detalle los componentes y mecanismos empleados en el diseño de la arquitectura.

3 Backend

Este apartado profundiza en los aspectos esenciales del backend: el modelo de dominio, la organización del código, los principios arquitectónicos y los mecanismos de seguridad y comunicación empleados.

3.1 Modelo de dominio

El modelo de dominio de nuestra aplicación organiza la lógica de negocio en subdominios claramente definidos, cada uno asignado a un microservicio. Estos subdominios encapsulan su funcionalidad principal, delimitan su bounded context, y establecen la lógica que justifica su existencia dentro del sistema.

A continuación, se presenta el modelo de dominio propuesto para el MVP del proyecto:

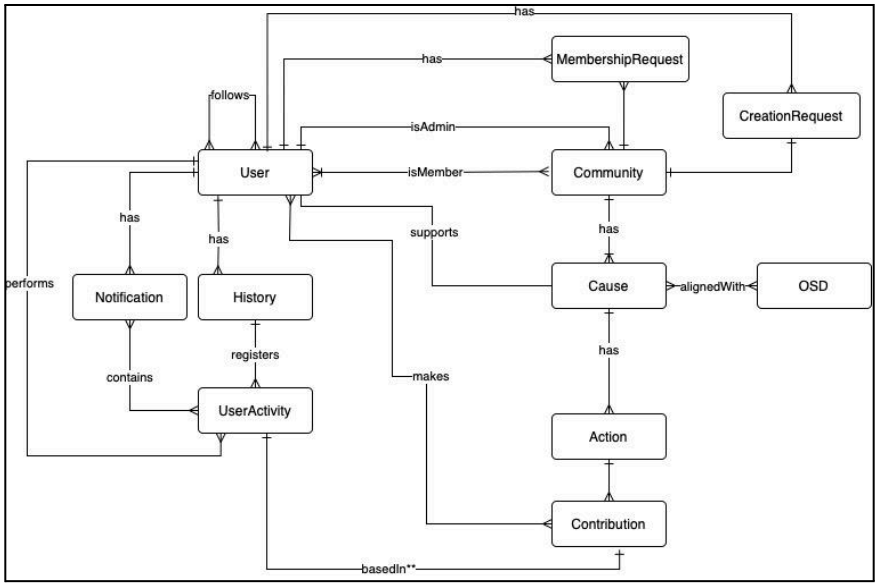

Figura 2: Entidades del dominio SolidarianID

En el diagrama presentado, se distinguen claramente dos bounded contexts principales:

- **Gestión de Usuarios** (lado izquierdo): abarca las entidades y relaciones relacionadas con los usuarios, su historial, notificaciones y solicitudes.

- **Gestión de Comunidades y Causas Solidarias** (lado derecho): incluye la gestión de comunidades, causas, acciones y contribuciones.

A continuación, detallamos los principales subdominios que sustentan los microservicios y su papel dentro de la plataforma

3.1.1 Usuarios

Este subdominio es responsable de la gestión de los perfiles, autenticación, autorización, notificaciones y registro de actividades de los usuarios. También incluye funcionalidades como solicitudes de membresía, seguimiento de otros usuarios y acceso al historial personal.

3.1.2 Comunidades

En este subdominio se gestiona la creación y administración de comunidades, así como las causas y acciones solidarias asociadas. Es el núcleo funcional de la plataforma, permitiendo a los usuarios colaborar y organizarse en torno a iniciativas solidarias. Además, se gestionan tanto las solicitudes de creación de nuevas comunidades como las solicitudes de unión a estas.

3.1.3 Estadísticas

Aunque no representa un modelo del dominio propiamente dicho, este “subdominio” encapsula la lógica necesaria para procesar y organizar los datos provenientes de las relaciones del dominio. Su existencia se justifica en la necesidad de generar informes y métricas relevantes que proporcionen información clave al administrador, facilitando la evaluación del impacto de las iniciativas solidarias y optimizando la toma de decisiones.

**Nota: Por razones de practicidad y claridad visual, no se han representado todas las relaciones de tipo "basedIn" asociadas a la entidad "UserActivity". Sin embargo, dichas relaciones las hemos considerado e implementado según las especificaciones establecidas.

3.2 Criterio de organización del código

La organización del código en el proyecto se basa principalmente en la modularización de conceptos bien estructurada y en una jerarquía clara de carpetas con nombres representativos que reflejan la funcionalidad de cada componente. Esto facilita la comprensión, el mantenimiento y la escalabilidad del sistema, alineándose con los principios de la arquitectura hexagonal y Domain-Driven Design (DDD).

Para implementar estos principios, se ha optado por el modelo *monorepo* debido a su integración nativa con NestJS, lo que permite centralizar todos los microservicios y librerías compartidas en un único repositorio. Este enfoque nos facilita la gestión del proyecto al tener todo el código organizado en un solo repositorio bajo Git, simplificando la integración, agilizando el control de versiones y mejorando la colaboración entre los miembros del equipo..

Cada microservicio está organizado en carpetas que reflejan las capas definidas por la arquitectura hexagonal:

- **Domain**: contiene las entidades, value objects y eventos de dominio, que encapsulan la lógica central del negocio.
- **Application**: gestiona el acceso a la lógica del dominio según los principios de DDD, e incluye servicios de aplicación específicos.
- **Infra**: agrupa los adaptadores necesarios para la persistencia de datos y la integración con servicios externos.

Asimismo, cada módulo incluye una réplica de esta estructura enfocada exclusivamente en el testing, con directorios separados para *src* y *test*, lo que permite asegurar la calidad del desarrollo mediante pruebas unitarias y de integración bien estructuradas.

En la siguiente figura se muestra un ejemplo simplificado de la estructura descrita:

Figura 3: Estructura simplificada del módulo Usuarios.

Por otro lado, este enfoque facilita la reutilización del código gracias a la posibilidad de definir una librería compartida, *common-lib,* que centraliza recursos reutilizables, como la autenticación, decoradores, utilidades generales y la definición del módulo de eventos basado en Kafka. Este estilo de organización promueve la consistencia entre los microservicios y nos facilita tanto el desarrollo como el mantenimiento del proyecto.

3.3 Aplicación de la arquitectura limpia/hexagonal

La arquitectura hexagonal aplicada en este proyecto garantiza un sistema robusto, flexible y escalable. Cada microservicio está organizado en módulos independientes que implementan las capas hexagonales, lo que nos permite desarrollar componentes de manera aislada, facilitando la incorporación de nuevas funcionalidades sin afectar la estructura existente.

Gracias a la separación entre la lógica de negocio y las dependencias externas, los adaptadores, como bases de datos, APIs o sistemas de mensajería, pueden ser cambiados o reemplazados sin impactar el dominio. Esto permite integrar nuevas tecnologías de forma sencilla, sin necesidad de refactorizar la lógica principal.

La testabilidad se mejora al permitir que los servicios de aplicación y el dominio se prueben de forma independiente, utilizando mocks para simular adaptadores externos. Esto asegura la calidad del código, garantizando que los cambios no introduzcan errores inesperados.

El desacoplamiento entre las capas asegura que los detalles tecnológicos, como la base de datos o la integración de nuevos servicios, no afecten las reglas del negocio. La separación entre los modelos de dominio y persistencia es, además de los servicios externos, un ejemplo de esta independencia, lo que facilita el mantenimiento y la evolución del sistema.

En resumen, como se ha mencionado en apartados previos, la arquitectura hexagonal ha facilitado la mantenibilidad y escalabilidad del sistema, permitiendo una evolución ágil y sin comprometer la calidad ni estabilidad.

3.4 Aplicación del Domain-Driven Design

En el desarrollo del backend, hemos aplicado los principios de Domain-Driven Design (DDD) para estructurar la lógica de negocio, asegurando que el código refleje claramente los conceptos y reglas del dominio de SolidarianID. Esta metodología nos ha permitido modelar el sistema de manera más robusta y alineada con las necesidades del negocio, facilitando la evolución y el mantenimiento del proyecto. A continuación se explica con más detalle los principios aplicados.

3.4.1 Lenguaje Ubicuo

Desde el inicio, hemos promovido el uso de un lenguaje ubicuo entre nuestro equipo y el experto en el dominio (Product Owner, profesor), utilizando una terminología compartida y coherente. Este lenguaje se refleja en el código a través de nombres significativos en clases, métodos y atributos. Por ejemplo, el concepto de "*Usuario*" se traduce directamente en la clase User, mientras que términos específicos como *UserEmail* y *UserBirthDate* representan objetos de valor que son de interés para los stakeholders. Esto nos ayuda a mejorar la comprensión mutua, además de reducir errores de interpretación al mantener la consistencia en toda la comunicación del equipo y el desarrollo.

3.4.2 Contextos delimitados (Bounded Contexts)

Para mantener una separación clara entre las distintas áreas del dominio, el sistema se ha dividido en contextos delimitados. Los principales contextos identificados son:

- **Gestión de Usuarios**: maneja la autenticación, perfiles, notificaciones y actividades de los usuarios.
- **Gestión de Comunidades y Causas**: administra la creación, validación y colaboración en iniciativas solidarias.

Como se mencionó con anterioridad, estos contextos están implementados en microservicios independientes, lo que garantiza que cada uno encapsule su propia lógica y modelos.

3.4.3 Entidades y Objetos de Valor

En el ejemplo del subdominio de Usuarios, las entidades como *User* y los objetos de valor como *UserEmail* y *Password* encapsulan atributos y la lógica clave del dominio, asegurando su consistencia. Del mismo modo, en el subdominio de Comunidades, entidades como Community y Cause representan conceptos centrales.

3.4.4 Agregados

Los agregados son clústeres de entidades y objetos de valor que garantizan la consistencia transaccional dentro de su ámbito. En el sistema, los agregados User y Community destacan como raíces principales: el primero incluye entidades como historial y notificaciones, preservando sus invariantes, mientras que el segundo agrupa elementos como Cause y solicitudes de unión, asegurando la coherencia en la lógica de negocio.

3.4.5 Eventos de Dominio

Para comunicar cambios significativos en el sistema, hemos implementado eventos de dominio como *UserJoinedCommunity* y *ActionCreatedEvent*. Estos eventos encapsulan información sobre lo ocurrido y son publicados mediante Kafka, permitiendo la sincronización entre microservicios. Por ejemplo, cuando un usuario se une a una comunidad, el evento correspondiente informa a otros servicios para que reaccionen adecuadamente, como actualizar estadísticas, hacer su registro en el historial del usuario o enviar notificaciones a sus seguidores. Este mecanismo mantiene el sistema desacoplado y facilita la extensión futura**.**

3.4.6 Repositorios

Cada microservicio utiliza repositorios que abstraen el acceso a la persistencia de datos. Por ejemplo, el *UserRepository* gestiona las operaciones CRUD de usuarios, asegurándonos un acceso desacoplado de la infraestructura, consistente y centralizado.

3.4.7 Beneficios del enfoque DDD

Seguir este enfoque nos ha ayudado a mejorar la claridad en el desarrollo y documentación del sistema al alinear la lógica del negocio con el código. Además, la separación de contextos y la adopción de principios como eventos de dominio y repositorios han facilitado el mantenimiento, escalabilidad, extensibilidad y desarrollo de la aplicación.

3.5 Control de errores

En este proyecto, el manejo de errores se gestiona utilizando un enfoque combinado que incluye tanto validaciones funcionales como el mecanismo de excepciones a través del uso de throw para lanzar excepciones específicas en situaciones de error.

3.5.1 Mecanismo de exception

En el proyecto, excepto en el módulo de *communities*, se utiliza un mecanismo de excepciones para manejar errores específicos de cada dominio. Cada módulo tiene su propio filtro de excepciones; por ejemplo, el módulo de *users* utiliza el *UserDomainExceptionFilter*, que captura y gestiona errores comunes en ese dominio, como *EmailAlreadyInUseError*, *EntityNotFoundError*, entre otros.

Cuando ocurre un error y se lanza una excepción, el filtro correspondiente identifica el tipo de error y asigna un código de estado HTTP adecuado. Este código se usa para generar una respuesta al cliente, que incluye el estado HTTP, un mensaje descriptivo sobre el error y una marca temporal.

Este enfoque permite que cada módulo maneje sus propios errores de manera centralizada y específica, garantizando respuestas claras y consistentes para el cliente. Además, facilita la depuración y el mantenimiento del sistema al contar con un manejo de excepciones bien estructurado y modular.

3.5.2 Validación funcional

Como se ha comentado, el módulo *communities* del microservicio *communities-ms* es el único que adopta un enfoque de validación funcional para el manejo de errores. En lugar de recurrir al mecanismo tradicional de excepciones, se utiliza un patrón basado en tipos como *Either* y *Result*, que explícitamente modelan los resultados de las operaciones.

Este enfoque asegura que tanto los casos de éxito como los de error sean gestionados de forma transparente, proporcionando un control más preciso sobre el flujo de ejecución y facilitando la gestión de posibles fallos.

En la implementación de este modelo, que se puede observar en los distintos servicios del módulo *communities*, cada operación retorna un resultado que indica si la ejecución fue exitosa o si ocurrió un error, dejando claro el tipo de fallo sin necesidad de lanzar excepciones.

A diferencia del enfoque basado en excepciones, donde un error se lanza y un filtro lo captura, el enfoque de validación funcional requiere que los errores sean gestionados explícitamente en el controlador. Este debe inspeccionar el resultado de cada operación, que se presenta como un tipo *Either*, y decidir cómo manejar cada tipo de error de forma específica, respondiendo con el código de estado HTTP y el mensaje de error correspondiente.

3.6 Autenticación y control de autorización

En este apartado se explicará cómo se ha implementado la autenticación y el control de autorización en la aplicación de SolidarianID.

3.6.1 Autenticación

En esta primera entrega, hemos implementado dos modelos de autenticación: la tradicional con usuario y contraseña, y la autenticación mediante OAuth2 con GitHub. La autenticación con usuario y contraseña requiere que el usuario ingrese su email y contraseña. En cambio, la autenticación mediante OAuth2 verifica que el email de la cuenta de GitHub esté registrado en el sistema. Si el email no está registrado, se lanza un error. No hemos implementado el registro completo por la falta de una interfaz de front-end que permita completar los datos necesarios desde GitHub.

Además, hemos creado un *AuthGuard* que protege las rutas de la aplicación mediante autenticación. Esta guarda verifica si el usuario proporciona un token JWT válido, y si es así, adjunta el payload (información del usuario) a la solicitud. Si el token es inválido o no se proporciona, se lanza una *UnauthorizedException*. Para las rutas que no requieren autenticación, hemos añadido el decorador *@Public*, que permite que esas rutas sean accesibles sin necesidad de un token. Esto es útil para funciones como el inicio de sesión con GitHub, donde el acceso no requiere que el usuario esté previamente autenticado.

En resumen, el *AuthGuard* asegura que las rutas protegidas sólo sean accesibles para usuarios autenticados, mientras que el decorador *@Public* facilita el acceso a rutas abiertas sin requerir autenticación previa.

3.6.2 Control de autorización

El control de autorización en la aplicación se maneja a través de un *RolesGuard* y un decorador *@Roles*. El *RolesGuard* se encarga de verificar si el usuario tiene el rol adecuado para acceder a una ruta específica. Cuando una ruta requiere un rol particular, se utiliza el decorador *@Roles* junto con el guardia *@UseGuards(RolesGuard)*.

Este guarda obtiene los roles necesarios desde los metadatos proporcionados por el decorador y compara el rol del usuario (almacenado en base de datos) con los roles permitidos para esa ruta. Si el rol del usuario coincide con alguno de los roles requeridos, se concede el acceso; en caso contrario, el acceso es denegado.

Este sistema de control de roles no se aplica de forma global, sino de manera específica en cada ruta que lo necesite, lo que permite proteger funcionalidades que deben ser accesibles únicamente para usuarios con ciertos roles. Por ejemplo, la validación de solicitudes de creación de comunidad sólo es accesible para usuarios con rol *admin*, es decir, administradores de la plataforma.

3.7 Mecanismo de comunicación

El sistema de comunicación en SolidarianID emplea un enfoque híbrido que combina eventos internos y externos, permitiendo una interacción fluida y desacoplada tanto dentro como entre microservicios.

3.7.1 Eventos Internos

Los eventos internos son gestionados mediante el *EventBus* de NestJS y facilitan la comunicación dentro de un mismo microservicio. Un ejemplo claro es el *UserFollowedEvent*, que se genera en el módulo de usuarios y se maneja en el módulo de historial (history). Este es capturado por un handler específico, '*user-followed.handler.ts*', que actualiza el historial del usuario. Este enfoque desacopla completamente ambos módulos, permitiendo que evolucionen de forma independiente sin romper la lógica de negocio, además de favorecer la aplicación de DDD.

3.7.2 Eventos Externos con Kafka

Para comunicar microservicios y bounded contexts diferentes, la aplicación utiliza Kafka como sistema de mensajería asíncrona. Los Domain Events, como *UserJoinedCommunity* o *ActionCreatedEvent*, son publicados en tópicos específicos mediante el *KafkaEventPublisherService*. Otros microservicios, como statistics-ms o users-ms, pueden suscribirse a estos tópicos y reaccionar en tiempo real, ya sea actualizando las estadísticas o el historial de actividad del usuario asociado, asegurando el desacoplamiento.

3.7.3 Flujo de Eventos

El flujo típico de eventos comienza con la emisión de un *DomainEvent* a través del *EventBus*. Si el evento es únicamente interno, se procesa mediante el handler en el módulo correspondiente, como sucede con los eventos del módulo usuarios. En cambio, si requiere propagación externa, como en el caso de los eventos del microservicio communities-ms, que cuentan con un listener centralizado para reenviar dichos eventos a Kafka, el *EventsService* se encarga de publicarlos en Kafka. Desde allí, los microservicios interesados pueden consumirlos.

La combinación de eventos internos y externos brinda al sistema flexibilidad y escalabilidad. Los eventos internos optimizan la comunicación dentro de cada microservicio, mientras que Kafka asegura la independencia y el desacoplamiento entre ellos. Este enfoque permite una sincronización eficiente y nos garantiza fiabilidad, incluso en escenarios de alta carga.

3.8 Documentación

Se ha documentado con OpenAPI la funcionalidad de listado, búsqueda y visualización de comunidades, causas y acciones, y la consulta del perfil público.

Para realizarlo, se han configurado las opciones de Swagger en los dos microservicios que lo utilizan, *communities-ms* y *users-ms* y se han anotado los endpoints de los controladores necesarios, así como los objetos DTO para realizar las peticiones (filtrado, ordenación y paginación de los listados). Swagger facilita la visualización interactiva de esta documentación, permitiendo probar los endpoints directamente desde la interfaz

La documentación está accesible a partir de la pasarela API, dentro de las rutas '*api/v1/doc/communities*' y '*api/v1/doc/users*'.

4 Frontend

En este apartado se detalla la implementación del frontend de la aplicación SolidarianID, desarrollado con un enfoque basado en la arquitectura **MVC** (Modelo-Vista-Controlador). Para garantizar una interfaz eficiente y bien estructurada, se ha utilizado Handlebars como motor de plantillas, junto con HTML para la estructura del contenido, y Tailwind como framework de CSS para lograr un diseño visual atractivo y responsivo.

Además, para obtener los datos necesarios y mostrarlos en pantalla, el frontend realiza llamadas a los microservicios que hemos desarrollado a través de la biblioteca **Axios**. Esto asegura una comunicación ágil y eficiente con la capa de servicios del sistema, permitiendo que la aplicación funcione de manera dinámica al consumir y renderizar información en tiempo real.

4.1 Adaptaciones responsivas en las vistas con Tailwind

Para la entrega del primer cuatrimestre se han desarrollado las siguientes vistas del administrador de la plataforma:

- Vista para la validación de las solicitudes de creación de una comunidad.
- Vista para ver las estadísticas de uso de la plataforma.
- Vista para ver los detalles sobre una comunidad en concreto y la generación de un informe con dichos detalles en formato PDF.

Adicionalmente, hemos implementado la vista de la página principal, ***Home***, y del inicio de sesión, ***Login***. Así conseguimos distinguir el rol del usuario que entra a la plataforma, debido a que las vistas implementadas son exclusivas para el administrador de la plataforma.

Hemos elegido Tailwind como framework CSS para dar estilo a nuestras vistas, porque permite crear un diseño limpio, moderno y responsivo de manera rápida y eficiente, gracias a sus clases utilitarias predefinidas. Estas clases facilitan la personalización de estilos como el espaciado, tipografía y adaptabilidad sin necesidad de escribir código CSS adicional, logrando un enfoque **mobile-first** y optimizando el desarrollo.

A continuación se explican las adaptaciones *responsive* para cada una de las vistas implementadas y las principales clases de Tailwind usadas para conseguir que el diseño sea responsivo.

4.1.1 Home

En la Figura 4, podemos ver la vista *Home* para usuario sin iniciar sesión en la aplicación. Debido a la falta de tiempo, a partir de esta página sólo se puede acceder a la funcionalidad del Login.

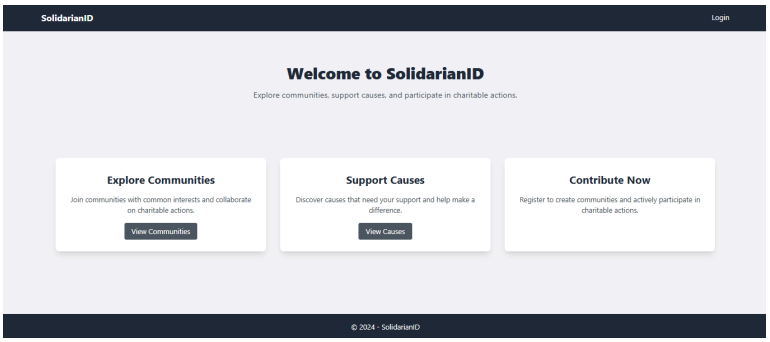

Figura 4. Home

Esta vista es responsive por el uso de algunas clases del framework de Tailwind que permite adaptar el diseño automáticamente al tamaño de la pantalla:

- **Grid adaptable**: la clase *grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3* cambia el número de columnas de 1 (móvil) a 2 (tablet) y 3 (escritorio), ajustando el contenido según el ancho disponible.
- **Contenedor fluido**: *container mx-auto px-4*, centra el contenido y añade espacio horizontal para pantallas pequeñas
- **Tipografía escalable**: las clases como *text-4xl* o *text-lg* garantizan que los tamaños de texto se ajusten al diseño.
- **Menú de hamburguesa**: usando *media query* para el control de la aparición de los elementos dentro del menú cuando alcanza un determinado tamaño.

Además, la vista es mobile-first, debido a que empieza con un diseño para dispositivos pequeños (1 columna) y se adapta a pantallas más grandes usando prefijos *md:* y *lg:*.

Cuando un usuario hace el login, en la barra de navegación podemos ver la entrada para acceder al *Dashboard*, que incluye las vistas para el administrador de la plataforma. Además, podemos cerrar sesión con el *Logout* y ver el rol que tiene el usuario.

4.1.2 Login

En la Figura 5, podemos ver la vista del Login. Al igual que la vista del Home, es responsive por el uso de algunas funciones del framework de Tailwind:

- **Contenedor ajustable**: el uso de *max-w-md* limita el ancho máximo a 768px, evitando que sea demasiado ancho en pantallas grandes y adaptándose bien a móviles.
- **Centrado dinámico**: el uso de *mx-auto* hace que el formulario se centre horizontalmente, garantizando una buena alineación en cualquier dispositivo.

- **Tipografía escalable:** el uso de clases como *text-2xl* y *text-sm* aseguran el tamaño de texto para distintas resoluciones.
- **Elementos que ocupan todo el ancho**: el uso del *w-full* hace que los elementos ocupen todo el ancho disponible, adaptándose a pantallas más pequeñas sin romper el diseño.

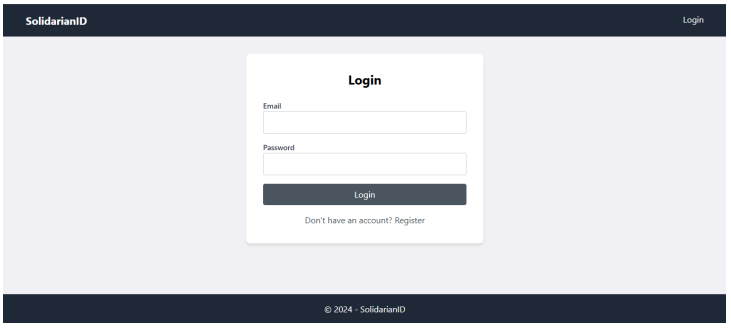

Figura 5. Login

4.1.3 Validation

En la Figura 6, podemos ver la vista en la que aparecen las solicitudes de creación de comunidad pendientes de validar. Esta vista también es responsive por el uso de las clases del framework de Tailwind:

- **Reorganización de elementos**: el uso *flex, flex-col, sm:flex-row* permite organizar los elementos en columna para pantallas pequeñas y en fila para pantallas grandes.
- **Ancho adaptativo**: usando *w-full* para que los botones y campos de entrada ocupen todo el ancho disponible en dispositivos pequeños.
- **Ancho del modal ajustable**: el uso del *sm:w1/3* hace que el modal ocupe todo el ancho en pantallas pequeñas y solo un tercio en pantallas grandes.
- **Espaciado dinámico**: usando la clase *space.y.-4* para definir el espaciado vertical en móviles y *sm:space-x-4* para el espaciado horizontal en pantallas grandes.
- **Desplazamiento en listas largas**: el uso de *max-h-64* y *overflow-y-auto* hace que el contenido dentro de un contenedor no sobresalga ni desorganice el diseño.

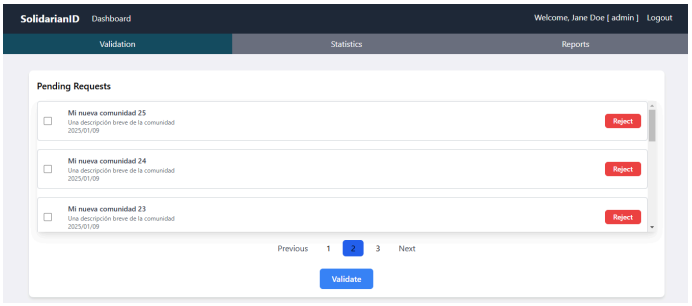

Figura 6. Solicitudes de creación de comunidad pendientes

4.1.4 Statistics

En la Figura 7, podemos ver la vista que muestra las gráficas de uso de la plataforma. Esta vista, a diferencia de las anteriores, usa CSS puro para conseguir el diseño responsive. Hemos

usado *flexbox*, *media queries* y propiedades como *flex-basis*, *max-width* y *w-full* para ajustar el diseño según el tamaño de la pantalla:

- En pantallas grandes, los gráficos se organizan en filas de dos, ocupando el 45% del ancho.
- En pantallas medianas (como tabletas), los gráficos se reorganizan para ocupar el 48% del contenedor, adaptándose a pantallas más pequeñas sin perder la estructura.
- En pantallas pequeñas (móviles), los gráficos se apilan uno sobre otro, ocupando el 100% del ancho, lo que facilita su visualización.

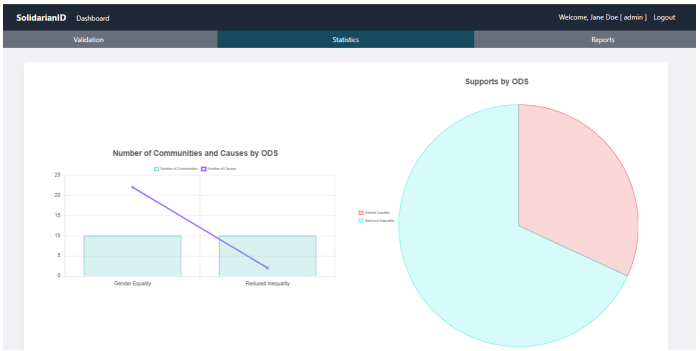

Figura 7. Estadísticas

4.1.5 Reports

En la Figura 8, se muestra la vista para seleccionar una comunidad junto a la funcionalidad de generar un informe en formato de PDF. Usa Tailwind para conseguir un diseño responsive:

- **Uso de flexbox**: *flex*, *flex-col* y *sm:flex-row* hacen que los elementos se alineen verticalmente en pantallas pequeñas y horizontalmente en pantallas más grandes.
- **Espaciado**: usando *sm:space-x-4* para ajustar el espacio horizontal en pantallas grandes y *mb-4* para agregar márgenes verticales en pantallas pequeñas.

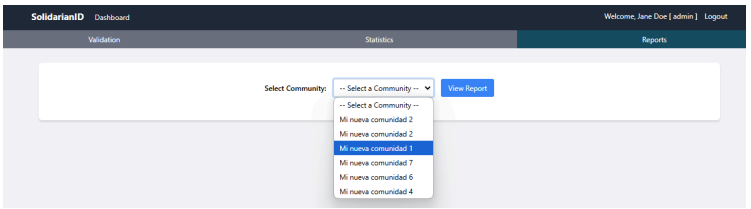

Figura 8. Estadísticas

En la Figura 9, muestra la vista de detalles de la comunidad seleccionada en la vista anterior. Utiliza las siguientes características de Tailwind:

- **Espaciado adaptable**: usando *p-4,* que aplica un espaciado uniforme en toda la pantalla en pantallas pequeñas y *md:p-6* y *lg:p8* que aumentan el espaciado en pantallas medianas y grandes, respectivamente.
- **Texto adaptativo**: el tamaño del texto de los títulos y párrafos cambia según el tamaño de la pantalla, usando:
  - *text-sm* en pantallas pequeñas
  - *md:text-lg* en pantallas medianas

- *lg:text-xl* en pantallas grandes

- **Lista con sangría flexible**: los márgenes de la lista se ajustan para ofrecer una sangría mayor en pantallas más grandes, con clases como:
  - *ml-4* en pantallas pequeñas
  - *md:ml-6* en pantallas medianas
  - *lg:ml-8* en pantallas grandes
- **Botón adaptable:** El tamaño del botón se adapta con:
  - *px-4* en pantallas pequeñas
  - *md:px-6* en pantallas medianas
  - *lg:px-8* en pantallas grandes

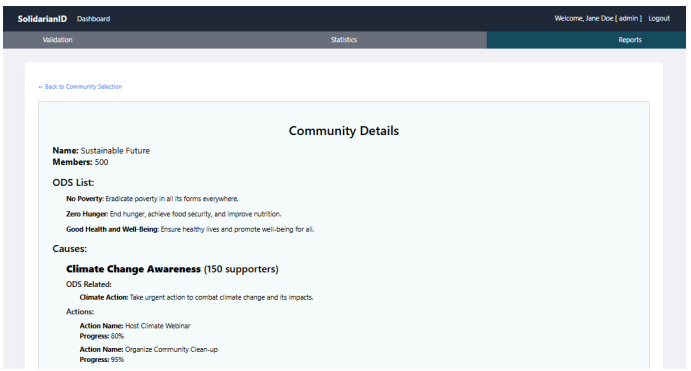

Figura 9. Detalles de una comunidad

Cabe destacar que en la mayoría de las vistas hemos utilizado algunos de los modificadores de tamaño de pantalla proporcionados por Tailwind CSS:

- **sm**: aplica estilos a pantallas con un ancho mínimo de 640px.
- **md:** aplica estilos a pantallas con un ancho mínimo de 768px.
- **lg:** aplica estilos a pantallas con un ancho mínimo de 1024px.
- **xl:** aplica estilos a pantallas con un ancho mínimo de 1280px
- **2xl**: aplica estilos a pantallas con un ancho mínimo de 1536px.

Estos modificadores permiten adaptar el diseño de la interfaz a diferentes tamaños de pantalla, asegurando una experiencia de usuario óptima en una amplia variedad de dispositivos.

4.2 Patrón de delegación de eventos

La delegación de eventos es una técnica en la que un único manejador de eventos se asigna a un contenedor o elemento superior, en lugar de asignarlo a cada elemento individual. Esto permite manejar eventos en elementos dinámicos sin necesidad de volver a asignar los manejadores.

En este proyecto, se optó por usar manejadores de eventos individuales en lugar de delegación de eventos debido a la naturaleza estática y predecible de los elementos del DOM. Dado que los elementos no se generan dinámicamente, asignar manejadores específicos a cada uno

permite un control más directo y simplifica la lógica al evitar condicionales adicionales en un manejador centralizado.

Además, este enfoque mejora la claridad del código y facilita la depuración, ya que cada evento está estrechamente vinculado a su elemento origen. Al tratarse de un número limitado de elementos interactivos, el impacto en el rendimiento es mínimo, lo que hace que los manejadores individuales sean una solución adecuada y práctica en este caso.

A continuación, se definen los manejadores de eventos definidos para cada vista.

4.2.1 Manejador de eventos en la vista de *Validation*

Para esta vista, se han definido los siguientes manejadores de eventos:

- ***handleValidateFormSubmit***: este manejador se activa con el evento *submit* en el formulario con el ID *validate-request-form* (para validar las solicitudes pendientes). Su principal función es verificar si el usuario ha seleccionado al menos una solicitud. Si no ha seleccionado nada, muestra una alerta avisando de que al menos necesita seleccionar una. En caso de que se haya seleccionado alguna solicitud, sus IDs se renderizan en formato JSON y se asignan a un campo oculto, *selected-requests-input*, para ser enviados junto con el formulario.

- ***openRejectModal***: este manejador se activa con el evento *click* en el botón de *Reject* de cada solicitud. Su principal función es mostrar un modal para escribir el motivo del rechazo. Además, asigna el *requestId* de la solicitud seleccionada al atributo *data-request-id* del modal para ser utilizado más tarde cuando se confirme el rechazo.

- ***closeRejectModal***: este manejador se ejecuta con el evento *click* en el botón de *Cancel* dentro del modal de rechazo. Su función principal es cerrar y ocultar el modal.

- ***handleRejectConfirm***: este manejador se activa con el *click* en el botón de *Submit* dentro del modal de rechazo. Su principal función es almacenar el id de la solicitud y el motivo de rechazo y enviarlos al servidor para gestionar el rechazo.

- ***setActivePageLink***: este manejador se ejecuta con el evento de *DOMContentLoaded*, que se dispara cuando el contenido del DOM ha sido completamente cargado. Su principal función es resaltar el enlace de la página actual en los controles de paginación.

4.2.2 Manejador de eventos en la vista de *Statistics*

El único manejador de eventos usado en esta vista es *window.onload.* Este manejador se ejecuta cuando la ventana y todo su contenido han sido completamente cargados. Su principal función es procesar los datos obtenidos del servidor y luego invocar las funciones que generan los distintos tipos de gráficos.

4.2.3 Manejador de eventos en la vista de *Reports*

En la vista de *Reports* se han definido dos manejadores de eventos:

- ***checkbox.addEventListener('change')***: este manejador se activa cuando el estado del checkbox con el ID *include-graphics* cambia. Su principal función es verificar si el checkbox está marcado o no. Si está marcado, muestra el contenedor de gráficos, en caso contrario lo oculta, permitiendo al usuario elegir si desea incluir o no las gráficas en el informe.

- ***generatePdfButton.addEventListener('click')***: este manejador se activa cuando se hace click al botón de *Generate PDF Report*. Su principal función es generar un informe en formato PDF utilizando la librería de *html2pdf*. Primero captura las gráficas como imágenes en caso de que se haya seleccionado la opción de incluir las gráficas, luego oculta elementos específicos en la vista para mejorar la apariencia del informe. Después, configura las opciones del PDF y genera y guarda el archivo PDF con el informe, restaurando los elementos ocultos después de completar la operación.

4.2.4 Manejador de eventos en las barras de navegación

Adicionalmente, tenemos dos manejadores de eventos para controlar el comportamiento de las barras de navegación.

Para la barra de navegación principal, *navbar.hbs,* tenemos un manejador que se define mediante la función de *addEventListener* que escucha el evento *click*. Este manejador se activa cuando el usuario hace click sobre el elemento con ID *menu-toggle*. Su principal función es realizar un cambio en el estilo de visibilidad del menú móvil al alternar la clase *hidden*.

Para la barra de navegación de las vistas que tiene el Dashboard, *adminnavbar.hbs*, también tenemos un manejador de eventos. En este caso es *DOMContentLoaded*, que se activa cuando el documento HTML ha sido completamente cargado y procesado. Este manejador se define en la función *setActiveLink*, que se encarga de verificar la URL actual de la página y luego aplicar una clase CSS para resaltar el enlace activo según la sección de la aplicación que el usuario esté visualizando.

4.3 Generación de gráficos

En este apartado se describe el proceso de generación de gráficos dinámicos utilizando la librería Chart.js, que permite crear representaciones visuales interactivas en la página web. Para obtener los datos necesarios, se realizan llamadas a los microservicios mediante Axios, una popular librería de JavaScript para hacer peticiones HTTP. Una vez que los datos son recuperados correctamente, se renderizan en el frontend utilizando archivos de plantilla *.hbs* (Handlebars), que permiten integrar y mostrar de manera eficiente los gráficos generados.

Para este entregable se han definido tres tipos de gráficas:

- Una gráfica combinada de barras y líneas que representa la información sobre el número de comunidades y causas por ODS.
- Dos gráficas circulares (*Pie Chart*) que representan la información porcentual sobre el número de apoyos por ODS y la información porcentual de apoyos para cada comunidad.
- Una gráfica de área polar (*polarArea*) que representa la media porcentual del progreso de las acciones por comunidad.

Para cada tipo de gráfica se ha implementado una función para su creación. A continuación, se explica con más detalle cómo se crean las gráficas y cómo podemos configurar su apariencia.

Debido a que no sabemos cuántos elementos vamos a disponer, hemos creado una función auxiliar que genera colores aleatorios, para no repetir colores en las gráficas. La función se llama *generateColors(count)*.

Para cada gráfica empezamos con la obtención del contexto de dibujo del elemento HTML donde se debe renderizar la gráfica (ver Figura 10).

```
const ctx = document.getElementById(elementId)?.getContext('2d');
```

Figura 10.Contexto

En la Figura 11, se crea un nuevo gráfico utilizando la clase *Chart* de Chart.js, proporcionando como parámetro el contexto del lienzo (ctx). El tipo de gráfico se especifica mediante la propiedad *type*, donde se indica el tipo de visualización deseado. Los datos que se mostrarán en el gráfico se definen dentro de *datasets* en la propiedad *data*. La propiedad *label* se emplea para asignar una etiqueta o nombre que describa cada conjunto de datos en el gráfico. Para personalizar los colores, se utilizan las propiedades *backgroundColor* y *borderColor*, que permiten definir los colores del fondo y los bordes de los elementos gráficos, respectivamente.

Figura 11. Creación de una gráfica

En la Figura 12, podemos ver la propiedad *options*, que es una parte crucial de la configuración de las gráficas. En este caso tenemos varias opciones para personalizar el comportamiento de la gráfica. La opción *responsive: true* asegura que el gráfico se ajuste automáticamente al tamaño de su contenedor. En *plugins*, se configuran tres elementos: *title*, que muestra un título en el gráfico con un tamaño de fuente de 18 píxeles; *legend*, que coloca la leyenda en la parte superior y la muestra solo en pantallas grandes (mayores a 768 píxeles), con un tamaño de fuente de 8 píxeles y ajustes para el tamaño de los cuadros de la leyenda; y *tooltip*, que habilita los tooltips para mostrar información adicional al pasar el ratón sobre los puntos del gráfico.

Figura 12. Opciones de la configuración

El ejemplo anterior es una gráfica combinada de barra y línea, pero la estructura y las propiedades aplicadas son muy similares en las demás gráficas.

4.4 Generación de informes

En este apartado se explica el proceso utilizado para generar los informes, describiendo específicamente las herramientas y métodos empleados. En este caso, se ha optado por el uso de *html2pdf*, una librería que permite convertir contenido HTML a archivos PDF de forma sencilla y rápida. *html2pdf* facilita la creación de documentos PDF a partir de código HTML,

lo que permite aprovechar las capacidades de diseño web como estilos CSS y estructura HTML para obtener un informe bien formateado.

Esta herramienta es especialmente útil cuando se desea mantener la flexibilidad de diseño y la posibilidad de personalizar el contenido del PDF mediante diversas opciones, como la inclusión de márgenes, tamaño de página y orientación. A continuación, se detallan las opciones específicas utilizadas para configurar el diseño del PDF.

En la Figura 13, podemos ver la implementación del manejador de eventos que hemos mencionado anteriormente. Dentro de *option* se define la configuración del informe que se va a generar:

- ***filename***: indica el nombre del pdf que se va a generar.
- ***jsPDF***: permiten personalizar el archivo pdf y ajustar su presentación. En este caso es un pdf de tamaño A4 con orientación vertical y estable la unidad de medidas para las dimensiones del documento en milímetros.
- ***html2canvas***: configura la librería *html2canvas* para capturar contenido HTML y convertirlo en una imagen. Se establece una resolución con *scale: 2*, un fondo blanco con *backgroundColor: '#ffffff'*, se habilita el acceso a recursos externos con *useCORS: true*, y se desactiva el registro en consola mediante *logging: false*. Estas opciones aseguran una captura de alta calidad y un manejo adecuado de recursos externos sin generar registros innecesarios.
- ***image***: estable el formato de la imágen como JPEG con una calidad de 0.98 siendo 1 la máxima calidad.
- ***margin***: establece el margen del documento.
- ***pagebreak***: Establece las reglas para la ruptura de páginas. *mode: 'avoid-all'* evita que el contenido se divida entre páginas de manera automática. Además, la opción *before: '#avoid-page-break'* especifica que la ruptura de página debe evitarse antes del elemento HTML con el ID *#avoid-page-break*, lo que garantiza que ese contenido se mantenga en una sola página.

Además, tenemos varias funciones auxiliares para mejorar el documento pdf final:

- ***captureChartAsImages()***: función que captura las gráficas y las convierte en imágenes.
- ***ocultarElementos()***: oculta los elementos que no deben aparecer en el pdf, ya que *html2pdf* coge todos los elementos y los pinta en el documento, pero hay elementos que no queremos que aparezcan, como botones o checkbox.
- ***restaurarElementos()***: función para restaurar los elementos ocultados con la función *ocultarElementos()*.

```
generatePdfButton.addEventListener('click', () => {
  const element = document.getElementById('community-details');
  captureChartsAsImages();
  const options = {
    filename: 'community-report.pdf',
    jsPDF: {
      unit: 'mm',
      format: 'a4',
      orientation: 'portrait',
    },
    html2canvas: {
      scale: 2,
      backgroundColor: '#ffffff',
      useCORS: true,
      logging: false,
    },
    image: {
      type: 'jpeg',
      quality: 0.98,
    },
    margin: [20, 20, 10, 10],
    pagebreak: { mode: 'avoid-all', before: '#avoid-page-break' },
  };
  ocultarElementos();
  // Generar el PDF

  html2pdf()
    .from(element)
    .set(options)
    .save()
    .then(() => {
      restaurarElementos();
    });
});
```

Figura 13. Configuración del formato del PDF

5 Conclusiones

En conclusión, a través del desarrollo del proyecto SolidarianID, se han adquirido valiosos conocimientos tanto a nivel técnico como metodológico. En el ámbito técnico, se ha profundizado en el diseño y desarrollo de aplicaciones basadas en microservicios, lo que permitió entender mejor la escalabilidad y la independencia de los componentes en sistemas distribuidos. La implementación de arquitecturas como la arquitectura limpia y Domain-Driven Design (DDD) ha proporcionado una comprensión más profunda de cómo estructurar aplicaciones de manera eficiente, asegurando que la lógica del negocio esté alineada con las necesidades del sistema.

Además, se ha aprendido a trabajar con tecnologías modernas como NestJS, Tailwind CSS y Kafka, que han sido fundamentales en el desarrollo tanto del backend como del frontend. El manejo de la autenticación, el control de acceso y la generación de informes también ha sido una parte crucial del aprendizaje, permitiendo comprender la importancia de la seguridad y la experiencia de usuario. Finalmente, a nivel metodológico, el trabajo en equipo y la implementación de buenas prácticas de desarrollo ágil y control de versiones con GitHub han permitido mejorar la colaboración y gestión del proyecto.

Lo único que nos gustaría comentar y que creemos que podría mejorarse a nivel de asignatura es el orden en el que se presentan los conceptos. Consideramos que sería beneficioso adelantar la parte de microservicios, ya que de esta manera podríamos contar con los conocimientos necesarios para desarrollar el backend de la aplicación antes de comenzar con el desarrollo del frontend. En nuestro caso, pasamos bastante tiempo realizando refactorizaciones porque al principio no utilizamos la estructura de microservicios, lo que nos llevó a una reestructuración más tarde. Como resultado, tuvimos un tiempo muy limitado para el desarrollo y no tuvimos la oportunidad de realizar mejoras importantes.

6 Bibliografía

1. NestJS Documentation. (s.f.). NestJS Documentation
2. Tailwind CSS Documentation. (s.f.). Tailwind CSS
3. Handlebars.js Documentation. (s.f.). Handlbars
4. HTML Living Standard. (s.f.). WHATWG. https://html.spec.whatwg.org

5. CSS: Cascading Style Sheets. (s.f.). MDN Web Docs .https://developer.mozilla.org/en-US/docs/Web/CSS

6. REST API Tutorial: How to Design a REST API. (s.f.). https://restfulapi.net/rest-api-design-tutorial-with-example/

7. Recursos del Aula Virtual. (s.f.). Universidad de Murcia. https://aulavirtual.um.es

8. Martinez, P. (2021, julio 9). *Hexagonal Architecture, there are always two sides to every story*. SSENSE-TECH. https://medium.com/ssense-tech/hexagonal-architecture-there-are-always-two-sides-to-every-story-bc0780ed7d9c

9. Hauer, P. (2015, marzo 4). *RESTful API Design. Best Practices in a Nutshell*. Philipp Hauer's Blog. https://phauer.com/2015/restful-api-design-best-practices/

10. Stemmler, K. (s.f.). *white-label: A Vinyl-Trading enterprise app built with Node.js + TypeScript using Domain-Driven Design*. GitHub. https://github.com/stemmlerjs/white-label