

Documentación del proyecto fin de grado

Información específica y general de la aplicación web



Autores:

Aitor Pascual Jimenez

Agustín Antonio Marquez Piña

Tutor:

Emilio Valencia

DAW (2023 - 2025)

Índice

Índice	1
1º. - Resumen	4
2º. - Introducción	5
2.1 - Síntesis del proyecto	5
2.2 - Módulos implicados en el desarrollo	5
3º. - Justificación del proyecto y objetivos	6
3.1. Estado del arte y necesidades del mercado	6
3.2. Objetivos del proyecto	6
3.2.1. Objetivos generales	6
3.2.2. Objetivos específicos	6
4º. - Análisis: ¿Qué hará la aplicación?	7
4.1. Gestión de requisitos	7
4.1.1. Análisis de requisitos	7
4.1.1.1. Requisitos de negocio	7
4.1.1.2 Requisitos de usuario	7
4.1.1.3 Requisitos funcionales	8
4.1.1.4. Requisitos no funcionales	11
4.1.1.5. Priorización y dependencias	12
4.1.2. Metodología Agile	13
4.1.2.1. Definición y principios del Agile	14
4.1.2.2. Marco escogido: Kanban	14
4.1.2.3. Sprint 0 y Kanban	15
4.1.3. Historias de usuario	16
4.1.3.1. Historias de usuario como alumno/cliente	16
4.1.3.2. Historias de usuario como personal de cafetería	17

4.1.3.3. Historias de usuario como administrador	17
4.1.3.4. Historias de usuario como sistema	17
4.1.4. Backlog inicial	17
4.2. Herramientas gráficas de análisis (para dar visión global)	19
4.2.1. Diagrama de entidades	19
4.2.2. Diagramas Entidad-Relación (E/R)	20
4.2.3. Modelo de datos	20
4.2.4. Diagramas de clase	20
4.2.5. Casos de uso y tabla requisitos	21
4.2.6. Diagrama de casos de uso	23
4.3. Planificación de realización del proyecto (cronograma genérico)	23
5º. - Diseño: ¿Cómo se hará la aplicación?	23
5.2. Diseño de la base de datos	23
Identificar las posibles entidades.	23
Identificar las posibles interrelaciones.	24
Identificar las posibles entidades-relaciones.	24
Identificar los atributos.	25
Atributos de users:	25
Atributos de userAuthorities:	25
Atributos de cart:	25
Atributos de products:	25
Atributos de orders:	25
Atributos de cartHasProducts:	25
Atributos de ordersHasProductos:	25
Desarrollo práctico.	26
5.3. Diseño de la interfaz de usuario (wireframes, mockups)	26
Wireframes	26
frontend - cliente	26
TopMenu (header):	26
Footer(footer):	27

Modales de autenticación:	28
Inicio	29
Carrito	29
Pedidos	30
frontend - personal	31
6º. - Implementación y pruebas	31
6.1. Estructura del proyecto y flujo Git	31
6.2. Tecnologías y dependencias utilizadas	34
6.2.1 Tecnologías y dependencias en el backend	35
6.2.2 Tecnologías y dependencias en el frontend	36
6.3. Desarrollo por módulos	36
6.3.1. Módulo de autenticación:	36
6.3.2. Módulo de gestión de pedidos	36
6.3.3 Módulo de servidor de pedidos en tiempo real	37
6.3.4 Módulo de envío de correos	37
6.3.5 Módulo de pagos	37
6.3.6 Módulo frontend para clientes	37
6.3.7 Módulo frontend para personal	38
6.3.8 Módulo de base de datos	38
6.3.9 Módulo de proxy inverso	38
6.4. Plan de pruebas	38
6.4.1. Pruebas unitarias	38
6.4.2. Pruebas de integración	39
7º. - Implantación	40
7.1. Pipeline CI/CD	40
7.2. Despliegue en entorno de producción	40
7.3. Guía de instalación y configuración	41
8º. - Resultados y discusión	41
8.1. Comparativa temporal: planificado vs. ejecutado	41
8.2. Dificultades más importantes encontradas	42

8.3. Hasta dónde se ha llegado: funcionalidades	42
8.4. Posibles extensiones y mejoras futuras	42
9º. - Conclusiones	43
9.1. Aportación al aprendizaje y competencias adquiridas	43
Desarrollo Frontend:	43
Desarrollo Backend:	43
Gestión de datos:	43
Herramientas de desarrollo:	44
Gestión de proyectos:	44
Trabajo en equipo:	44
Pensamiento crítico:	44
Aprendizaje específico del dominio:	44
Seguridad y autenticación:	44
Arquitectura moderna:	44
10º. - Bibliografía y referencias	45
ANEXO A - SEGURIDAD Y AUTENTICACIÓN	46
ANEXO B - ARQUITECTURA DE PAQUETES	60
ANEXO C - SOCKET.IO	63
ANEXO D - ASTRO	72
ANEXO E - PAGINACIÓN	80

1º. - Resumen

Vivimos en tiempos modernos, la era de la tecnología, de los servicios digitales, de la agilización de los servicios cotidianos, es por eso que surge como necesidad el mejorar la eficiencia de los procesos cotidianos y las labores del día a día incrementando la satisfacción del usuario en general al mismo tiempo que se fomenta y facilita el acceso a bienes y servicios para todas las personas.

Cuando elegimos una de las docenas de ideas que tuvimos para realizar nuestro proyecto nos vino en mente algo, pasamos cientos de horas cursando nuestros estudios y uno de los ambientes que más frecuentamos y que menos digitalización de sus servicios tiene está en nuestro propio instituto, en la cafetería, sabiendo la relación de gran calidad que nosotros, nuestros compañeros y profesores tienen con el personal de la cafetería vimos un gran oportunidad, enfocar nuestro tiempo, nuestros recursos y nuestros conocimientos adquiridos en el ciclo de formación de grado superior en desarrollo de aplicaciones web (DAW) para diseñar, crear e implementar una solución/aplicación tecnológica real y útil para la cafetería.

Así nació nuestra idea de proyecto como un sistema de gestión de pedidos para cafeterías de instituto (escalable a cafeterías de cualquier tipo) basado en una arquitectura de microservicios (gracias Aitor por seguirme enseñando cada día más cosas nuevas) que permite digitalizar, agilizar y gestionar de forma eficiente toda la lógica de negocio, entre ella los procesos de pedido, pago y notificaciones en tiempo real sobre el estado de los mismo.

Nuestro objetivo principal es contundente, facilitar la vida de todos los usuarios de la aplicación (alumnos, profesores, personal del instituto, visitantes, etc) así como a los encargados de gestionarla (personal de la cafetería), agilizando la atención, permitiendo una administración más eficientes de los recursos y al mismo tiempo no solo poner en práctica todo lo aprendido en los últimos casi dos años, si no seguir aprendiendo nuevas tecnologías, tales como metodologías o patrones de diseño del mercado laboral actual, integración de pagos online, autenticación segura, comunicación en tiempo real e incluso el despliegue de la propia aplicación utilizando contenedores, preparándonos cada vez un poco más para un mercado laboral cambiante, evolutivo y emocionante.

2º. - Introducción

2.1 - Síntesis del proyecto

Este proyecto se ha desarrollado como parte de la formación del CFGS Desarrollo De Aplicaciones Web como una prueba fundamental de todos los conocimientos adquiridos y las habilidades desarrolladas en el transcurso de este ciclo, en él recopilamos toda la teoría y la práctica, así como metodologías y patrones de diseño y se utilizan de forma organizada para diseñar y desarrollar aplicaciones web sostenibles, escalables, robustas y seguras.

2.2 - Módulos implicados en el desarrollo

Asignaturas de primero

- **PROGRAMACIÓN** — Nuestra aplicación tiene microservicios en Java y usa el paradigma de la orientación a objetos.
- **ENTORNOS DE DESARROLLO** — Hacemos uso de un repositorio de control de versiones (github) y de pruebas unitarias, además hacemos 3 iteraciones en un ciclo de vida iterativo incremental.
- **LENGUAJE DE MARCAS** — Utilizamos HTML y CSS para el desarrollo frontend de nuestras páginas web, además del uso de XML en algunos ficheros de configuración.
- **BASES DE DATOS** — Nuestro proyecto utiliza una base de datos relacional MySQL y hemos elaborado un esquema Entidad Relación de nuestra base de datos normalizada en FNBC.
- **SISTEMAS INFORMÁTICOS** — Empleamos un sistema operativo Linux para el servicio de nuestra aplicación en una red IPV4 a través del protocolo HTTP.

Asignaturas de segundo

- **DESARROLLO EN ENTORNO SERVIDOR** — Nuestra aplicación utiliza API RESTFUL para comunicarse, algunas de ellas desarrolladas en Java Spring.
- **DESARROLLO EN ENTORNO CLIENTE** — Utilizamos el lenguaje JavaScript tanto para nuestro frontend como para parte del backend.
- **DESPLEGUE DE APLICACIONES** — Hacemos uso de Docker para el despliegue contenedrizado de nuestros servicios.
- **INTERFACES WEB** — Nuestras interfaces web son completamente responsive a los tamaños estándar y emplean algunos trucos aprendidos en el transcurso de la asignatura.

Para buena suerte de nuestro equipo (Aitor y mi persona) documentamos todo el conocimiento adquirido en **DAWº1** y **DAWº2** en documentos que detallan todas las actividad, la teoría y la práctica de las asignaturas, siendo los dos ejemplares más completos, públicos y a disposición de todos las [Guía de contenido en Entornos de Desarrollo](#) y la [Guía de contenido en Programación](#) siendo de las cuales sacamos el principal flujo del

3º. - Justificación del proyecto y objetivos

3.1. Estado del arte y necesidades del mercado

En nuestra investigación del estado del arte, nos dimos cuenta de que todas las aplicaciones que cumplen este cometido de pedidos en cafetería en tiempo real, son más enfocadas en pedidos a domicilio (Glovo, Uber Eats), desarrolladas únicamente para una cadena de cafeterías en concreto (Starbucks, Tim Hortons), también encontramos una solución más global para pequeñas cafeterías abiertas a todo el público (Kyte), pero no encontramos ninguna aplicación orientada a la gestión de una cafetería para un público concreto delimitado por una zona de estudio o trabajo, además en nuestra experiencia personal trabajando en grandes empresas, no nos hemos topado con ningún servicio por el estilo ya implantado o en estado de desarrollo en alguna de ellas, fuera de nuestra experiencia desconocemos si en alguna otra empresa se ha tratado de implantar este sistema de una forma privada.

3.2. Objetivos del proyecto

Nuestro objetivo es desarrollar una aplicación web que nos permita realizar pedidos pagados y no pagados de forma rápida y segura, diluyendo los picos de trabajo de los trabajadores de la cafetería en horas punta y facilitando al personal del centro la capacidad de recoger pedidos entre clases de una forma ágil.

3.2.1. Objetivos generales

Nos referimos a “objetivo general” cuando hablamos del resumen breve y orientado a desarrollo de lo que se pretende diseñar o crear, en ese sentido, el objetivo general de nuestro proyecto de grado es desarrollar una aplicación web integral para la gestión de pedidos en la cafetería del instituto, que digitalice y automatice el proceso de realización, gestión y pago de pedidos, mejorando la eficiencia operativa, la experiencia de usuario y la transparencia en la administración.

3.2.2. Objetivos específicos

Sin embargo, cuando hablamos de “objetivos específicos” se refiere más concretamente a las acciones/operaciones/agilizaciones que se desean crear y desarrollar con la aplicación, la lista puede sufrir modificaciones pero en la versión actual del documento son:

- 1. Permitir a los usuarios realizar pedidos de forma sencilla y rápida desde cualquier dispositivo (ordenador, tablet o móvil).**
- 2. Implementar un sistema de autenticación seguro que gestione diferentes roles de usuario (clientes, personal de cafetería, administradores).**
- 3. Facilitar la gestión interna de la cafetería mediante una interfaz para el personal que permita visualizar, actualizar y servir pedidos en tiempo real.**
- 4. Integrar pasarelas de pago online para permitir el pago seguro y automatizado de los pedidos.**

5. Ofrecer notificaciones y actualizaciones en tiempo real tanto a clientes como al personal de la cafetería, utilizando tecnologías como WebSockets.
6. Registrar y almacenar el histórico de pedidos y transacciones para su posterior consulta, análisis y auditoría.
7. Garantizar la escalabilidad, seguridad y mantenibilidad del sistema mediante una arquitectura basada en microservicios y el uso de contenedores.
8. Facilitar la integración con servicios externos (correo electrónico, bancos, APIs de pago) para ampliar las funcionalidades y mejorar la experiencia del usuario.
9. Proporcionar una documentación clara y completa para facilitar el despliegue, mantenimiento y futuras ampliaciones del sistema.

Promover el crecimiento económico y sustancial de la cafetería al aprovechar los recursos como tiempo (el tiempo ahorrado en la captación de pedidos) para mejorar o desarrollar otros sectores de la lógica propia del negocio.

4º. - Análisis: ¿Qué hará la aplicación?

4.1. Gestión de requisitos

Para la gestión de requisitos hemos puesto en práctica especialmente el contenido aprendido en entornos de desarrollo en las unidades 1, 2, 3, 4 y 6 donde hemos aprendido a realizar análisis de requisitos basados en la lógica de negocio, las necesidad de usuario, dependencias, etc, dando paso a la siguiente estructura del documento.

4.1.1. Análisis de requisitos

Para el análisis de requisitos pasamos por varias entrevistas previas con el personal de la cafetería y de otros alumnos lo que nos llevó a mantener un vínculo activo y constante con ello que nos permitió dar con los requisitos principales del negocio y de los usuarios finales explicados en los siguientes puntos.

4.1.1.1. Requisitos de negocio

Los principales requisitos de negocio para la cafetería del instituto serían la capacidad de facilitar las órdenes y los encargos de profesores y alumnos al momento de querer adquirir productos de una forma rápida y eficiente, manejando en tiempo real las solicitudes y proporcionando la posibilidad de concretar el pago tanto en efectivo como con tarjeta, además, proporcionando una web para un histórico de pedidos y otra para la visualización en tiempo real de estos.

4.1.1.2 Requisitos de usuario

Por otra parte, cuando hablamos de los requisitos de usuarios podemos identificar, basados en las necesidades de nuestra audiencia, separa en dos principales grupos, los cuales son el personal del instituto (incluidos profesores y personal no docente como secretaría) y los alumnos que se espera sean principalmente los pertenecientes a los grados básicos, medios y superiores, sin ningún tipo de discriminación por sexo, género o gustos, pero si por edad en aquellos productos donde existiese una edad mínima legal para el consumo.

4.1.1.3 Requisitos funcionales

Requisito Funcional N.º1 - Sistema de autenticación de usuarios

Debe existir un sistema de autenticación seguro para proteger la información personal y financiera de los usuarios, además deben poder registrarse en la plataforma, permitiendo además crear sus perfiles.

ID	RF_AUTH_1
Descripción	Permitir a los usuarios identificarse y acceder a sus perfiles
Entradas	Nombre de Usuario y Contraseña
Salidas	Permitir acceso a la plataforma
Prioridad	Alta
Excepciones	El usuario no existe, le invita a registrarse

Requisito Funcional N.º2 - Sistema de carrito de compras

Debe existir un sistema que otorgue al usuario la capacidad de gestionar un carrito de compras persistentes con múltiples productos que facilite la gestión y tramitación posterior de productos.

ID	RF_PROD_1
Descripción	Permitir a los usuarios añadir productos al carrito
Entradas	Productos
Salidas	Confirmación del añadido
Prioridad	Alta

Excepciones	Ocurre un error, se notifica al usuario
-------------	---

Requisito Funcional N.º3 - Sistema de tramitación de pedidos no pagados

Debe existir un sistema que permita al usuario usar el **carrito** con los productos deseados que permite gestionar y tramitar un pedido sin pagar permitiendo así la finalización del pedido posteriormente con algún método de pago válido.

ID	RF_PED_1
Descripción	Permitir a los usuarios tramitar pedidos sin pagar
Entradas	Productos
Salidas	Confirmación del pedido
Prioridad	Alta
Excepciones	Ocurre un error, se notifica al usuario

Requisito Funcional N.º4 - Sistema de tramitación de pedidos pagados

Paralelamente al requisito funcional anterior debe existir un sistema de tramitación de pedidos que permita finalizar los mismos con un **pago** de forma tal que el paso posterior sea solamente la recogida/adquisición de los productos, permitiendo al usuario elegir si desea pagar mediante métodos a distancia o no con el sistema de pedidos no pagados.

ID	RF_PED_2
Descripción	Permitir a los usuarios tramitar pedidos pagados
Entradas	Productos, método de pago
Salidas	Confirmación del pedido
Prioridad	Alta
Excepciones	Ocurre un error, se notifica al usuario, no se realiza el cobro ni el pedido.

Requisito Funcional N.º5 - Sistema de notificación de pedidos vía dispositivo

Debe existir un sistema de notificación en vivo y en directo que permita a los dependientes de la cafetería que permita conocer los pedidos tramitados pagos y no pagos para la preparación

de los mismos mediante dispositivos físicos tales como tablets, smartphones, portátiles y/o ordenadores.

ID	RF_PED_3
Descripción	Mandar un webhook con el pedido al servicio de notificación en tiempo real a la tablet de la cafetería, registrar pedido en DB
Entradas	Productos, método de pago
Salidas	Pedido
Prioridad	Alta
Excepciones	Ocurre un error, se registra un log

Requisito Funcional N.º6 - Sistema de finalización de pedidos

Los dependientes de la cafetería deberían ser capaces de finalizar pedidos existentes dando de alta los mismos en la base de datos para generar un historial consistente y persistente de pedidos realizados.

ID	RF_PED_4
Descripción	La tablet de la cafetería marca un pedido como concluido, se registra en DB
Entradas	pedido
Salidas	Mensaje de confirmación
Prioridad	Alta
Excepciones	Ocurre un error, se avisa al usuario

Requisito Funcional N.º7 - Sistema de notificación de pedidos via email

Paralelamente a la notificación mediante un dispositivo físico (tablet, smartphone, portátil) de los pedidos gestionados y tramitados debe existir un sistema de notificaciones via email con los datos concretos del pedido, tal como número de orden, factura/ticket y detalles completos.

ID	RF_MAIL_1
Descripción	Enviar mails tanto al correo de la cafetería como al correo del usuario al momento de efectuar un pedido
Entradas	Usuario, productos, pago
Salidas	Mail de confirmación
Prioridad	Media
Excepciones	Mensaje de error al administrador de correos y usuario final

Requisito Funcional N.º8 - Sistema de gestión de pagos

Por la propia complejidad de la plataforma debe existir un sistema de gestión de métodos de pago (PayPal, Stripe, API de los TPV y puede que, incluso, API bancaria) para el uso de métodos de pago válidos en la plataforma diferente de los presenciales.

ID	RF_PAY_1
Descripción	Realizar pagos mediante una plataforma segura
Entradas	pago
Salidas	Mensaje de confirmación
Prioridad	Alta
Excepciones	Mensaje de error al usuario

Contexto sobre la lista de requisitos funcionales:

Uno de los desafíos principales del proyecto fue mantener la lista de requisitos funcionales “pequeña”, como buena práctica, la cantidad de componentes y módulos para el despliegue de aplicaciones web se recomienda que no sea de una cantidad excesivamente larga, sin embargo, de ser necesario, en posteriores versiones podría ampliarse la lista en caso de que fuese necesario.

4.1.1.4. Requisitos no funcionales

Requisito No Funcional N.º1 - Sistema de gestión de seguridad

La autenticación del usuario debería ser mediante **JWT** con encriptación de datos sensibles cumpliendo con normativas internacionales, nacionales y europeas como la **LOPD**.

ID	RNF_AUTH_1
Descripción	Debe haber una API Rest encargada de la autenticación utilizando JWT y los tokens no deben de ser accesibles, sólo a la hora de la petición
Entradas	Usuario y Contraseña
Salidas	Token JWT
Prioridad	Media
Excepciones	Control de excepciones personalizadas al usuario y administrador

Requisito No Funcional N.º2 - Sistema de gestión de Usabilidad

La aplicación ha de ser capaz de ofrecer a los usuarios finales y a los administradores una interfaz responsive correctamente adaptada a **móviles, tablets, ordenadores y portátiles** en diferentes tamaños con un tiempo de aprendizaje máximo de 30 minutos.

ID	RNF_INFC_1
Descripción	La interfaz debe ser responsive y accesible para personas con discapacidades visuales
Prioridad	Alta

Requisito No Funcional N.º3 - Sistema de escalabilidad horizontal

La plataforma debe tener una arquitectura con la utilización de **API's REST / RESTful y microservicios** que faciliten una escalabilidad y el mantenimiento de la aplicación, por ejemplo, automatizando los backups de la base de datos o aportando documentación técnica completa.

ID	RNF_INFR_1
Descripción	La arquitectura debe estar en microservicios contenedorizados, de tal forma que se pueda escalar una de las partes sin la necesidad de escalar la aplicación concreta
Entradas	Usuario y Contraseña
Salidas	Token JWT
Prioridad	Media

Requisito No Funcional N.º4 - Sistema de integración

Por la propia complejidad de nuestra aplicación debe ser integrable con **websockets para comunicación en tiempo real donde se necesite**.

ID	RNF_INFR_2
Descripción	Integración de websockets donde sea necesaria la comunicación en tiempo real
Prioridad	Alta

ID	RNF_INFR_3
Descripción	Uso de apis externas de pagos
Prioridad	Media

Requisito No Funcional N.º5 - Documentación técnica del proyecto

El propio requisito es bastante descriptivo por el título, incluye este propio documento, se espera, como parte de buenas prácticas, documentar todo referente al proyecto fomentando el entendimiento y la escalabilidad del mismo.

ID	RNF_DOC_1
Descripción	Nuestro código debe estar debidamente comentado usando herramientas de documentación como JSDOC, JavaDoc y nuestras APIS usando Swagger
Prioridad	Baja

4.1.1.5. Priorización y dependencias

Antes de dar con una tabla de prioridades y dependencias, creamos una tabla que lista todos los requisitos funcionales (FN) y los requisitos no funcionales (RNF):

REQUISITOS FUNCIONALES	REQUISITOS NO FUNCIONALES
RF_AUTH_1	RNF_AUTH_1
RF_PROD_1	RNF_INF_C_1
RF_PED_1	RNF_INF_R_1
RF_PED_2	RNF_INF_R_2
RF_PED_3	RNF_INF_R_3
RF_PED_4	RNF_DOC_1
RF_MAIL_1	

RF_PAY_1	
----------	--

Finalmente, con esta tabla, podemos crear una tabla de prioridades y dependencias que nos permita identificar qué elementos o módulos son dependientes uno de otros.

ID	Descripción breve	Prioridad	Dependencia
RF_AUTH_1	Autenticación de usuarios	Alta	Ninguna
RF_PROD_1	Carrito de compras	Alta	RF_AUTH_1
RF_PED_1	Tramitación de pedidos no pagados	Alta	RF_PROD_1
RF_PED_2	Tramitación de pedidos pagados	Alta	RF_PROD_1 RF_PAY_1
RF_PED_3	Notificación de pedidos en dispositivo	Alta	RF_PED_1 RF_PED_2
RD_PED_4	Finalización de pedidos	Alta	RF_PED_1 RF_PED_2
RF_MAIL_1	Notificación de pedidos por email	Media	RF_PED_1 RF_PED_2
RF_PAY_1	Gestión de pagos	Alta	RF_AUTH_1
RNF_AUTH_1	Seguridad y autenticación con JWT	Media	RF_AUTH_1
RNF_INFC_1	Usabilidad y accesibilidad	Alta	Ninguna
RNF_INFR_1	Escalabilidad y microservicios	Media	Ninguna
RNF_INFR_2	Integración con websockets y APIS	Media	RF_PED_3 RF_PAY_1

RNF_DOC_1	Documentación técnica	Baja	Ninguna
-----------	-----------------------	------	---------

Contexto de la tabla

Como podemos apreciar, la base de la aplicación son la autenticación y el carrito de compra ya que son la base para la mayoría de las funcionalidades, por lo que, en teoría, deben implementarse primero, sin embargo, la gestión de pagos y la integración con APIs son fundamentales para la tramitación de pedidos y sus pagos, finalmente las notificaciones dependen de la existencia previa del requisitos de pedidos tramitados y la documentación se debe mantener durante todo el desarrollo aunque su prioridad sea baja.

4.1.2. Metodología Agile

El encabezado está pensado para dar contexto, guía y sentido a toda la información del proyecto relacionada con la aplicación de metodologías ágiles, los orígenes de la misma y nuestra elección propia de la misma.

4.1.2.1. Definición y principios del Agile

La adopción de metodologías ágiles en este TFG responde a la necesidad de iterar rápidamente sobre las funcionalidades principales (registro, autenticación y gestión de datos), obteniendo feedback continuo y ajustando los requisitos sin perder eficiencia. Partiendo de los cuatro valores fundamentales del Manifiesto Ágil (desarrollado en 2001 como alternativa a los modelos de desarrollo y creación de software tradicionales y rígidos hasta el momento) y apoyándome en principios como la entrega frecuente y la autoorganización del equipo (Aitor, nuestra tutora, el personal de la cafetería y mi persona), he planificado un Sprint 0 orientado a sentar las bases del proyecto y un seguimiento diario mediante un Kanban Board.

4.1.2.2. Marco escogido: Kanban

Para gestionar el desarrollo de este TFG hemos optado por Kanban board, ya que permite a equipos de cualquier tamaño visualizar el flujo de trabajo de forma continua, limitar el trabajo en curso y ajustar prioridades en tiempo real, para ello utilizaremos la tabla de **Kanban** proporcionada por **Mercedes** en el primer año de **DAW**, se puede acceder a la tabla completa en **Google sheets** desde el siguiente enlace “ Kanban board TFG (DAW 2025)” aunque finalmente también se inserta la tabla en el documento.

Este apartado se ha dedicado expresamente a dar contexto sobre la metodología ágil sirviendo como una ayuda a quienes leen el documento del **TFG** para comprender primero el “marco de trabajo” y luego los elementos gestionados con él, por lo tanto, vamos a desglosar el contenido de la **KANBAN BOARD** en un formato amigable en los siguientes puntos:

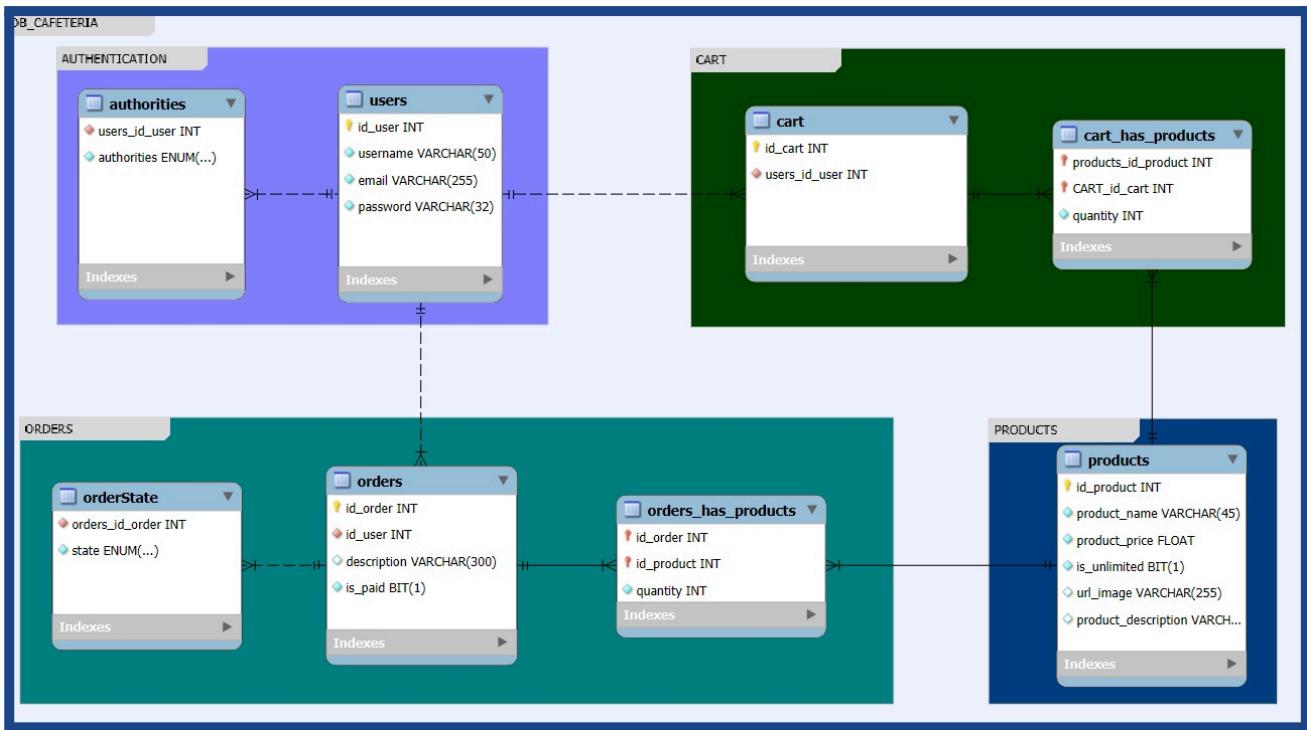
- **Flujo: columnas (Backlog, por hacer, en progreso, pruebas, finalizado):** Son las columnas que definen el flujo de trabajo, el **backlog** es donde se expresan todas las tareas/actividades que se necesitan hacer en el tablero, luego viene el **por hacer** que son las actividades aún no iniciadas, **en progreso** significa que actualmente se está trabajando en ello, **pruebas** es la fase donde intentamos con pruebas de caja negra o blanca probar todo y se marca una actividad como **finalizada** una vez se supera la fase de pruebas.

Tipo	Rol	Funcionalidad o tarea	Razón	Prioridad	Puntos	Días	Detalles
Backlog							
Funcionalidad	Team	Customize Headings and Card Types	Edit the Legend in the Type column as needed.	Alta	5		Look at the conditional formatting rules to see how the color-coding works.
Contenido	All	Add ideas to the backlog	The backlog is where you stick the to-dos that you might work on later.	Media	2		Some ideas may not lead to deliverables, but time may need to be allocated to researching them.
Actualización	Who For	Be specific	User Story: As a [Role] I want [Feature] so that [Reason]	Baja	6		
Tarea		Articles, White Papers, Documentation		Medium	3		We use the Content type for writing blog posts, preparing marketing materials, and support content.

- **Límites WIP (Work-In-Progress):** Esto se refiere al límite (capacidad) de trabajo real que podemos enfrentar en la metodología ágil, en nuestro caso hemos establecido (siguiendo estándares) en 3, esto quiere decir que en la columna de flujo llamada “en progreso” habrá un máximo simultáneo de 3 actividades en progreso.
- **Cadencia de revisión (reunión de seguimiento o Replenishment):** Por la propia complejidad del proyecto hemos ideado 3 formas de hacer seguimiento y revisión del proyecto, la primera es la **Replenishment Meeting** que es una reunión informal que tenemos un día a la semana (en el que nuestros horarios concuerden) para conversar sobre el curso actual del proyecto, adicionalmente, siguiendo la metodología ágil de kanban, una reunión en cada **Sprint** del proyecto.

4.1.2.3. Sprint 0 y Kanban

- **Definición y validación de los requisitos iniciales (funcionales y no funcionales):** Esta fase la vivimos ya en el análisis de requisitos, es importante durante todas las fases del desarrollo de software mantener a los usuarios y al personal de la cafetería presentes para que el diseño sea adecuado, damos por válidos los requisitos que establecimos previamente.
- **Configuración del repositorio de código y establecimiento del flujo de trabajo con Git:** En esta fase **Aitor** se encargó de la creación del repositorio así como de la organización de las ramas, el estándar de commits, la forma de mergear los cambios y organizar el flujo de trabajo en general.
- **Selección y configuración de las herramientas de desarrollo (IDE, frameworks, dependencias, Docker, etc.):** Para mantener un entorno de desarrollo moderno con tecnologías profesionales ampliamente usadas y reconocidas en entornos de trabajo actuales elegimos tecnologías como **Nestjs, React, Astro, Spring, Spring Security, JWT, docker, etc**, para diseñar y desarrollar hemos elegido como **IDE** Visual Studio Code e IntelliJ que permiten una gran personalización a través de plugins y extensiones.
- **Diseño preliminar de la arquitectura (microservicios, base de datos, comunicación entre módulos):** En esta fase se elaboró un diagrama que explica visualmente la relación entre entidades, roles, operaciones, red interna, externa, etc, este diagrama es del tipo **E/R**.



- **Creación del tablero Kanban para la gestión visual de tareas y seguimiento del avance:** Utilizando la plantilla proporcionada por **Mercedes** el año anterior en **entornos de desarrollo** adaptamos y creamos un tablero kanban a nuestro idioma y preferencias sin abandonar los estándares del mismo, accesible desde el hiperenlace [+ Kanban board TFG \(DAW 2025\)](#).

Tablero Kanban		Fecha de inicio del Sprint	Días	Progreso	Progreso (representado en banderas)		
		18/3/2025	120	18,2%			
Tipo	Rol	Funcionalidad o tarea	Razón	Prioridad	Puntos	Días	Detalles
Backlog							
Funcionalidad	Team	Customize Headings and Card Types	Edit the Legend in the Type column as needed.	Alta	5		Look at the conditional formatting rules to see how the color-coding works.
Contenido	All	Add ideas to the backlog	The backlog is where you stick the to-dos that you might work on later.	Media	2		Some ideas may not lead to deliverables, but time may need to be allocated to researching them.
Actualización	Who For	Be specific	User Story: As a [Role] I want [Feature] so that [Reason]	Baja	6		
Tarea		Articles, White Papers, Documentation		Medium	3		We use the Content type for writing blog posts, preparing marketing materials, and support content.
Research	Tutor	Task 3: Identification of experts (Preparatory work for the workshop of experts to exchange information)	To have time enough to contact them and organise the agenda of the workshop well in advance	Medium	58		The experts are/ will be mainly identified by the Tutor
Research	Tutor	Task 4: Selection/ availability of a meeting room (Preparatory work for the workshop of experts)	To get the reservation of the meeting room for the date of interest	Medium	58		The booking has to be arranged by the Tutor

- **Identificación de los primeros entregables mínimos viables (MVP):** En esta fase definieron los primeros entregables que permitirían tener una versión funcional básica del sistema (MVP). Entre ellos se incluyeron:

1. Un sistema de autenticación básico
2. Un frontend mínimo para realizar pedidos

3. Un backend capaz de registrar y mostrar pedidos
4. Comunicación básica entre frontend y backend
5. Estos entregables permiten validar la viabilidad técnica y funcional del proyecto desde las primeras etapas, facilitando la detección temprana de problemas y la obtención de feedback.

4.1.3. Historias de usuario

Las historias de usuario son un recurso que también aprendimos en entornos de desarrollo en **DAWº1**, podemos darnos una idea de que es si desde el siguiente enlace revisamos la [Guia de contenido en Entornos de Desarrollo](#) un documento con ejemplos el cual podemos consultar desde el documento alojado en Google Drive el siguiente enlace: [Historias de usuario con ejemplos y plantilla.pdf](#), basado en esto dimos con las siguientes historias de usuario:

4.1.3.1. Historias de usuario como alumno/cliente

- **HU1:** Como alumno, quiero poder registrarme y acceder a la plataforma para poder realizar pedidos en la cafetería de forma personalizada y segura.
- **HU2:** Como usuario, quiero consultar el menú de productos disponible para elegir lo que deseo pedir.
- **HU3:** Como usuario, quiero añadir productos a un carrito para gestionar mi pedido antes de confirmarlo.
- **HU4:** Como usuario, quiero tramitar un pedido sin necesidad de pagarlo en el momento, para poder abonar presencialmente al recogerlo.
- **HU5:** Como usuario, quiero poder pagar mi pedido online (tarjeta, PayPal, etc.) para agilizar la recogida y evitar colas.
- **HU6:** Como usuario, quiero recibir una confirmación y el estado de mi pedido en tiempo real para saber cuándo está listo para recoger.
- **HU7:** Como usuario, quiero recibir un correo electrónico con el resumen y comprobante de mi pedido para tener constancia de la compra.
- **HU8:** Como usuario, quiero consultar el histórico de mis pedidos para revisar lo que he comprado anteriormente.

4.1.3.2. Historias de usuario como personal de cafetería

- **HU9:** Como dependiente, quiero acceder a una aplicación interna donde pueda ver en tiempo real los pedidos que llegan para poder prepararlos rápidamente.
- **HU10:** Como dependiente, quiero marcar los pedidos como preparados/concluidos para que el sistema notifique al cliente y quede registrado en el histórico.
- **HU11:** Como dependiente, quiero recibir notificaciones en la tablet/PC de la cafetería

cuando se realice un nuevo pedido para no perder ninguna solicitud.

- **HU12:** Como dependiente, quiero poder consultar el histórico de pedidos para llevar un control de la actividad diaria.

4.1.3.3. Historias de usuario como administrador

- **HU13:** Como administrador, quiero gestionar el catálogo de productos (añadir, modificar, eliminar) para mantener el menú actualizado.
- **HU14:** Como administrador, quiero consultar estadísticas de ventas y pedidos para tomar decisiones informadas sobre la gestión de la cafetería.
- **HU15:** Como administrador, quiero gestionar los roles y permisos de los usuarios para garantizar la seguridad y el correcto funcionamiento del sistema.

4.1.3.4. Historias de usuario como sistema

- **HU16:** Como sistema, debe integrar pasarelas de pago externas para permitir pagos online de forma segura.
- **HU17:** Como sistema, debe enviarse notificaciones por email y en tiempo real para mantener informados a usuarios y personal.
- **HU18:** Como sistema, debe garantizar la seguridad de los datos personales y financieros de los usuarios mediante autenticación y cifrado.

4.1.4. Backlog inicial

Ahora que tenemos un análisis de requisitos principal, inicial, rico, complejo, pero fácil y entendible y tenemos las historias de usuario sí que podemos dar lugar a un **backlog** inicial que poder emplear en nuestro flujo de trabajo y tablero kanban.

ID	Historia/Tarea	Prioridad	Módulo/Servicio	Estado inicial
HU1	Registro y autenticación de usuarios	Alta	authentication-service web-pedidos	Por hacer
HU2	Consulta de menú de productos	Alta	pedidos-service web-pedidos	Por hacer
HU3	Añadir productos al carrito	Alta	pedidos-service	Por hacer

			web-pedidos	
HU4	Tramitación de pedidos no pagados	Alta	pedidos-service web-pedidos websocket-service cafetería-app	Por hacer
HU5	Tramitación de pedidos apagados (integración de pagos)	Alta	pedidos-service web-pedidos	Por hacer
HU6	Notificación en tiempo real de nuevos pedidos al personal	Alta	pedidos-service websocket-service cafetería-app	Por hacer
HU7	Confirmación y estado del pedido para el usuario	Alta	websocket-service cafetería app pedidos-service web-pedidos	Por hacer
HU8	Envío de email de confirmación de pedido	Baja	order-service mail-service	Por hacer
HU9	Visualización de pedidos en tiempo real en la app de la cafetería	Alta	websocket-service cafetería-app	Por hacer
HU10	Marcar pedidos como preparados/concluidos	Alta	pedidos-service cafetería-app	Por hacer
HU11	Gestión del catálogo de productos (CRUD)	Media	pedidos-service admin-frontend	Por hacer
HU12	Consulta de histórico de pedidos (usuario y personal)	Media	pedidos-service frontend	Por hacer

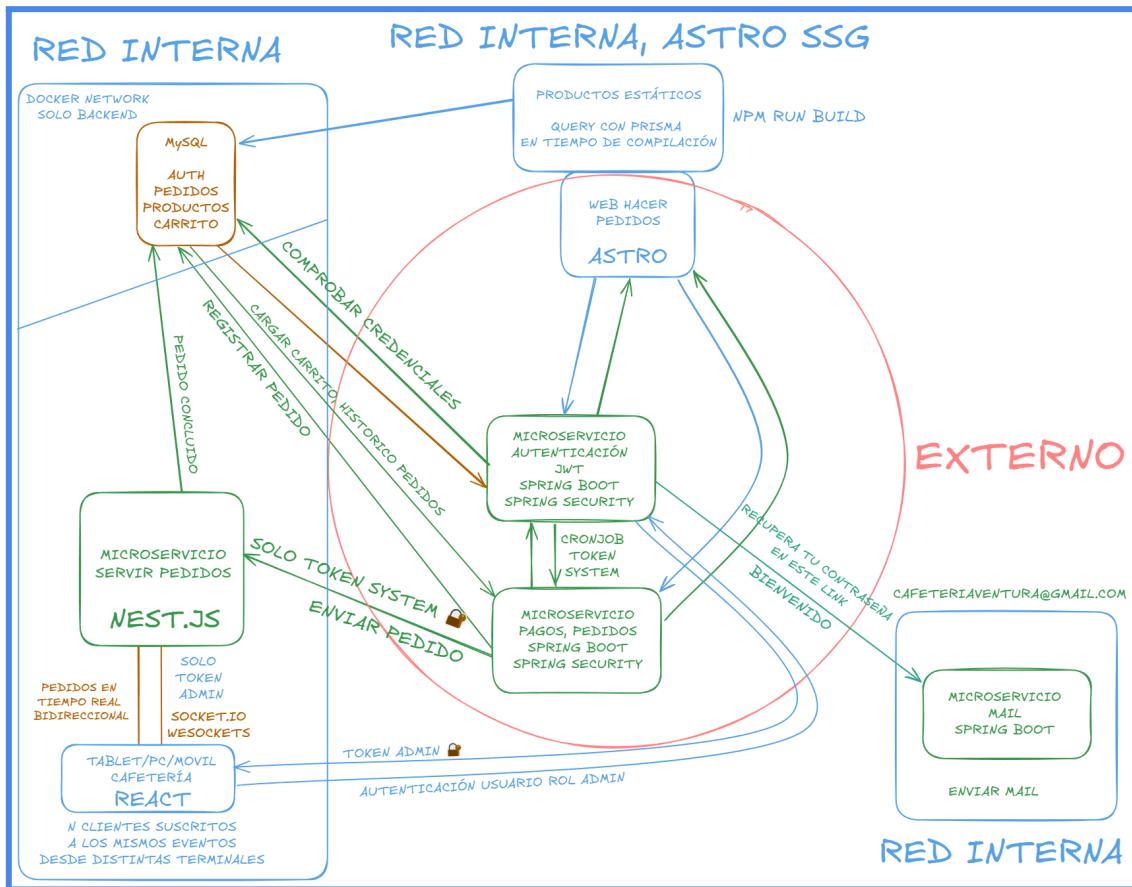
HU1 3	Gestión de roles y permisos	Media	authentication-service admin	Por hacer
HU1 4	Estadísticas y reportes para administración	Baja	pedidos-service admin frontend	Por hacer
RNF1	Seguridad (autenticación JWT y cifrado de datos)	Alta	authetication-service	Por hacer
RNF2	Interfaz responsive y accesible	Alta	cafeteria-app frontend	Por hacer
RNF3	Arquitectura de microservicios y contenedores docker	Alta	Todos	Por hacer
RNF4	Integración de WebSockets y APIs externas de pago	Alta	websocket-service pagos-service	Por hacer
RNF5	Documentación técnica y de usuario	Media	todos	Por hacer

4.2. Herramientas gráficas de análisis (para dar visión global)

Hemos elegido herramientas para esta fase del proyecto, alguna de ellas mostradas en el transcurso del ciclo de formación así como con las que hemos dado investigación en internet, entre ellas **MySQL Workbench**, **mermaid chart/live**, **etc**, presentando los siguientes diagramas.

4.2.1. Diagrama de entidades

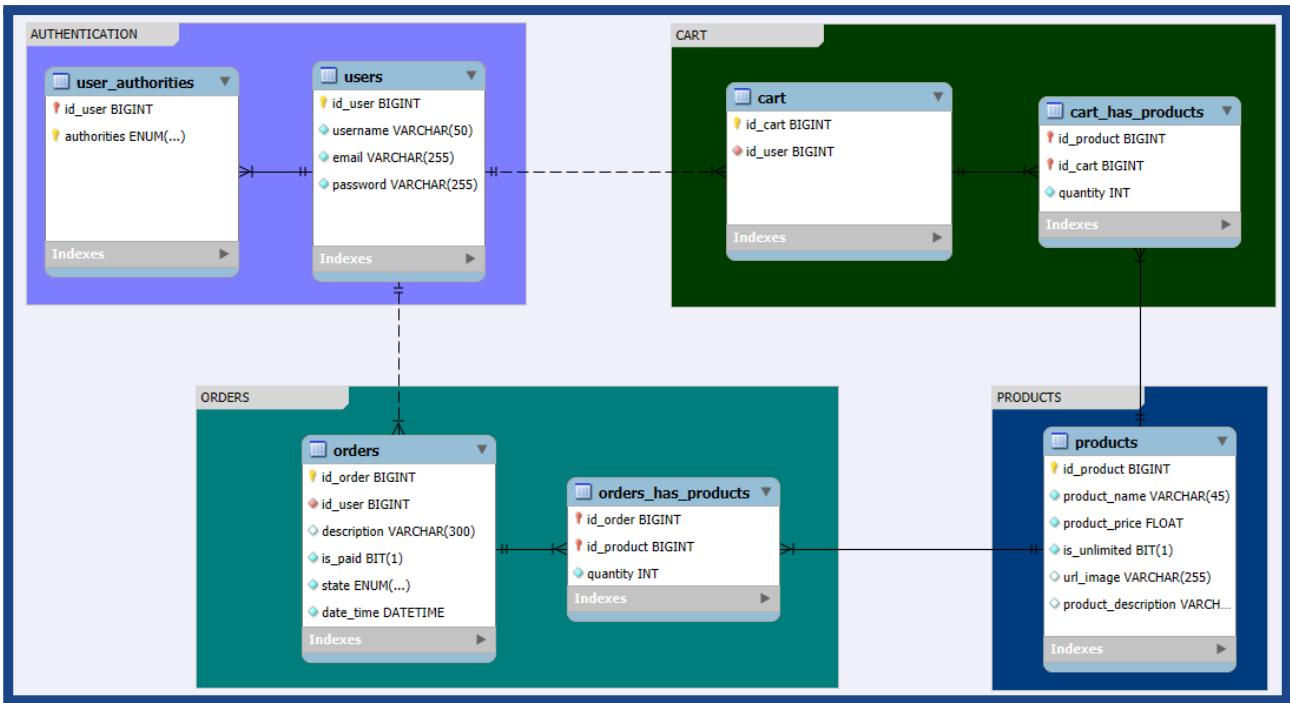
El diagrama de flujo de datos fue generado tomando en cuenta la red interna y externa del proyecto así como un apartado para lo “no controlado” usando la web de **mermaid chart**.



4.2.2. Diagramas Entidad-Relación (E/R)

4.2.3. Modelo de datos

En nuestro caso se ha generado utilizando **MySQL workbench**, la imagen se actualiza en cada cambio realizado, siendo la última versión del diagrama de modelo de datos::



Se puede apreciar la imagen del diagrama en su versión más reciente en el siguiente hiperenlace donde está alojada en este mismo google drive [Diagrama de entidad relacion.png](#), en el podemos apreciar 4 sectores principales los cuales son:

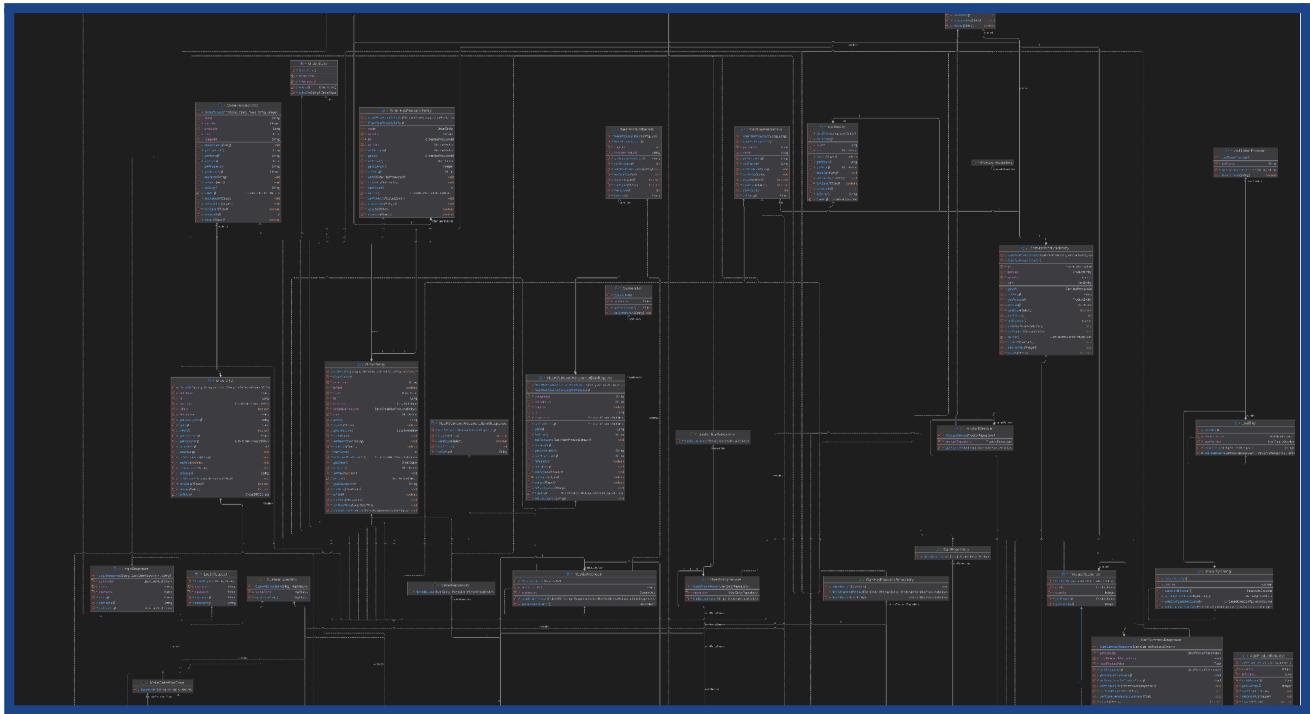
- **Autenticación:** Donde existen las tablas de usuarios y autoridades y se maneja la autenticación y validación de usuarios.
- **Carrito:** Todos los usuarios tienen un carrito propio intransferible el cual es capaz de almacenar con una entidad todos los productos que tiene “X” carrito facilitando la gestión de órdenes (pedidos).
- **Órdenes:** Todas las órdenes están asociadas a un usuario, las órdenes se hacen a los productos existentes dentro de un carrito, además, tienen un “estado” que puede marcarlas como “en curso”, “pendientes”, “finalizadas” o cualquier nomenclatura que se elija finalmente en el proyecto.
- **Productos:** Los productos de la cafetería existen en su propia tabla y sector, un carrito puede tener entre cero y varios productos y las órdenes iteran sobre los productos individualmente para concretar una orden.

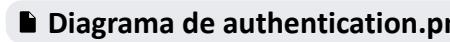
Esto finalmente nos da lugar a una tabla de base de datos inicial que en posteriores encabezados del documento serán sometidas a los procesos de normalización y estandarización de la misma, hasta entonces, queda como constancia de nuestro modelo de datos planteado inicialmente en el proyecto.

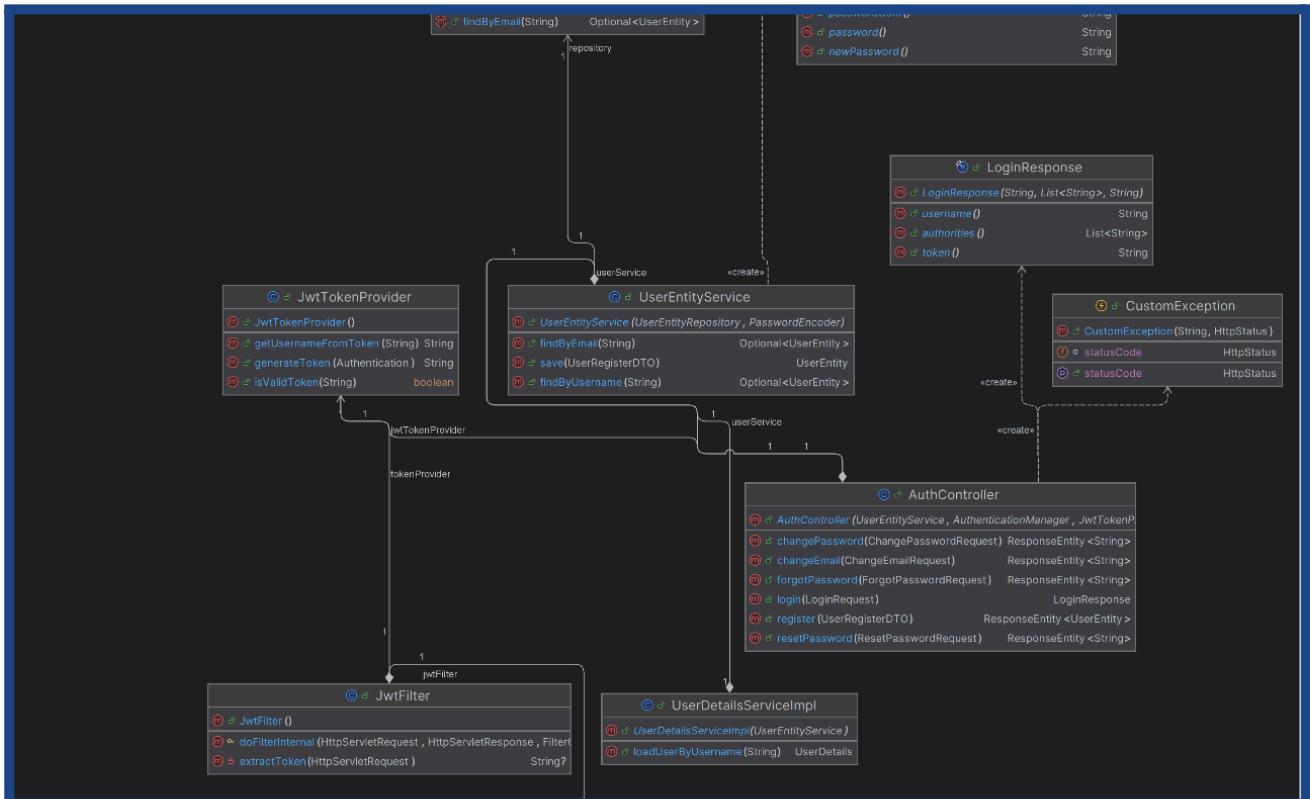
4.2.4. Diagramas de clase

Los diagramas de clase en nuestro caso suponen una dificultad considerable ya que nuestro proyecto por su propia complejidad y las tecnologías utilizadas cuenta con sistemas complejos aunque eficientes y robustos de clases, están disponibles para su visualización en Google drive los siguientes diagramas de clases:

- Diagrama de order-service:  diagrama de order-service.png.



- Diagrama de authentication:  Diagrama de authentication.png.



4.2.5. Casos de uso y tabla requisitos

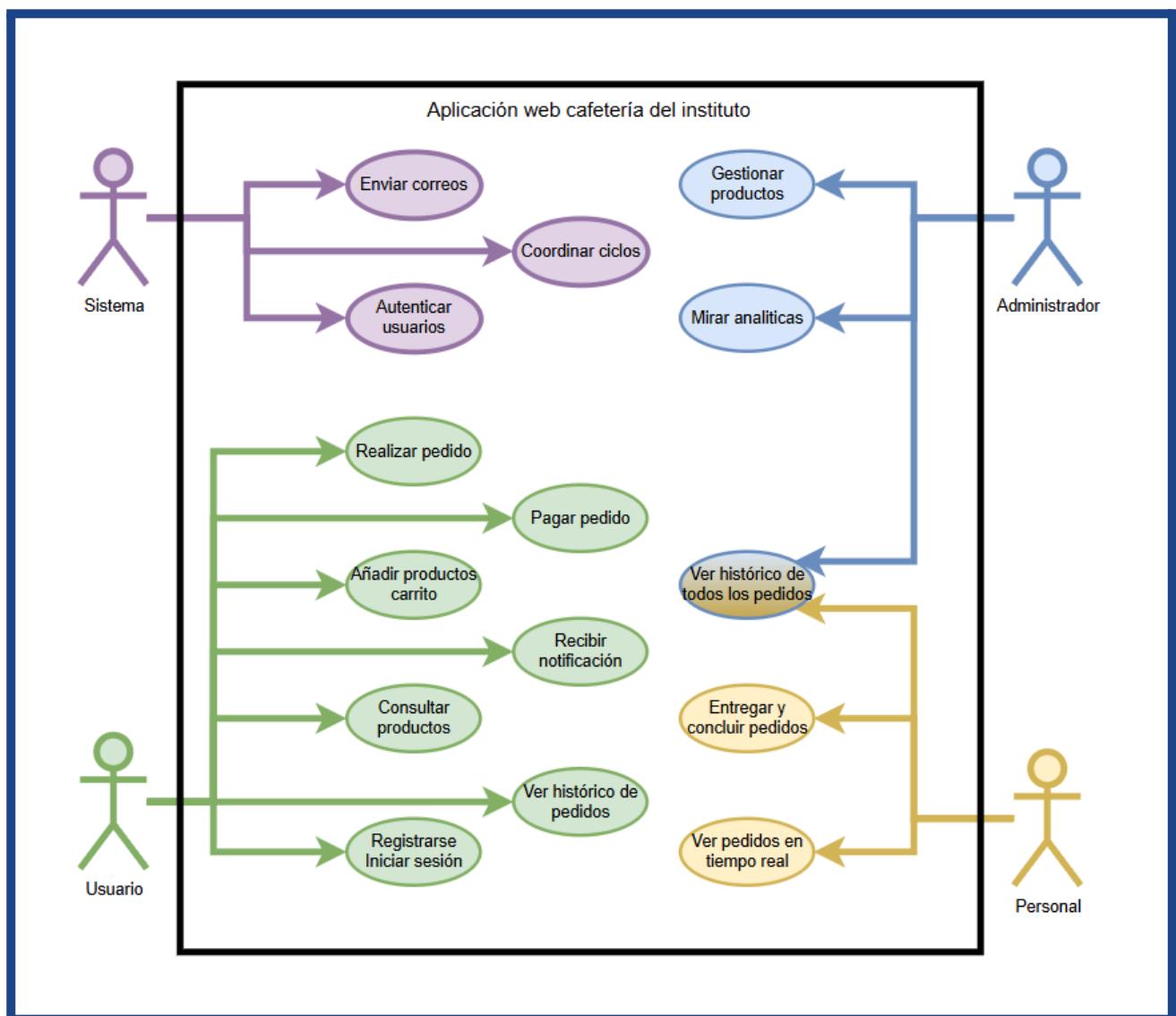
Ahora que tenemos historias de usuarios, una base de datos inicial, requisitos funcionales, no funcionales, de usuario, de negocio y una idea más o menos establecida del proyecto vamos a plasmar nuestra primera ronda de casos de uso en la siguiente tabla:

ID	Nombre	Actor	Breve descripción
CU1	Registrarse Iniciar sesión	Alumno / Cliente	El usuario se registra y/o inicia sesión en la plataforma.
CU2	Consultar productos	Alumno / Cliente	El usuario visualiza los productos disponibles
CU3	Añadir productos al carrito	Alumno / Cliente	El usuario añade productos al carrito
CU4	Realizar pedido	Alumno / Cliente	El usuario tramita un pedido (pagado o no pagado)
CU5	Pagar pedido online	Alumno / Cliente	El usuario paga el pedido mediante una pasarela de pago.
CU6	Recibir notificación de pedido	Alumno / Cliente	El usuario recibe notificaciones sobre el estado de su pedido.
CU7	Ver histórico de pedidos	Alumno / Cliente	El usuario consulta sus pedidos anteriores en el tiempo.
CU8	Ver pedidos en tiempo real	Personal / Cafetería	El personal visualiza los pedidos en tiempo real.
CU9	Marcar pedido como preparado	Personal / Cafetería	El personal entrega y concluye un pedido.
CU10	Ver histórico de pedidos	Personal / Cafetería / Admin	El personal o admin puede ver el histórico de todos los pedidos de todos los usuarios
CU11	Gestionar productos	Administrador	El admin puede añadir, modificar o eliminar productos (CRUD)
CU12	Consultar estadísticas	Administrador	El admin consulta estadísticas de ventas y pedidos.

CU1 3	Enviar correos	Sistema	El sistema puede enviar correos de confirmación, de cambio de contraseña y de pedidos
CU1 4	Valida datos	Sistema	El sistema puede validar la autenticidad de los usuarios
CU1 5	Coordinar ciclos	Sistema	El sistema puede sincronizar la información en tiempo real

4.2.6. Diagrama de casos de uso

La simplicidad del diagrama de casos de uso comparado con la enorme complejidad del backend solo refleja lo diverso y enriquecedor del sistema aportando soluciones profesionales, seguras, abstractas, escalables y originales a problemas del día a día, dando como resultado al siguiente diagrama de casos de uso:



4.3. Planificación de realización del proyecto (cronograma genérico)

5º. - Diseño: ¿Cómo se hará la aplicación?

5.2. Diseño de la base de datos

Cuando hablamos del **diseño profesional** de bases de datos estos nos lleva a recordar el patrón y metodología de diseño aprendido en el primer año de **DAW** el cual consiste de los siguientes pasos:

Identificar las posibles entidades.

El **usuario** debe ser capaz de realizar **pedidos** a la cafetería, los **pedidos** pueden tener uno o varios **productos** los cuales previamente a ser pagados deben pasar por un **carrito** que permite consultar, modificar y/o finalizar los pedidos realizando un pago, además del **usuario** común debe haber un **usuario** para los empleados de la cafetería y/o en cualquier caso un **usuario** administrador encargado de la gestión total de la aplicación, debe haber un historico de **pedidos** constante con el paso del tiempo, resaltando en negro todo lo que parecen ser entidades quedamos con una lista inicial de entidades la cual es:

- **users (usuarios)**
- **orders (pedidos)**
- **cart (carritos)**
- **products (productos)**
- **user authorities (rol de usuarios)**
- **cart_has_products (carrito tiene productos)**
- **order_has_productos (pedidos tienen productos)**

Identificar las posibles interrelaciones.

Las interrelaciones son los términos utilizados para aclarar y establecer las relaciones entre las entidades, por ejemplo, un **usuario TIENE carrito**, un **carrito TIENE productos**, el **usuario REALIZA pedidos**, el **usuario POSEE roles**, etc, por lo tanto, hemos dado con la siguiente lista de interrelaciones:

- **Posee**
- **Tienen.**
- **Tiene.**
- **Pertenece.**
- **Participa.**
- **Define.**

- **Aparece en.**
- **Realiza.**
- **Gestiona.**
- **Registra**

Identificar las posibles entidades-relaciones.

Gracias al análisis de requisitos y los pasos anteriores podemos establecer las entidades-relaciones de forma profesional, recordando que las tablas pivote/auxiliares definidas anteriormente con “**has**” queda sobreentendida su existencia sin definir una entidad-relación completa que es subjetiva, quedando de la siguiente forma:

- **users → tiene → user_authorities.**
- **users → tienen → cart.**
- **users → realiza → orders.**
- **users → registra → products.**
- **cart → posse → products.**
- **cart → pertenece a → users.**
- **products → participa en → orders.**
- **products → aparece en → cart.**

Identificar los atributos.

Una vez establecido todas las entidad-relaciones procedemos a definir los atributos de las entidades basado en el análisis de requisitos para poder registrar y gestionar toda la información necesaria para la aplicación, quedando así de la siguiente forma:

Atributos de users:

- **(PK) id_user (BIGINT):** Clave primaria de los usuarios, identificador único.
- **username (VARCHAR 50):** Nombre de usuario **único** e irrepetible del usuario.
- **email (VARCHAR 255):** Correo electrónico del usuario **único** e irrepetible del usuario.
- **password (VARCHAR 255):** Contraseña (cifrada) del usuario, cifrada en **BCRYPT**.

Atributos de userAuthorities:

- **(PK) id_user (BIGINT):** Clave primaria de la tabla de roles y **FK** de la tabla de **users**.
- **authorities (ENUM → Admin, System, User):** Define la autoridad/rol de “X” usuario.

Atributos de cart:

- **(PK) id_cart (BIGINT)**: Identificador único e irrepetible del carrito.
- **(FK) id_user (BIGINT)**: Clave foránea que asocia “X” carrito a un usuario específico.

Atributos de products:

- **(PK) id_product (BIGINT)**: Clave primaria que identifica todos los productos.
- **product_name (VARCHAR 45)**: Nombre del producto.
- **product_price (FLOAT)**: Precio del producto (permite decimales y enteros).
- **is_unlimited (BIT 1)**: Bit que permite marcar si un producto es ilimitado o limitado.
- **url_image (VARCHAR 255)**: URL de la ruta relativa de la imagen en el directorio de public.
- **product_description (VARCHAR 255)**: Descripción del producto.

Atributos de orders:

- **(PK) id_order (BIGINT)**: Clave primaria que identifica todas las órdenes una sobre otra.
- **(FK) id_user (BIGINT)**: Clave foránea que asocia “X” orden a “X” pedido.
- **description (VARCHAR 300)**: Descripción de la orden específica.
- **is_paid (BIT 1)**: Bit que permite marcar si un pedido ha sido pagado o no.
- **state (ENUM → Pendiente, Finalizado)**: Enum que permite diferenciar si está pendiente o finalizado, es un enum por si pudiesen haber más estados en el futuro.
- **date_time (DATETIME)**: Fecha y hora exacta de la realización de un pedido.

Atributos de cart_has_products:

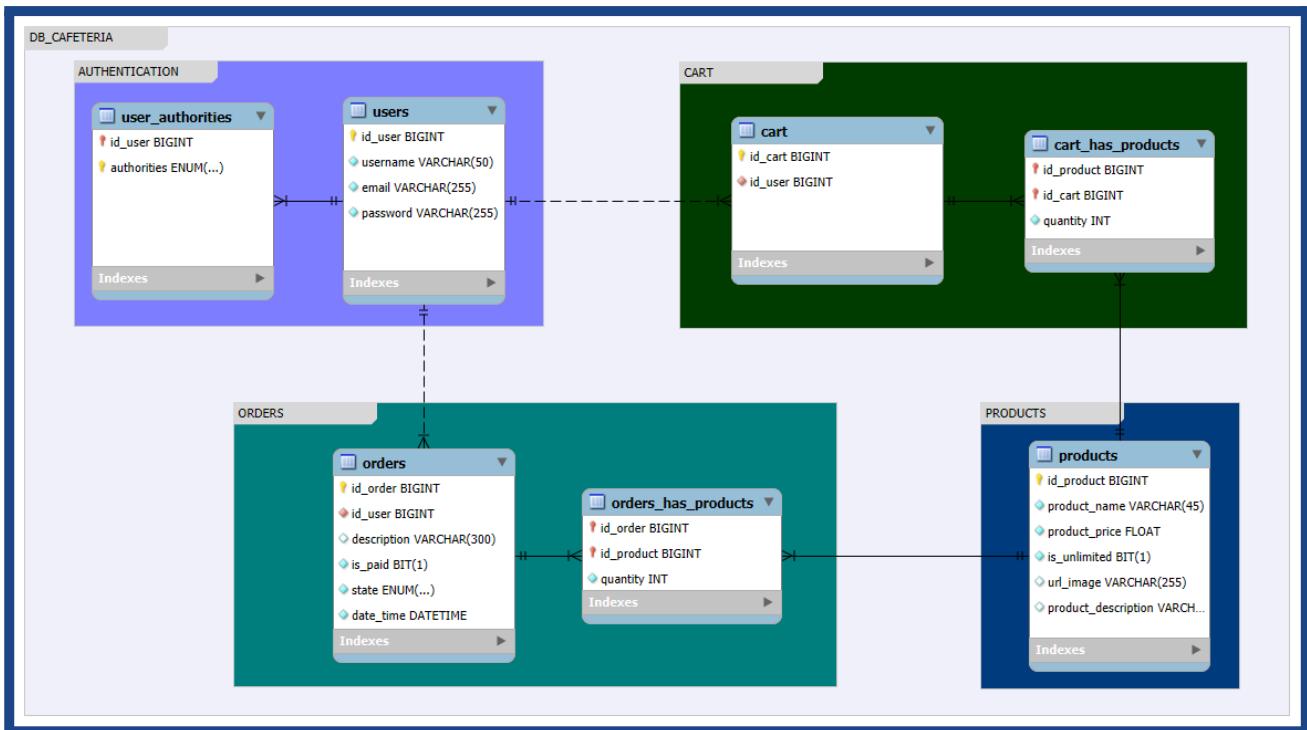
- **(PK) id_product (BIGINT)**: Clave primaria foránea del producto específico de “X” carrito.
- **(PK) id_cart (BIGINT)**: Clave primaria foránea del carrito donde aparece “X” producto.
- **quantity (INT)**: La cantidad exacta de ese “X” producto en ese “X” carrito.

Atributos de orders_has_productos:

- **(PK) id_order (BIGINT)**: Clave primaria foránea del pedido específico de “X” producto.
- **(PK) id_product (BIGINT)**: Clave primaria foránea del producto donde aparece “X” pedido.
- **quantity (INT)**: La cantidad exacta de ese “X” producto en ese “X” carrito.

Desarrollo práctico.

Llegados a este paso, utilizando **MySQL Workbench** nos logramos montar una base de datos profesional normalizada en **FNBC** por naturaleza, se adjunta el diagrama real y actualizado en la siguiente imagen:



Normalización de la base de datos:

Nuestro esquema y base de datos cumple **FN1, FN2, FN3 y FNBC**, esto lo sabemos porque antes de intentar un proceso de normalización comprobamos los requisitos que suelen ser delatores y chivatos de la falta de normalización, donde llegamos a la siguiente conclusión:

- **FN1:** Todos los valores son atómicos (no hay grupos repetitivos)
- **FN2:** Está en FN1 y no hay dependencias parciales (todos los atributos dependen completamente de la clave primaria id_user)
- **FN3:** Está en FN2 y no hay dependencias transitivas (no hay atributos que dependen de otros atributos no clave)
- **FNBC:** Está en FN3 y todas las dependencias funcionales determinan superclaves (cada dependencia tiene como determinante una clave candidata)

Es más, se puede identificar en el diagrama a simple vista que la “llave del castillo” directa e indirectamente que permite atravesar todas las habitaciones es la clave primaria de los usuarios, típico escenario de e-commerce pequeños.

5.3. Diseño de la interfaz de usuario (wireframes, mockups)

Wireframes

En esta parte del proyecto hemos elegido como tecnología para la elaboración de wireframes una plataforma online gratuita llamada Visibly que permite crear wireframes de forma agil, sencilla, completamente gratuita y con docenas de plantillas prediseñadas disponibles, a nivel de front tenemos que recordar que existen dos de ellos, uno es del lado cliente/usuario que adquiere servicios/productos de la cafetería y otro del personal que administra y gestiona la cafetería, por eso tenemos dos conjuntos de wireframes que son los siguientes:

frontend - cliente

El diseño del frontend del lado cliente/consumidor se hizo con especial cautela, astro permite la generación de sitios web estáticos super rápidos y potentes con arquitectura de islas por lo que el dinamismo debe ser el mínimo posible, dando lugar a los siguientes wireframes limpios, profesionales, con el menor nivel de dinamismo posible:

TopMenu (header):

Es uno de los componentes más importantes del frontend del consumidor, maneja toda la lógica de autenticación (inicio de sesión, registro, cierre de sesión) con componentes react que agregan el dinamismo mínimo necesario para el funcionamiento estático potencial de Astro, se observa el siguiente wireframe con los dos escenarios principales, el estado sin iniciar sesión y el estado de sesión iniciada:

The wireframe shows two horizontal navigation bars for 'Cafeteria Ventura Rodriguez'. The top bar represents the state 'session not started' and includes a user icon, 'Usuario', and links for 'Inicio', 'Carrito', and 'Pedidos'. The bottom bar represents the state 'session started' and includes a user icon, 'Iniciar Sesión' (highlighted in dark blue), 'Registrarse', and 'Inicio'.

Cualquiera de las opciones de autenticación (iniciar sesión o registrarse) comparten un modal cuya única diferencia es, dependiendo de la opción elegida, mostrar un campo adicional (repetir contraseña) o un mensaje adicional de “recuperar contraseña” que cambia a un tercer modal (el formulario de recuperar contraseña).

Footer(footer):

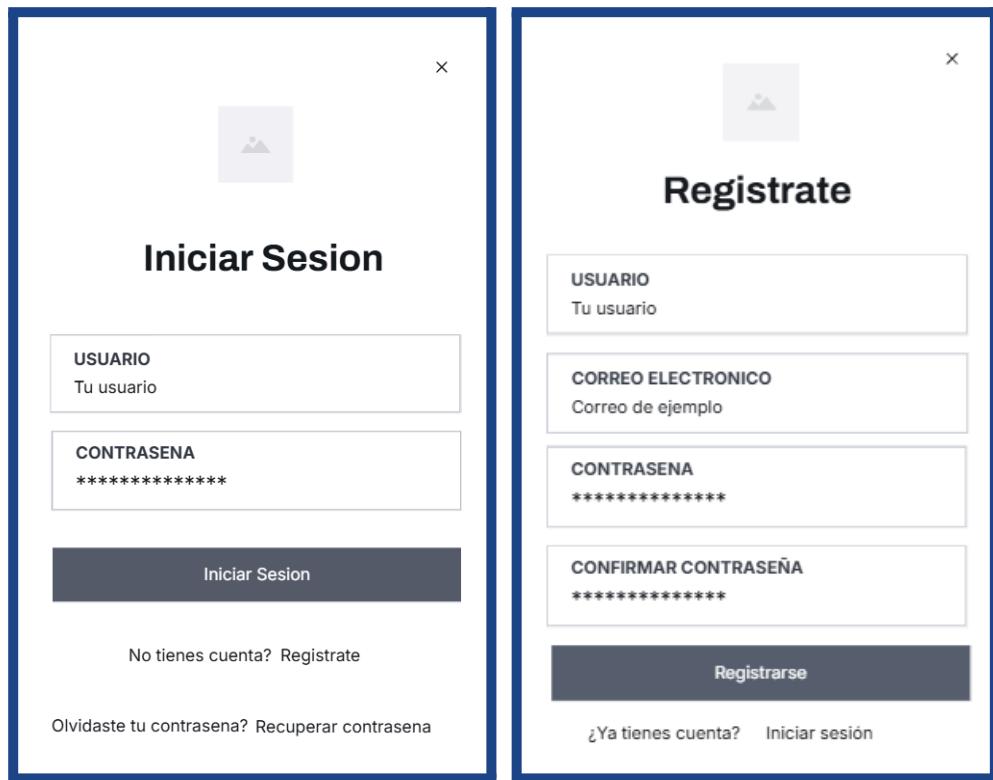
El componente global del pie de página no tiene un nombre extrovertido como TopMenu porque decidimos simplificar los nombres, tiene la siguiente estructura de texto básico estático con botones de CSS nativos:

The footer wireframe displays a layout with a logo, social media icons, and legal links. It includes sections for 'Inicio', 'Carrito', 'Pedidos', contact information ('Contacto: cafeteriaventurajose@gmail.com', 'Dirección: C. Severo Ochoa, 4, 28660 Boadilla del Monte, Madrid'), and legal links ('Aviso legal', 'Política de privacidad'). A copyright notice at the bottom states '© 2025 Cafeteria Ventura Rodriguez. Todos los derechos reservados.'

Los enlaces de carrito y pedidos son visibles y accesibles pero incontrolables sin la sesión iniciada como una buena práctica de SEO que permite a google mapear rápidamente las vistas/páginas principales de un sitio web así como la asociación de la misma a las redes sociales, en este caso, del instituto.

Modales de autenticación:

Los modales de autenticación están completamente renderizados cuando se carga el sitio web por primera vez, permanecen ocultos utilizando display none hasta que son "llamados" desde el componente global de TopMenu desde donde aparecen y se puede intercambiar cual de ellos se muestra a través de propiedades CSS nativas, a continuación se muestra el wireframe de los dos modales principales de autenticación y el tercero de recuperación de contraseñas:



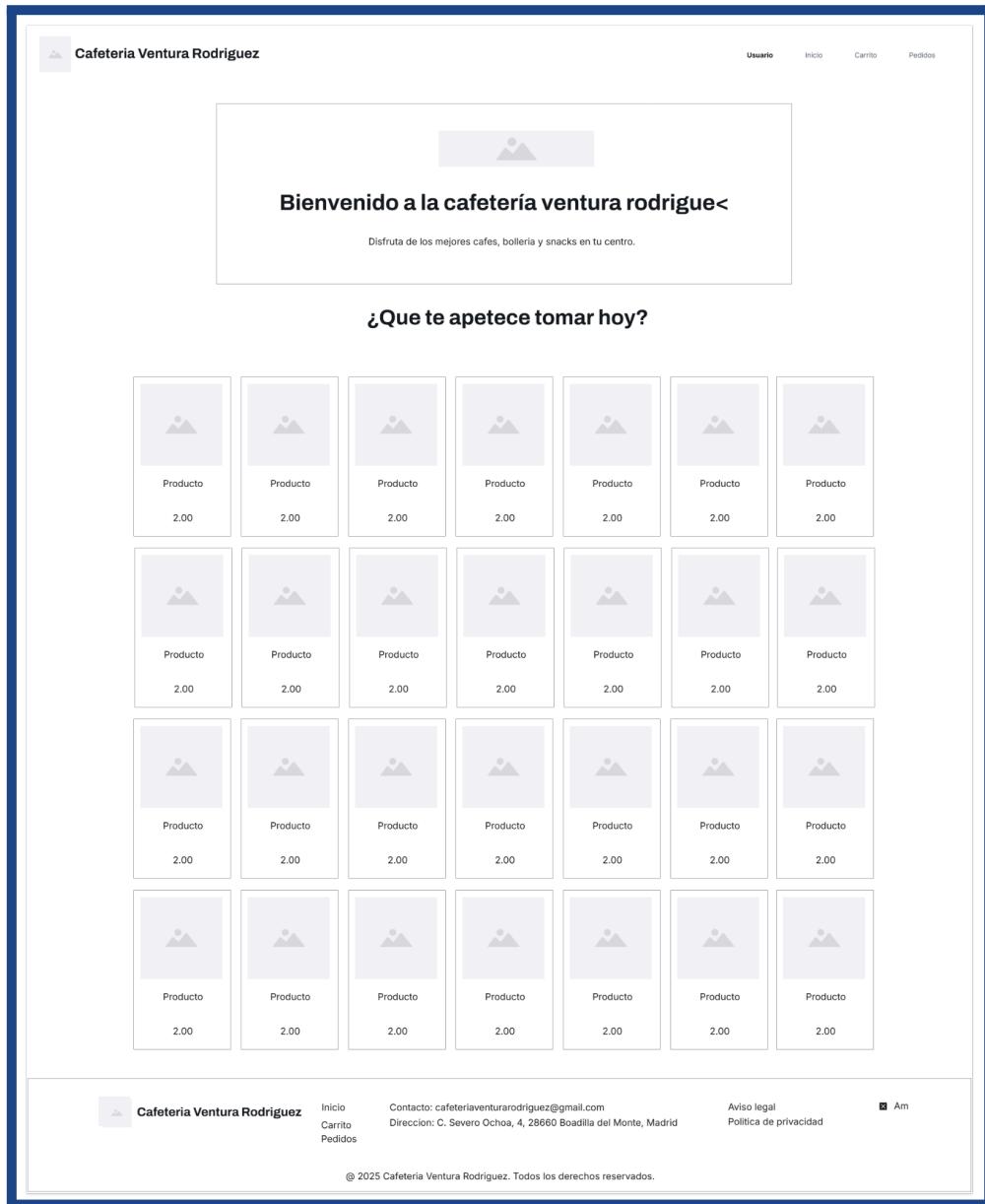
The wireframe shows two side-by-side modal windows. Both have a dark blue header bar with a close button (X) and a small user icon. The left window is titled "Iniciar Sesión" (Login) and contains fields for "USUARIO" (User) with placeholder "Tu usuario" and "CONTRASEÑA" (Password) with placeholder "*****". A "Iniciar Sesión" (Login) button is at the bottom. Below it are links for "No tienes cuenta? Regístrate" (Don't have an account? Register) and "Olvidaste tu contraseña? Recuperar contraseña" (Forgot your password? Recover password). The right window is titled "Regístrate" (Register) and contains fields for "USUARIO" (User) with placeholder "Tu usuario", "CORREO ELECTRÓNICO" (Email) with placeholder "Correo de ejemplo", "CONTRASEÑA" (Password) with placeholder "*****", and "CONFIRMAR CONTRASEÑA" (Confirm Password) with placeholder "*****". A "Registrarse" (Register) button is at the bottom. Below it are links for "¿Ya tienes cuenta? Iniciar sesión" (Do you have an account? Log in).



The wireframe shows a single modal window titled "Recuperar Contraseña" (Recover Password). It contains a field for "USUARIO" (User) with placeholder "Tu usuario" and a "Enviar enlace de recuperación" (Send recovery link) button. Below the button are links for "¿Ya tienes cuenta? Iniciar sesión" (Do you have an account? Log in) and "¿No tienes cuenta? Regístrate" (Don't have an account? Register).

Inicio

El sitio web de inicio del lado del cliente/consumidor final de la cafetería tiene un mensaje de bienvenida estatico, un display de productos utilizando grid, todo diseñado con CSS con el patrón de modulos CSS, sin dinamismo innecesario, con arquitectura de islas, utilizando componentes globales de header (TopMenu) y footer que se comparten en todas las vistas cuyo renderizado cambia dependiendo de si se ha iniciado sesión o no afectando solo sus propias secciones logicas.



Carrito

El carrito carga renderiza la vista nuevamente en cada iteración del mismo (añadir/quitar un producto) dejando disponible siempre una vista estatica y rápida al usuario final, cuenta con los elementos típicos de cualquier carrito como la opción de modificar cantidades, consultar subtotales y el total así como gestionar la realización de un pedido.

The wireframe shows a clean layout for a shopping cart. At the top, there's a header with the logo and navigation links: 'Agu1406', 'Inicio', 'Carrito', and 'Pedidos'. Below the header is a main title 'Tu carrito' (Your cart). Underneath it is a section titled 'Resumen del carrito.' (Cart summary) which lists four items:

Producto	Precio
Cafe con leche.	1.50
Cafe americano.	1.30
Muffin de arandanos.	1.50
Empanada de atun.	2.00
Total:	6.30

A large 'Realizar pedido' button is located at the bottom of the cart summary. At the very bottom of the page, there's a footer with the company name, contact information, legal links, and a copyright notice.

Pedidos

Los pedidos siguen el mismo patrón de los wireframes anteriores, utilizando arquitectura de islas sin ningún tipo de dinamismo, con estilos CSS 100% nativos diseñados con modulos CSS, compartiendo los componentes globales de TopMenu y footer mostrando toda la información relevante y posible del historial de pedidos:

This wireframe displays a detailed view of an order. At the top, it shows the order identifier 'IDENTIFICADOR: 28'. Below it is a table listing three items:

Producto	Precio	Cantidad
Cafe espresso	1.20	1 unidad
Napolitana de crema	2.00	1 unidad
Cafe con leche	5.00	1 unidad

Below the table, there's a summary section with the text: 'DESCRIPCION', 'PRECIO TOTAL: 8.20€', and 'PAGADO: no'.

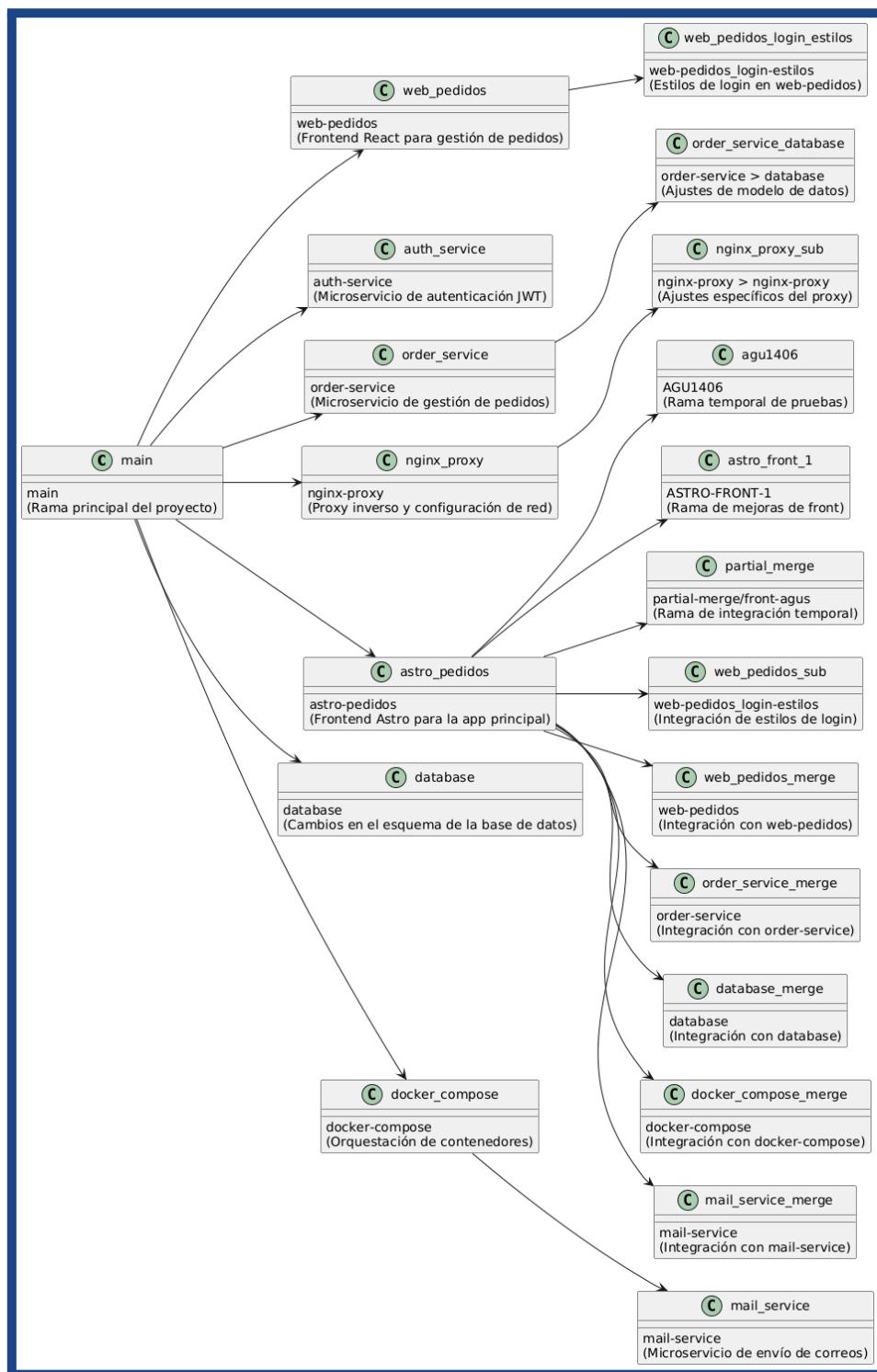
At the bottom of the page, there's a footer with the company name, contact information, legal links, and a copyright notice. A small 'Made with Visly' logo is also present.

frontend - personal

6º. - Implementación y pruebas

6.1. Estructura del proyecto y flujo Git

Tenemos un diagrama del flujo git generado automáticamente gracias a **PlantUML** que muestra de forma jerárquica todas las ramas del proyecto con una breve descripción del objetivo de cada rama, también está disponible en tamaño completo y de acceso público en el siguiente enlace [diagrama de git.png](#)



1. main

- a. **Origen:** Es la rama raíz del proyecto.
- b. **Descripción:** Rama principal donde se integran todas las funcionalidades y desarrollos estables.
- c. **Propósito:** Consolidar el código estable y servir de base para despliegues y nuevas ramas.
- d. **Apunta a:** Todas las ramas principales y sub-ramas parten de aquí y suelen integrarse de vuelta aquí tras ser completadas.

2. web-pedidos

- a. **Origen:** Sale de main.
- b. **Descripción:** Rama para el desarrollo del frontend React de la gestión de pedidos en la cafetería.
- c. **Propósito:** Implementar la interfaz de usuario en tiempo real para la cafetería.
- d. **Apunta a:** Se fusiona regularmente en main tras cada ciclo de desarrollo.

3. web-pedidos_login-estilos

- a. **Origen:** Sale de web-pedidos.
- b. **Descripción:** Subrama para trabajar en los estilos y mejoras visuales del login en el frontend React.
- c. **Propósito:** Mejorar la experiencia de usuario en el proceso de autenticación.
- d. **Apunta a:** Se fusiona en web-pedidos y, posteriormente, en main.

4. auth-service

- a. **Origen:** Sale de main.
- b. **Descripción:** Microservicio de autenticación basado en JWT, gestión de usuarios y roles.
- c. **Propósito:** Proveer autenticación segura y control de acceso a los distintos servicios.
- d. **Apunta a:** Se fusiona en main tras cada mejora o corrección.

5. order-service

- a. **Origen:** Sale de main.
- b. **Descripción:** Microservicio para la gestión de pedidos, lógica de negocio y persistencia.
- c. **Propósito:** Gestionar la creación, actualización y consulta de pedidos.

- d. **Apunta a:** Se fusiona en main tras cada ciclo de desarrollo.

6. order-service > database

- a. **Origen:** Sale de order-service.
- b. **Descripción:** Subrama para ajustes y migraciones en el modelo de datos de pedidos.
- c. **Propósito:** Adaptar la base de datos a nuevas necesidades del microservicio de pedidos.
- d. **Apunta a:** Se fusiona en order-service y luego en main.

7. nginx-proxy

- a. **Origen:** Sale de main.
- b. **Descripción:** Configuración del proxy inverso y reglas de red para los microservicios.
- c. **Propósito:** Gestionar el acceso, balanceo de carga y seguridad de la red.
- d. **Apunta a:** Se fusiona en main.

8. nginx-proxy > nginx-proxy

- a. **Origen:** Sale de nginx-proxy.
- b. **Descripción:** Sub-rama para ajustes específicos y pruebas en la configuración del proxy.
- c. **Propósito:** Experimentar y validar cambios antes de integrarlos en la rama principal del proxy.
- d. **Apunta a:** Se fusiona en nginx-proxy y luego en main.

9. astro-pedidos

- a. **Origen:** Sale de main.
- b. **Descripción:** Frontend principal para clientes, desarrollado con Astro.
- c. **Propósito:** Permitir a los clientes hacer pedidos, ver productos y gestionar su cuenta.
- d. **Apunta a:** Se fusiona en main tras cada iteración.

10. astro-pedidos > AGU1406

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Rama temporal para pruebas o desarrollo de features específicas.
- c. **Propósito:** Experimentar con nuevas funcionalidades o correcciones antes de integrarlas.

- d. **Apunta a:** Se fusiona en astro-pedidos.

11. astro-pedidos > ASTRO-FRONT-1

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Rama para mejoras visuales y de experiencia de usuario en el frontend Astro.
- c. **Propósito:** Desarrollar y probar nuevas interfaces o componentes.
- d. **Apunta a:** Se fusiona en astro-pedidos.

12. astro-pedidos > partial-merge/front-agus

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Rama de integración temporal para pruebas de merges complejos o parciales.
- c. **Propósito:** Validar la integración de cambios antes de fusionarlos definitivamente.
- d. **Apunta a:** Se fusiona en astro-pedidos.

13. astro-pedidos > web-pedidos_login-estilos

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de los estilos de login desarrollados en web-pedidos dentro del frontend Astro.
- c. **Propósito:** Unificar la experiencia de usuario entre ambos frontends.
- d. **Apunta a:** Se fusiona en astro-pedidos.

14. astro-pedidos > web-pedidos

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de funcionalidades o componentes de web-pedidos en el frontend Astro.
- c. **Propósito:** Compartir o migrar funcionalidades entre ambos frontends.
- d. **Apunta a:** Se fusiona en astro-pedidos.

15. astro-pedidos > order-service

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de funcionalidades del microservicio de pedidos en el frontend Astro.
- c. **Propósito:** Sincronizar cambios entre el backend de pedidos y el frontend.

- d. **Apunta a:** Se fusiona en astro-pedidos.

16. astro-pedidos > database

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de cambios en la base de datos que afectan al frontend Astro.
- c. **Propósito:** Adaptar el frontend a nuevas estructuras o datos.
- d. **Apunta a:** Se fusiona en astro-pedidos.

17. astro-pedidos > docker-compose

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de cambios en la orquestación de contenedores que afectan al frontend Astro.
- c. **Propósito:** Asegurar la compatibilidad y despliegue correcto del frontend.
- d. **Apunta a:** Se fusiona en astro-pedidos.

18. astro-pedidos > mail-service

- a. **Origen:** Sale de astro-pedidos.
- b. **Descripción:** Integración de funcionalidades de notificación por correo en el frontend Astro.
- c. **Propósito:** Permitir notificaciones y confirmaciones por email a los usuarios.
- d. **Apunta a:** Se fusiona en astro-pedidos.

19. database

- a. **Origen:** Sale de main.
- b. **Descripción:** Rama para cambios y migraciones en el esquema de la base de datos MySQL.
- c. **Propósito:** Adaptar la base de datos a las necesidades de los microservicios y frontends.
- d. **Apunta a:** Se fusiona en main.

20. docker-compose

- a. **Origen:** Sale de main.
- b. **Descripción:** Rama para la orquestación de contenedores Docker de todos los servicios.
- c. **Propósito:** Facilitar el despliegue y la integración de los distintos componentes del

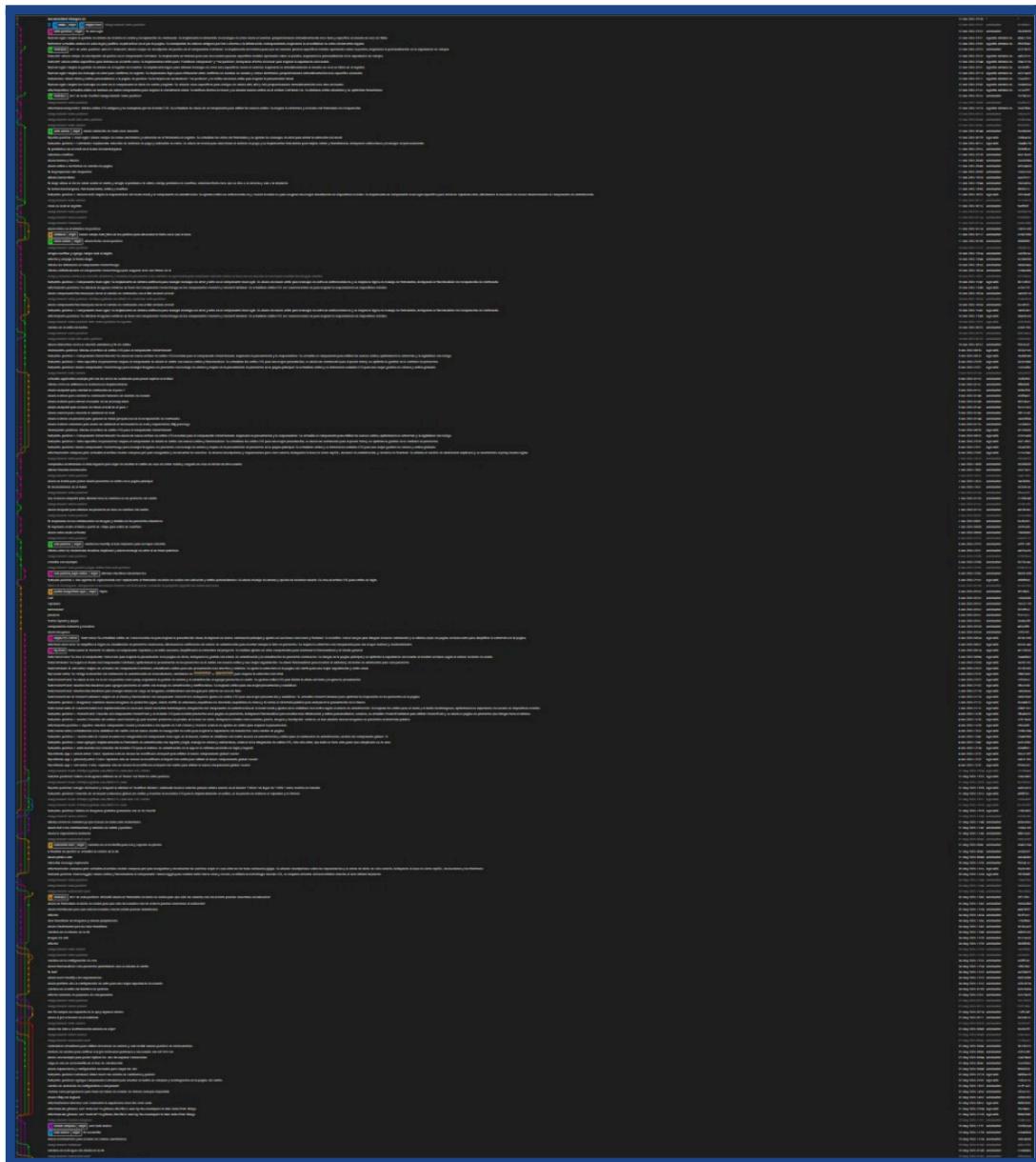
sistema.

- d. **Apunta a:** Se fusiona en main.

21. mail-service

- a. **Origen:** Sale de docker-compose.
- b. **Descripción:** Microservicio para el envío de correos electrónicos de confirmación y notificaciones.
- c. **Propósito:** Gestionar la comunicación por email con los usuarios.
- d. **Apunta a:** Se fusiona en docker-compose y luego en main.

Además, existen más de 350 commits diferentes del proyecto, en total, repartidos en todas las ramas, pintarlo es bastante complejo, sin embargo podemos adjuntar una captura directa del flujo que permite darse una idea del total:



The image shows a GitHub commit history visualization. The interface features a dark background with a grid of small, colorful cards representing individual commits. A vertical timeline bar on the right side tracks the sequence of commits over time. A legend on the left side identifies the colors used for different commit types, such as green for merges and blue for pushes. The overall layout is a dense, horizontal scrollable view of a massive amount of commit data.

6.2. Tecnologías y dependencias utilizadas

El enfoque que hemos ideado para el desarrollo ha sido el de aprovechar todas las tecnologías conocidas y dar un paso más allá ampliando nuestro conocimiento personal y profesional al aprovechar nuestra natural curiosidad por nuevas tecnologías, metodologías y patrones, por eso, por eso hemos separado la lista de tecnologías y dependencias en subapartados dedicados cada uno a una cosa.

6.2.1 Tecnologías y dependencias en el backend

Las tecnologías empleadas en el backend son todas aquellas que manejan la lógica, gestión, organización, tratamientos de datos, seguridad y funcionamiento del servidor, en el proyecto actual se han empleado las siguientes tecnologías y dependencias:

- **Java:** La elección de Java como lenguaje servidor ha sido personal, Java lleva existiendo muchos años, es un lenguaje maduro, muy enriquecido, con frameworks muy potentes como Spring Boot, Spring Security, etc, con IDEs muy potentes y estables como IntelliJ o VSCode con las extensiones correctas, además la maquina virtual de Java permite su ejecución en prácticamente cualquier entorno, como dice el eslogan “**Write Once, Run Anywhere**”, siendo nuestra elección precisamente, por ello, a raíz de decidir usar Java se han empleado las siguientes tecnologías y dependencias relacionadas con el propio lenguaje:
 - **SpringBoot:** Es uno de los frameworks más populares a nivel mundial que permite la creación de microservicios y software RESTful con configuraciones mínimas, se ha utilizado en el proyecto para la mayoría de microservicios tal y como la gestión de órdenes / pedidos, los correos electrónicos y la autenticación.
 - **SpringSecurity:** Es un framework de la familia **Spring** al igual que **SpringBoot**, se encarga de toda la gestión de autenticación en aplicaciones Java, es altamente personalizable permitiendo no solo autenticar a los usuarios si no definir lo que son capaces de hacer o no en la aplicación, ampliamente utilizado para definir las autoridades de los tres roles de la aplicación, “ADMIN”, “USER” y “SYSTEM”.
- **TypeScript:** Es un lenguaje potente y flexible utilizable en ambos, backend y frontend, es una versión potente de JavaScript que incluye tipado de variable, compilación, etc, que se “transpila” a JavaScript en tiempo de ejecución para máxima compatibilidad, a raíz de elegir TypeScript se implementaron las siguientes tecnologías o dependencias:
 - **Node.js:** Es un entorno de ejecución en servidor que permite superar la barrera típica de JavaScript que requiere un navegador para la ejecución de código, con Node.js podemos ejecutar código JavaScript (y consecuentemente TypeScript) en servidor.
 - **Nestjs:** En nuestro caso, haciendo uso de **TypeScript** como lenguaje tipado de construcción y **Node.js** como compilador y ejecutor del código empleamos **Nestjs** en el proyecto para la creación de un microservicio **websocket** que permite la comunicación en tiempo real entre el frontend del personal de la cafetería y los

pedidos en la base datos usando **prisma** para recibirlos en vivo y en directo.

- **JWT (JSON WEB TOKEN):** Es un estándar ampliamente utilizado para crear tokens cifrados de acceso que permiten la comunicación segura entre los entornos de cliente y servidor, cuando se codifica tiene 3 partes principales en **Base64** las cuales son el header, el payload y el signature, cada uno con funciones específicas, para implementarlo hemos hecho uso de una librería muy popular de Java llamado **JJWT (Java JSON Web Token)** que permite la creación y manipulación de los mismos.

6.2.2 Tecnologías y dependencias en el frontend

La cantidad de tecnologías empleadas en el desarrollo del lado cliente se han limitado a las dos cuya curva de aprendizaje pero potencial y escabilidad han ido más de la mano y esas son:

- **Astro:** Es un framework muy moderno de desarrollo web que prioriza la entrega de páginas web completamente estáticas con práctica nada de dinamismo, el contenido de las páginas se renderiza / compila en servidor y a cambio entrega sitios web rápidos con HTML y CSS estático, para emplear Astro se debe utilizar la **arquitectura de islas** donde los sitios web estáticos solo se **hidratan** de dinamismo (JavaScript) cuando es necesario, ni más, ni menos, normalmente se hidratación ocurre a través de otros frameworks siendo nuestro caso el de React.
- **React:** Es un framework muy popular en el diseño web moderno que permite la creación de interfaces a través de pequeños componentes que manejan lógica y permiten la modularización reciclando lo máximo posible todos los recursos, en nuestro caso, por ejemplo, el poco dinamismo que tiene la aplicación ocurre precisamente con componentes React.

6.3. Desarrollo por módulos

6.3.1. Módulo de autenticación:

Hemos desarrollado un microservicio de autenticación con Spring Boot que utiliza JWT para una seguridad robusta. El sistema permite el registro de usuarios con validación de contraseñas (mínimo 12 caracteres), encriptación BCrypt y asignación automática del rol USER. El inicio de sesión genera tokens JWT válidos por 8 horas, y se incluye un proceso de recuperación de contraseña mediante tokens especiales enviados por email. La autenticación en cada petición se gestiona con filtros JWT que validan el token y establecen el contexto de seguridad. Este microservicio, que utiliza Spring Security para roles (USER, ADMIN, SYSTEM) y MySQL como base de datos, además envía emails de forma asíncrona a través de webhooks.

6.3.2. Módulo de gestión de pedidos

Estamos implementando un microservicio de gestión de pedidos integral que cubre todo el ciclo de compra. Los usuarios pueden gestionar sus carritos, y al crear un pedido, este se guarda en una base de datos MySQL (persistencia con Spring Boot y JPA/Hibernate) y genera notificaciones en tiempo real. Un historial de pedidos con paginación está disponible, y los administradores reciben actualizaciones instantáneas de nuevos pedidos (y pueden marcarlos como completados) gracias a la comunicación en tiempo real vía WebSockets (con Nestjs). La autenticación entre

microservicios se maneja con JWT, garantizando una experiencia de compra fluida y un control eficiente para el personal de la cafetería.

6.3.3 Módulo de servidor de pedidos en tiempo real

Hemos desarrollado un servidor WebSocket con Nestjs para la comunicación en tiempo real entre el sistema de pedidos y el panel de administración. Utiliza Socket.IO y Fastify, con Zod para validación y Prisma ORM para MySQL. Implementa doble autenticación JWT (sistema y administrador) y mantiene los pedidos activos sincronizados en memoria. Los pedidos se reciben vía webhook (desde el microservicio de pedidos), se validan y se emiten a todos los clientes, garantizando notificaciones instantáneas y sincronización en tiempo real para el personal de la cafetería.

6.3.4 Módulo de envío de correos

El módulo de envío de correos es un microservicio independiente (mail-service) que utiliza Spring Boot y JavaMail para gestionar el envío de emails en la aplicación de cafetería. Está compuesto por un servicio principal (MailService) que valida direcciones de correo mediante expresiones regulares, verifica que el asunto no exceda 50 caracteres, y envía mensajes HTML utilizando SMTP (configurado para Gmail). El módulo se integra con el servicio de autenticación a través de webhooks asíncronos (MailWebhook) que permiten enviar correos de bienvenida cuando un usuario se registra y correos de recuperación de contraseña cuando se solicita un restablecimiento, manteniendo una arquitectura desacoplada donde cada servicio tiene su responsabilidad específica y se comunica mediante HTTP REST.

6.3.5 Módulo de pagos

El módulo de pagos es una implementación frontend completa que maneja cuatro métodos de pago (efectivo, tarjeta, Bizum y transferencia) a través de un componente React (CartIsland.tsx) que incluye validación de datos específica para cada método (número de tarjeta de 16 dígitos, CVV de 3 dígitos, teléfono de 9 dígitos para Bizum, IBAN válido para transferencias), simulación de tiempos de procesamiento diferenciados según el método, y una interfaz de usuario con pasos secuenciales (selección, formulario, procesamiento). Sin embargo, en el backend (MakeOrderUseCaseImpl.java) se observa que actualmente todos los pedidos se marcan como isPaid = false con un comentario "todo: webhook a sumup", indicando que la integración real con pasarelas de pago está pendiente de implementación, por lo que el módulo funciona como una simulación completa del flujo de pago pero sin procesamiento real de transacciones financieras.

6.3.6 Módulo frontend para clientes

El módulo frontend para clientes es una aplicación web desarrollada con Astro y React que proporciona una interfaz completa para que los usuarios puedan realizar pedidos en la cafetería. La aplicación incluye una página principal con un banner de bienvenida y un catálogo de productos obtenidos desde una base de datos Prisma, un sistema de navegación responsive con menú hamburguesa para móviles que incluye autenticación de usuarios, un carrito de compras interactivo con gestión de productos y múltiples métodos de pago (efectivo, tarjeta, Bizum, transferencia), una página de historial de pedidos, y un sistema de diseño consistente con variables

CSS personalizadas que definen una paleta de colores específica (amarillo-verde, limón, beige y bistre). La aplicación utiliza transiciones de navegación fluidas, componentes React para funcionalidad interactiva, y está optimizada para dispositivos móviles con diseño responsive y accesibilidad web.

6.3.7 Módulo frontend para personal

El módulo frontend para personal es una aplicación React desarrollada con Vite que proporciona una interfaz de gestión de pedidos en tiempo real para el personal de la cafetería, incluyendo un sistema de autenticación que valida credenciales contra el servicio de autenticación y requiere permisos de administrador, comunicación en tiempo real mediante Socket.IO para recibir pedidos automáticamente con notificaciones toast, una interfaz que muestra cada pedido en tarjetas con información del comprador, productos, estado de pago y funcionalidad para marcarlos como concluidos, además de características adicionales como toggle de tema claro/oscuro, indicador de estado de conexión WebSocket y animaciones de fade-out al completar pedidos.

6.3.8 Módulo de base de datos

El módulo de bases de datos utiliza MySQL 8.0.33 como sistema de gestión principal, implementado mediante un contenedor Docker que ejecuta automáticamente el script SQL de inicialización, y está estructurado con siete tablas principales: users (gestión de usuarios con autenticación), products (catálogo de productos con precios y disponibilidad), orders (pedidos con estados PENDIENTE/FINALIZADO), orders_has_products (relación muchos-a-muchos entre pedidos y productos), cart y cart_has_products (carrito de compras temporal), y userAuthorities (sistema de roles ADMIN/SYSTEM/USER). La base de datos utiliza Prisma como ORM en el frontend y servicios WebSocket, mientras que los servicios Spring Boot emplean JPA/Hibernate para el mapeo objeto-relacional, manteniendo una arquitectura híbrida que permite consultas optimizadas y transacciones seguras con claves foráneas y restricciones de integridad referencial.

6.3.9 Módulo de proxy inverso

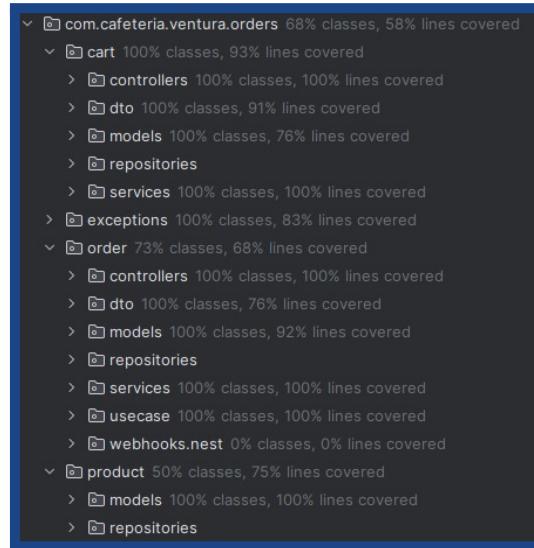
El módulo de proxy inverso utiliza Nginx como servidor web y proxy inverso, implementado mediante un contenedor Docker basado en Alpine Linux que expone los puertos 80 y 443, y está configurado para redirigir el tráfico HTTP entrante hacia los diferentes servicios backend según las rutas: /auth/ se dirige al servicio de autenticación Spring Boot (puerto 8080), mientras que /order/ y /cart/ se redirigen al servicio de pedidos Spring Boot (puerto 8081), centralizando así el acceso a todos los microservicios backend desde un único punto de entrada y proporcionando una capa de abstracción que simplifica la gestión de rutas y mejora la seguridad al ocultar la arquitectura interna de servicios.

6.4. Plan de pruebas

6.4.1. Pruebas unitarias

El plan de pruebas unitarias del proyecto se basa en la automatización y análisis de la cobertura mediante las herramientas elegidas para nuestro proyecto (JUnit, Mockito y SonarQube) abarcando los principales módulos del backend (carrito, pedidos, productos y excepciones) para asegurar que la lógica de negocio, los controladores y los servicios funcionen correctamente de

forma aislada. Las pruebas se diseñan para validar tanto casos exitosos como situaciones de error, y su ejecución regular permite detectar rápidamente regresiones o fallos. El uso de SonarQube facilita la medición precisa de la cobertura de código, identificando qué clases y líneas han sido ejecutadas por los tests, lo que garantiza un alto nivel de fiabilidad, se adjunta en la siguiente imagen el porcentaje total de cobertura de las pruebas:



6.4.2. Pruebas de integración

El plan de pruebas de integración del proyecto consiste en verificar que los diferentes módulos y componentes del backend interactúan correctamente entre sí, utilizando pruebas automatizadas que simulan flujos completos de negocio, como la gestión de pedidos, el funcionamiento del carrito y la comunicación entre servicios. Estas pruebas se implementan principalmente con JUnit y Mockito, y se ejecutan en un entorno controlado donde se mockean o integran dependencias reales según el caso, asegurando que la integración entre controladores, servicios y repositorios funcione como se espera. El análisis de cobertura realizado con SonarQube permite identificar qué rutas y procesos de integración están correctamente validados, garantizando así la robustez y fiabilidad del sistema en escenarios reales de uso, para ello adjuntamos la siguiente imagen que demuestra la superación de las pruebas:

		1 sec 38 ms
✓	<default package>	
✓	CartControllerTest	
✓	addProduct_whenQuantityisValid_shouldReturn200()	53 ms
✓	getAllProducts_whenCartIsEmpty_shouldReturn200()	45 ms
✓	deleteOneProduct_whenInCartExist_cart_shouldReturn200()	21 ms
✓	getCartByUsername_whenCartIsEmpty_shouldReturn204()	17 ms
✓	getCartByUserEntry_whenCartIsEmpty_shouldReturn204()	4 ms
✓	addProduct_whenQuantityIsInvalid_shouldThrow400()	24 ms
✓	deleteOneProduct_whenProductNotExistInCart_shouldThrow404()	12 ms
✓	MakeOrderUseCaseImplTest	111 ms
✓	makeOrder_whenCartHasProducts_shouldThrow400()	102 ms
✓	makeOrder_whenHappyPath_shouldReturnOrderDetails()	7 ms
✓	makeOrder_whenUsernameNotExist_shouldThrow403()	2 ms
✓	OrderControllerTest	37 ms
✓	makeNewOrder()	10 ms
✓	getOrderUserHistory()	27 ms
✓	CartServiceTest	238 ms
✓	addProduct_whenProductNotExistInCart_shouldCreateWithQuantity()	172 ms
✓	getCartByUsername_whenUserNotExist_shouldThrow403()	8 ms
✓	getCartByUserEntry_whenCartExist_shouldReturnExistingCart()	3 ms
✓	getProductId_whenProductExist_shouldReturnOptionalCartHasProduct()	4 ms
✓	deleteOneProduct_whenQuantityIsMoreThanOne_shouldReturnOptionalCartHasProducts()	3 ms
✓	clearCart()	2 ms
✓	addProduct_whenProductNotExistDB_shouldThrow404()	5 ms
✓	getCartByUsername_whenCartExist_shouldReturnExistingCart()	5 ms
✓	getProductById_whenProductNotExist_shouldReturn404()	3 ms
✓	getAllProductsCart()	3 ms
✓	getAllProductsByUsername()	2 ms
✓	getCartByUserEntry_whenCartExist_shouldReturnNewCart()	5 ms
✓	getCartByUsername_whenCartExist_shouldReturnNewCart()	5 ms
✓	deleteOneProduct_whenQuantityIsOneOrLess_shouldReturnOptionalEmpty(int)	14 ms
✓	[1] quantity=1	11 ms
✓	[2] quantity=0	2 ms
✓	[3] quantity=-1	1 ms
✓	addProduct_whenProductExistInCart_shouldSumQuantity()	4 ms
✓	OrderServiceTest	119 ms
✓	registerOrder_whenNestAvailable_shouldReturnOrderDTO()	99 ms
✓	getOrdersByUsername_whenUserNotExist_shouldThrow403()	2 ms
✓	getOrdersByUsername_whenUserExist_shouldReturnOrderDTO()	15 ms
✓	registerOrder_whenNotAvailable_shouldThrow500()	3 ms

7º. - Implantación

7.1. Pipeline CI/CD

En el desarrollo de software moderno, la integración de un pipeline **CI/CD** (Integración Continua y Entrega/Despliegue Continuo) es una práctica recomendada para automatizar la construcción, pruebas y despliegue de aplicaciones. Un pipeline **CI/CD** permite que, cada vez que se suben cambios al repositorio, el sistema ejecute automáticamente pruebas, compile el código y, si todo es correcto, despliegue la nueva versión en el entorno de producción o pre-producción.

En este proyecto, no se ha implementado un pipeline **CI/CD** automatizado debido a limitaciones de tiempo y a que no era un requisito obligatorio para los objetivos del trabajo. Sin embargo, la arquitectura basada en microservicios y el uso de **contenedores Docker** facilitan la futura integración de este tipo de herramientas.

Actualmente, el proceso de despliegue se realiza de forma manual, utilizando Docker y Docker Compose para levantar los diferentes servicios del sistema. Este enfoque permite una gestión eficiente y reproducible de los entornos, aunque requiere intervención manual para cada actualización.

Como mejora futura, se recomienda la integración de un pipeline **CI/CD** utilizando herramientas como **GitHub Actions**, **GitLab CI** o **Jenkins**.

7.2. Despliegue en entorno de producción

El despliegue en entorno de producción consiste en poner en funcionamiento la aplicación para que pueda ser utilizada por los usuarios finales, asegurando que todos los servicios estén correctamente configurados, conectados y accesibles. En este proyecto, el despliegue se ha realizado utilizando Docker y Docker Compose, lo que permite levantar todos los microservicios (backend, frontend, base de datos, servicios de correo, etc.) de forma sencilla, reproducible y aislada. Esta metodología facilita la gestión de dependencias y la portabilidad del sistema entre diferentes entornos, el flujo de despliegue sería el siguiente:

- **Preparación del entorno:** Se requiere tener instalado Docker y Docker Compose en el servidor de producción.
- **Configuración de variables de entorno:** Se deben definir los archivos .env y los archivos de configuración necesarios para cada microservicio (por ejemplo, credenciales de base de datos, claves JWT, etc.).
- **Levantado de servicios:** Desde la raíz del proyecto, se ejecuta el comando docker-compose up -d, que inicia todos los contenedores definidos en el archivo docker-compose.yml.
- **Verificación de servicios:** Se comprueba que todos los servicios estén en funcionamiento y correctamente conectados (backend, frontend, base de datos, correo, etc.).
- **Acceso a la aplicación:** Una vez desplegada, la aplicación es accesible a través de la URL o IP pública del servidor, gestionada por el proxy inverso (Nginx) incluido en la arquitectura.

7.3. Guía de instalación y configuración

Para instalar y configurar el sistema de gestión de pedidos para cafetería, sigue estos pasos detallados:

- **Clonar el repositorio:** Este paso requiere acceder al repositorio en **Github** o **Git** a través de la URL <https://github.com/filuf/TFG-DAW> o bien utilizando una interfaz gráfica como **Github Desktop** o la extensión de GitHub en **VSCode** o desde una terminal utilizando el comando **git clone** seguido de la **URL** estando dentro del directorio deseado para la clonación.
- **Instalar Docker Desktop y Docker Compose:** Con el repositorio clonado el siguiente paso es descargar **Docker Desktop** para poder correr los microservicios en sus respectivos contenedores y **Docker Compose** para la instalación de dependencias, todo ello desde el enlace oficial de docker desktop: [Docker Desktop: The #1 Containerization Tool for Developers](#) y el enlace oficial de docker compose [Install | Docker Docs](#).
- **Configurar variables de entorno y archivos de configuración:** Para esto es necesario disponer de los archivos de entorno llamados “.env” que son privados del proyecto, se puede solicitar acceso a los mismos al correo de agustin6041@outlook.es o aitorpj93@gmail.com, a continuación se muestra en forma de lista todos los directorios relativos que necesitan uno de estos archivos de configuración:
 - **frontend/web-pedidos/.env**
 - **frontend/cafeteria-app/astro-pedidos/.env**
 - **backend/pedidos-service/orders/src/main/resources/application.yml**
 - **backend/authentication-service/auth/src/main/resources/application.yml**
 - **backend/mail-service/mailer/src/main/resources/application.yml**
- **Configurar correctamente la base de datos:** Hay que asegurarse de definir correctamente las credenciales de la base de datos, claves **JWT**, y datos de correo electrónico.
- **Arrancar los servicios con Docker Compose:** Desde la raíz del proyecto se tiene que ejecutar el comando “**docker-compose up -d**” o bien dirigirse al archivo **docker-compose.yml** y ejecutar manualmente uno a uno los servicios. Esto levantará todos los microservicios, la base de datos MySQL, el proxy inverso Nginx y los frontends.

8º. - Resultados y discusión

8.1. Comparativa temporal: planificado vs. ejecutado

En términos generales el desarrollo ha sido completo de la aplicación, nos ha faltado solamente (en términos de tiempo) desarrollar el **pipeline CI/CD** y la **API** real de pagos que fue inaccesible debido a los largos tiempos de respuestas inconclusas además que se nos brindó por parte de la compañía encargada de la gestión de la pasarela de pago real utilizada actualmente en la cafetería del instituto, además, nos hubiera gustado implementar un sistema de traducciones

que permitiese elegir el idioma de la aplicación.

8.2. Dificultades más importantes encontradas

Para explicar las dificultades más importantes o destacadas del proyecto podemos separarlas en la siguiente lista auto-explicativa:

- **SpringSecurity:** Es muy extenso, tiene muchos niveles de abstracción, hace falta ver en muchos escenarios como corre por debajo muchas funciones o métodos clásicos del framework lo que hizo que su curva de aprendizaje fuese alta, llevando casi un mes entero leer y entender la documentación para implementarlo de forma correcta.
- **Complejidad “n+1”:** Es un problema muy común cuando se hacen consultas masivas de un conjunto de datos y luego consultas sucesivas de varios de esos datos uno tras otros, en nuestro caso se consultaban todos los productos y acto seguido de forma individual los atributos de los productos previamente cargados, se solucionó de forma nativa al renderizar la carga de dato en el servidor y enviar los datos estáticos al cliente aprovechando **Astro**.

8.3. Hasta dónde se ha llegado: funcionalidades

El proyecto es 100% operativo, quedando solamente pendiente la implementación de una pasarela de pagos real que no se pudo conseguir debido a la ausencia de respuestas por parte de la compañía encargada de la pasarela de pagos actual y real de la cafetería, además, faltó implementar el **CI/CD**, sin embargo, el resto de la aplicación de desarrollo tal y como se esperaba sin problemas.

8.4. Posibles extensiones y mejoras futuras

Se ha considerado en versiones futuras del proyecto (que no será abandonado después de la presentación) modificar o mejorar el comportamiento de la plataforma empleando los siguientes cambios, mejoras y/o extensiones:

- **Uso de proyecciones:** Implementar proyecciones en la base de datos para optimizar las consultas de productos y pedidos. Esto permitiría crear vistas específicas que contengan solo los datos necesarios para cada operación, reduciendo el tiempo de respuesta y mejorando el rendimiento general de la aplicación. Por ejemplo, crear proyecciones para "Productos con stock disponible", "Pedidos pendientes de pago" o "Resumen de ventas diarias".
- **Uso de oauth:** Integrar autenticación OAuth 2.0 para permitir que los usuarios se registren e inicien sesión usando sus cuentas de Google, Facebook, Apple o Microsoft. Esto simplificará el proceso de registro, aumentaría la confianza del usuario y reduciría la fricción en el proceso de compra. También facilita la recuperación de contraseñas y mejoraría la seguridad general del sistema.
- **Refresh tokens:** Implementar un sistema de refresh tokens para mejorar la seguridad y experiencia del usuario. Los refresh tokens permitirían que las sesiones de usuario se mantengan activas por períodos más largos sin comprometer la seguridad, renovando

automáticamente los tokens de acceso cuando sea necesario. Esto es especialmente útil para aplicaciones móviles y para mantener la sesión activa durante largas sesiones de compra.

- **Uso de kubernetes:** Migrar la aplicación a un entorno con Kubernetes para mejorar la escalabilidad, disponibilidad y mantenimiento del sistema. Kubernetes permitiría desplegar múltiples instancias de la aplicación, gestionar automáticamente el balanceo de carga, realizar actualizaciones sin tiempo de inactividad y escalar horizontalmente según la demanda. Esto sería especialmente beneficioso durante períodos de alta actividad como eventos especiales o promociones.
- **Uso de estándar jsonapi:** Adoptar el estándar JSON:API para estandarizar todas las comunicaciones entre el frontend y backend. Este estándar proporciona una estructura consistente para las respuestas de la API, incluyendo manejo de errores, paginación, filtrado y relaciones entre recursos. Esto simplificará el desarrollo del frontend, mejoraría la mantenibilidad del código y facilitaría la integración con aplicaciones móviles futuras.

9º. - Conclusiones

9.1. Aportación al aprendizaje y competencias adquiridas

Este proyecto ha representado una oportunidad invaluable para el desarrollo de competencias técnicas y profesionales, permitiendo la aplicación práctica de conocimientos teóricos adquiridos durante el ciclo formativo. A través del desarrollo de esta aplicación de cafetería, se han consolidado y ampliado las siguientes competencias:

Desarrollo Frontend:

- Dominio de React.js para la creación de interfaces de usuario interactivas y responsivas
- Implementación de Astro como framework meta para optimización de rendimiento, SEO y generación de sitios estáticos
- Configuración y uso de Vite como herramienta de build para desarrollo rápido y eficiente
- Gestión avanzada de estado con hooks de React (useState, useEffect, useRef)
- Desarrollo de componentes modulares y reutilizables
- Implementación de sistemas de autenticación y autorización en el frontend
- Manejo de formularios complejos con validación en tiempo real
- Optimización de rendimiento con lazy loading y code splitting

Desarrollo Backend:

- Diseño e implementación de APIs RESTful con Spring Boot
- Gestión de bases de datos relacionales con Prisma ORM para type-safe database queries
- Implementación de sistemas de autenticación JWT

- Manejo de errores HTTP y respuestas estructuradas
- Integración de servicios de correo electrónico para recuperación de contraseñas
- Implementación de lógica de negocio para gestión de pedidos y carritos

Gestión de datos:

- Diseño de esquemas de base de datos optimizados con Prisma Schema
- Implementación de relaciones entre entidades (usuarios, productos, pedidos)
- Gestión de transacciones y integridad referencial
- Optimización de consultas para mejorar el rendimiento
- Migraciones de base de datos con Prisma Migrate

Herramientas de desarrollo:

- Configuración de entornos de desarrollo con Vite para hot reload y optimización
- Gestión de dependencias y build processes
- Configuración de variables de entorno y configuración de desarrollo
- Optimización de bundles y assets con Vite
- Competencias transversales adquiridas:

Gestión de proyectos:

- Planificación y organización del desarrollo en fases
- Gestión de versiones con Git y control de cambios
- Documentación técnica y de usuario
- Resolución de problemas y debugging de aplicaciones complejas

Trabajo en equipo:

- Colaboración efectiva en el desarrollo de funcionalidades
- Comunicación técnica entre frontend y backend
- Revisión de código y mejora continua
- Distribución de tareas y coordinación de esfuerzos

Pensamiento crítico:

- Análisis de requisitos y diseño de soluciones
- Evaluación de diferentes enfoques técnicos

- Optimización de rendimiento y experiencia de usuario
- Consideración de aspectos de seguridad y escalabilidad

Aprendizaje específico del dominio:

- E-commerce y gestión de pedidos:
- Comprensión de flujos de compra y experiencia de usuario
- Implementación de sistemas de carrito de compras
- Gestión de estados de pedidos y pagos
- Consideraciones de UX/UI para aplicaciones comerciales

Seguridad y autenticación:

- Implementación de sistemas de login/registro seguros
- Manejo de tokens JWT y sesiones
- Validación de datos y prevención de vulnerabilidades
- Recuperación segura de contraseñas

Arquitectura moderna:

- Integración de Astro con React para optimizar el rendimiento
- Uso de Vite para desarrollo rápido y builds optimizados
- Implementación de Prisma para type-safe database operations
- Arquitectura híbrida que combina SSR, SSG y CSR según las necesidades

Este proyecto ha servido como puente entre la formación académica y el desarrollo profesional adquiridos en el primer año del ciclo de formación de grado superior más los esfuerzos más allá del instituto llevados a cabo por Aitor y por mi persona, Agustín, para ser cada día un poco mejores, como personas y como profesionales, proporcionando experiencia práctica en tecnologías modernas y metodologías de desarrollo actuales. La aplicación desarrollada demuestra la capacidad de crear soluciones completas y funcionales que pueden ser utilizadas en un entorno real de negocio.

10º. - Bibliografía y referencias

Spring Security / JWT

OpenWebinars. (s.f.). Seguridad JWT en API REST con Spring Boot. Recuperado de <https://openwebinars.net/academia/aprende/seguridad-jwt-api-rest-spring-boot/>

Spring. (s.f.). Spring Security: Reference Documentation. Recuperado de <https://docs.spring.io/spring-security/reference/index.html>

JWT.io. (s.f.). JSON Web Tokens - Introduction. Recuperado de <https://jwt.io/>

Saxena, T. (2023, marzo 8). Spring Security: The Security Filter Chain. Medium. Recuperado de <https://medium.com/@tanmaysaxena2904/spring-security-the-security-filter-chain-e09e1f53b73d>

CodeWithTech. (2023, abril 3). Understanding SecurityContext and SecurityContextHolder in Spring Security. Medium. Recuperado de <https://medium.com/@CodeWithTech/understanding-securitycontext-and-securitycontextholder-in-spring-security-e8ec9c030819>

Spring Data JPA

Spring. (s.f.). Getting Started – Spring Data JPA. Recuperado de <https://docs.spring.io/spring-data/jpa/reference/jpa/getting-started.html>

Shamaii, K. (2021, junio 30). What is N+1 Query Problem in Spring Data JPA and How to Solve It. Medium. Recuperado de <https://medium.com/@kiarash.shamaii/what-is-n-1-query-generate-problem-in-spring-data-jpa-and-how-to-solve-it-2f3b9f1a7a0b>

Baeldung. (s.f.). Spring JPA: Using @Query with Embedded Method Parameters. Recuperado de <https://www.baeldung.com/spring-jpa-embedded-method-parameters>

Baeldung. (s.f.). Spring Data JPA Pagination and Sorting. Recuperado de <https://www.baeldung.com/spring-data-jpa-pagination-sorting>

Spring Mail

Baeldung. (s.f.). Sending Emails with Spring Boot. Recuperado de <https://www.baeldung.com/spring-email>

Gmail para aplicaciones

Google. (s.f.). Permitir aplicaciones menos seguras. Recuperado de <https://support.google.com/mail/answer/185833?hl=es>

Arquitectura de Software

Martin, R. C. (2011, septiembre 30). Screaming Architecture. The Clean Code Blog. Recuperado de <https://blog.cleancoder.com/uncle-bob/2011/09/30/Screaming-Architecture.html>

Dvmhn07. (2023). Screaming Architecture: Letting your code tell its story. Medium. Recuperado de <https://dvmhn07.medium.com/screaming-architecture-letting-your-code-tell-its-story-203de594cf74>

Socket.IO

Socket.IO. (s.f.). Use Socket.IO with React. Recuperado de <https://socket.io/how-to/use-with-react>

NestJS. (s.f.). WebSockets - Gateways. Recuperado de <https://docs.nestjs.com/websockets/gateways>

NestJS

Hindavi, S. (2023, julio 28). NestJS Crash Course. [Video]. YouTube.
<https://www.youtube.com/watch?v=-ahCssisfwQ>

NestJS. (s.f.). Techniques: Performance (Cache, Compression, etc.). Recuperado de <https://docs.nestjs.com/techniques/performance>

React

Traversy Media. (2023, abril 15). React Full Course 2023. [Video]. YouTube.
https://www.youtube.com/watch?v=7iobxzd_2wY

Astro + Prisma

Lee, A. (2023, octubre 12). Astro + Prisma: Tutorial Completo. [Video]. YouTube.
<https://www.youtube.com/watch?v=4vlyfUMDFnI>

TypeScript

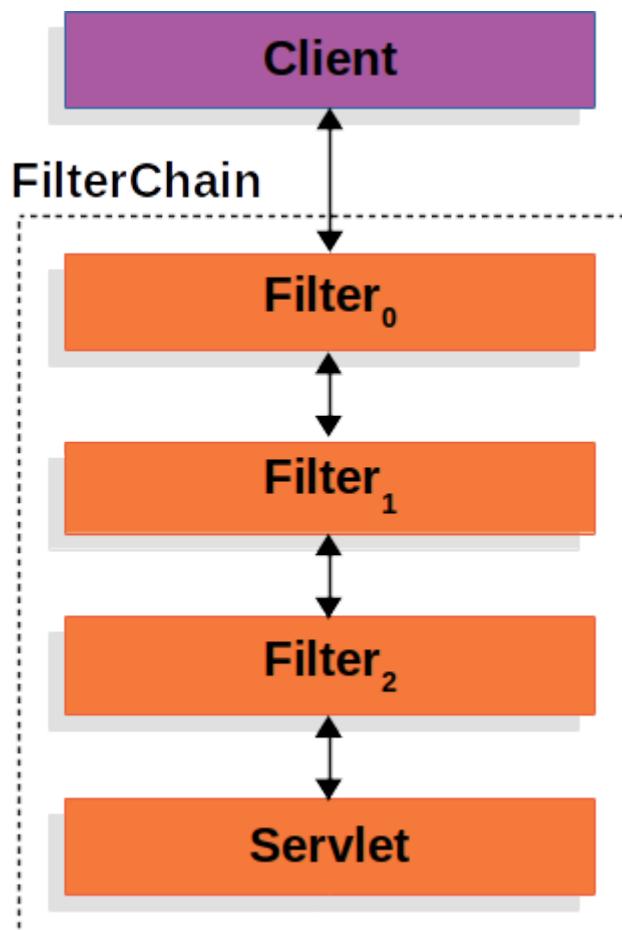
midudev. (2023, agosto 9). TypeScript Full Course. [Video]. YouTube.
https://www.youtube.com/watch?v=fUgxxhI_bvc

ANEXO A - SEGURIDAD Y AUTENTICACIÓN

1. Autenticación en Spring Security

Tanto para nuestro microservicio de autenticación como de pedidos (construidos en Spring Boot) hemos utilizado el framework de Spring Security que es el estándar a la hora de implementar seguridad en Spring Framework.

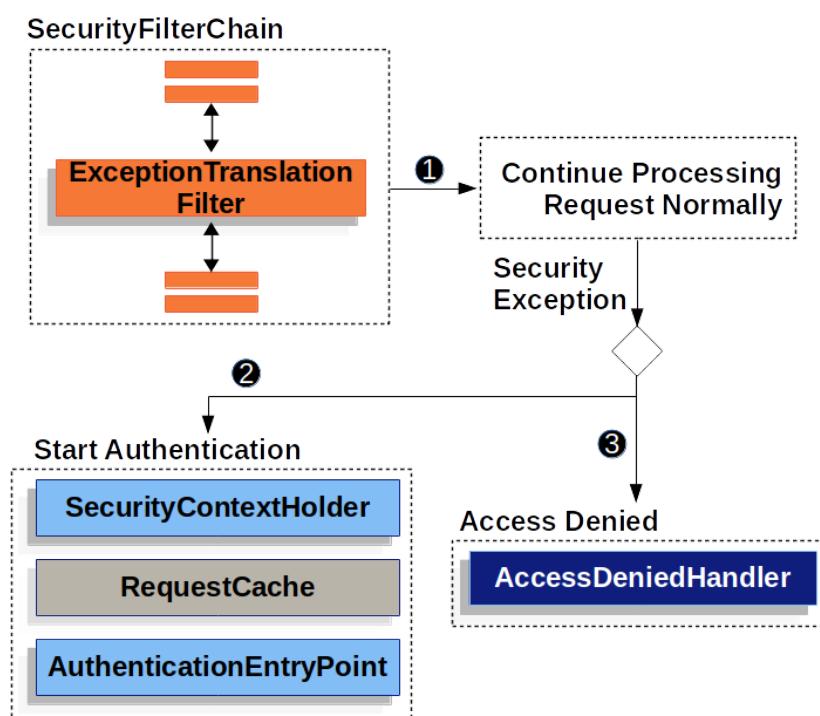
Spring security intercepta las peticiones http antes de que estas lleguen a nuestros controladores, es ahí donde una cadena de filtros con un funcionamiento de middleware encadenado, tiene el poder de decidir si aprobar, rechazar o pasar al siguiente filtro, cuando uno de los filtros aprueba una solicitud, aunque sigue pasando por el resto de filtros, esta solicitud ya pasa como aprobada.



Cuando una solicitud va a ser aprobada en muchas ocasiones se pasa por un AuthenticationManager, que es quien se encarga de comprobar que ese usuario realmente tiene el permiso de hacer esa solicitud, ya sea comprobando en base de datos o de otra manera.

Cuando se decide que todo está en orden, se insertan los detalles del usuario en un objeto que implementa la interfaz Authentication, que posteriormente se guardaran en el SecurityContext, es en ese momento donde se llama al método setAuthentication donde la solicitud está aprobada, si una solicitud es denegada se lanza un AccessDeniedException el cual captura el AccessDeniedHandler devolviendo un 403 Forbidden al cliente.

Posteriormente en cualquier parte de la aplicación (generalmente en el controlador) se puede acceder a ese contexto de seguridad ya sea mediante SecurityContext.getContext como mediante la anotación @AuthenticationPrincipal en los parámetros de la anotación, lo que facilita enormemente el poder operar con las credenciales del usuario.



2. Configuración personalizada y filtros personalizados.

Por defecto Spring Security arranca con una configuración de sesiones y añadiendo un formulario de usuario y contraseña en cualquiera de nuestros endpoints, en la gran mayoría de ocasiones vamos a querer definir que endpoints están protegidos y cuáles no o de que manera vamos a autenticar al usuario.

Para modificar todo esto debemos crear un componente de configuración (@Configuration) y añadir la anotación @EnableWebSecurity.

Dentro de esta clase como en cualquier clase de configuración, definiremos los beans que queremos exponer, en este caso necesitaremos un SecurityFilterChain, PasswordEncoder y AuthenticationManager, este último requiere que el contexto de spring exponga tres beans para funcionar, los cuales son SecurityFilterChain, PasswordEncoder y UserDetailsService.

En nuestro SecurityFilterChain recibimos un builder de HttpSecurity, dentro de él definiremos rutas protegidas, añadiremos o quitaremos filtros y definiremos si usamos o no sesiones entre otras muchas posibles cosas.

En mi caso para el microservicio de pedidos he deshabilitado las sesiones y el filtro de Cross Site Request Forgery que suele ser una gran preocupación cuando usamos sesiones, además he proporcionado un bean definido de Cors para autorizar las peticiones con Header Origin desde mis clientes, he definido que todas las request deben pasar por autenticación y he añadido un filtro del que hablaremos más adelante y que se ejecutará justo antes del de usuario y contraseña.

```
@Bean
public SecurityFilterChain securityFilterChain (HttpSecurity http) throws Exception {
    http.cors(httpSecurityCorsConfigurer ->
        httpSecurityCorsConfigurer.configurationSource(corsConfigurationSource())
        .csrf(AbstractHttpConfigurer::disable)
        .sessionManagement( sesion -> sesion.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

    // todos los endpoint piden autenticación previa
    http.authorizeHttpRequests( req ->
        req.anyRequest()
            .authenticated()
    );

    //filtro previo al de usuario y contraseña
    http.addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

(SecurityFilterChain del microservicio de pedidos)

Como hemos comentado antes, Spring Security nos obliga a definir un bean PasswordEncoder, que es el que usarán por defecto algunos filtros predeterminados para comprobar si una contraseña en base de datos es igual a la que nos mandan cifrada con este algoritmo.

```
@Bean  
public PasswordEncoder passwordEncoder () {  
    return new BCryptPasswordEncoder();  
}
```

En mi caso he escogido Bcrypt porque es un algoritmo seguro y algo lento que permite que 2 usuarios con una misma contraseña no obtengan el mismo hash, ya que bcrypt coge una contraseña y un salt aleatorio, genera un hash y añade al final el salt para poder utilizarse en comparaciones, de esta manera eres inmune a ataques de fuerza bruta global ya que no puedes generar un hash y ver si alguno coincide con 3 millones de registros.

Para finalizar en una clase aparte he definido un servicio que implementa UserDetailsService.

```
@Service  
@AllArgsConstructor  
public class UserDetailsServiceImpl implements UserDetailsService {  
  
    private final UserService userService;  
    1 usage  + adminaitor  
    @Override  
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
        return this.userService.findByUsername(username)  
            .orElseThrow( () -> new UsernameNotFoundException(username + " no encontrado"));  
    }  
}
```

Una vez tenemos nuestros tres beans configurados, podemos crear en nuestra clase de configuración el Bean AutenticationManager de esta forma.

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

Anteriormente el método recibía un AuthenticationManagerBuilder donde se le proporcionaba el UserDetailsService y PasswordEncoder pero este método se obsoletizó a favor de una configuración más automatizada y abstracta.

En nuestro SecurityFilterChain habíamos definido un JwtFilter que se ejecutará antes del filtro de usuario y contraseña, este JwtFilter tiene que ser un componente que extienda OncePerRequestFilter que es la clase de la que extienden todos los filtros.

```
3 usages 39 inheritors
public abstract class OncePerRequestFilter extends GenericFilterBean {

    © JwtFilter (com.cafeteria.ventura.orders.security.config)
    © AbstractRequestLoggingFilter (org.springframework.web.filter)
    © Anonymous in LoginPageFilter (org.springframework.boot.web.servlet.su
    © ApplicationContextHeaderFilter (org.springframework.boot.web.servlet.
    © AuthenticationFilter (org.springframework.security.web.authentication
    © BasicAuthenticationFilter (org.springframework.security.web.authenticat
    © CharacterEncodingFilter (org.springframework.web.filter)
    © CommonsRequestLoggingFilter (org.springframework.web.filter)
    © CorsFilter (org.springframework.web.filter)
    © CsrfFilter (org.springframework.security.web.csrf)
    © DefaultLogoutPageGeneratingFilter (org.springframework.security.web.a
    © DefaultOneTimeTokenSubmitPageGeneratingFilter (org.springframework.se
    © DefaultWebAuthnRegistrationPageGeneratingFilter (org.springframework.
```

Sobreescribiendo el método doFilterInternal, debemos definir en qué condición almacenaremos un Authentication en el contexto de seguridad, de esta manera aprobamos la solicitud y pasamos a los siguientes filtros.

```
@Override
protected void doFilterInternal(
    HttpServletRequest request,
    HttpServletResponse response,
    FilterChain filterChain) throws ServletException, IOException {

    String token = this.extractToken(request);

    if (this.tokenProvider.isValidToken(token)) {
        //extrae el nombre de usuario y trae sus detalles de la db
        String username = this.tokenProvider.getUsernameFromToken(token);
        UserDetails user = this.userService.loadUserByUsername(username);

        //almacenamos el UserDetails completo para poder acceder a él en los controladores con la anotación @AuthenticationPrincipal
        Authentication auth = new UsernamePasswordAuthenticationToken(
            user,
            credentials: null,
            user.getAuthorities());

        //almacenamos en el contexto de seguridad (para extraer más tarde donde necesitemos) y autenticamos
        SecurityContextHolder.getContext().setAuthentication(auth);
    }

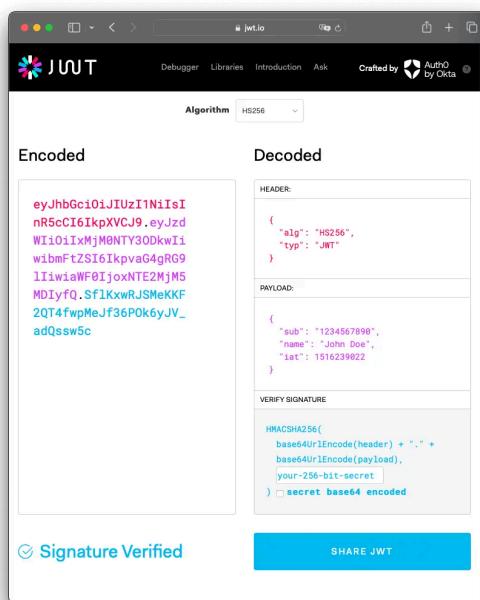
    //continuamos la chain de las peticiones
    filterChain.doFilter(request, response);
}
```

3. El estándar JSON Web Tokens y su implementación

Los tokens JWT son un estándar que nos permite tanto autenticar clientes como intercambiar información de forma autenticada entre pares, estos tokens se componen de:

- Header - Un Json codificado en base64 que nos dice el tipo y algoritmo de cifrado en dos campos “typ” y “alg”.
- Payload - La carga útil/contenido del token, aquí es donde guardamos la información que queremos intercambiar, cualquier cosa que necesitemos a posteriori, estos se definen como claims y los hay de distintos tipos.
 - Registrados - Campos predefinidos que aunque no son obligatorios son bastante recomendables, como cuando expira este token, quien lo expide, su asunto.
 - Públicos - Siguen una convención <https://www.iana.org/assignments/jwt/jwt.xhtml>.
 - Privados - Aquí es donde mandamos esta información acordada con el resto de servicios o con nosotros mismos, desde campos como usuario o rol a cualquier tipo de información que queramos mandar.
- Signature - La firma del JWT se genera cifrando con una clave privada en un algoritmo Hmac o RSA el header en base64 un punto “.” y el payload en base64, si alguien cambia el Payload y no tiene la clave privada no podrá regenerar la firma y este token no será válido.

Es importante recalcar que tanto el header como el payload van en base64 al descubierto, no se debería mandar información confidencial en estos tokens, si existiese esa necesidad se pueden usar estándares como JWE, los tokens JWT que usan JWS solo nos dan la seguridad de que enviamos a otro servidor o nos envían desde el cliente información que no ha sido alterada.



(Imagen de JWT.io)

A la hora de generar los tokens en un login en nuestro microservicio de autenticación, recibimos un usuario y contraseña, como este endpoint para logear debe ser público y sin restricciones, no pasamos por una cadena de filtros, esto no significa que no podamos autenticar la solicitud ya que podemos invocar al authenticationManager, aquí cobra sentido el haber configurado antes un PasswordEncoder y un UserDetailsService para que este authenticationManager los use por debajo.

```
@PostMapping("/login")
public LoginResponse login(@RequestBody LoginRequest loginDTO) {
    Authentication authDTO = new UsernamePasswordAuthenticationToken(loginDTO.getUsername(), loginDTO.getPassword());

    Authentication authentication = this.authManager.authenticate(authDTO);
    UserEntity user = (UserEntity) authentication.getPrincipal();

    String token = this.jwtTokenProvider.generateToken(authentication);

    return new LoginResponse(
        user.getUsername(),
        user.getAuthorities().stream().map(GrantedAuthority::getAuthority).toList(),
        token);
}
```

Ya que nuestro manager debe saber el encoder para comparar la contraseña y debe saber cómo acceder a la base de datos, en este caso desde nuestro servicio.

Una vez ya hemos autenticado la solicitud, podemos ver que se llama al método generateToken de un componente JwtTokenProvider, este método nos permite generar un jwt bien formado apoyándonos en las librerías de jsonwebtoken para Spring.

```
public String generateToken(Authentication authentication) {
    // trae el usuario del contexto en el que lo guardamos en doFilterInternal
    UserEntity user = (UserEntity) authentication.getPrincipal();

    return Jwts.builder()
        .signWith(Keys.hmacShaKeyFor(jwtSecret.getBytes()), SignatureAlgorithm.HS512)
        .setHeaderParam("typ", "JWT") //type JWT estandar
        .setSubject(Long.toString(user.getIdUser()))
        .setIssuedAt(new Date())
        .setExpiration(new Date(System.currentTimeMillis() + (jwtDurationSeconds * 1000)))
        .claim("username", user.getUsername()) //guarda username, email y authorities en el map de claims
        .claim("email", user.getEmail())
        .claim("authorities", user.getAuthorities().stream() Stream<capture of extends GrantedAuthority>
            .map(GrantedAuthority::getAuthority) Stream<String>
            .toList())
        .compact();
}
```

Es importante almacenar en variables de entorno las claves secretas para que estas no sean expuestas y cualquiera pueda vulnerar los JWT dando como resultado en un acceso fraudulento a cualquier usuario.

```
@Component
public class JwtTokenProvider {

    // variables de entorno de resources/application.yml
    5 usages
    @Value("${app.security.jwt.secret}")
    private String jwtSecret;

    2 usages
    @Value("${app.security.jwt.expiration}")
    private Long jwtDurationSeconds;
```

4. Schedule y tareas programadas

Dado que en el flujo de programa el microservicio de pedidos debe comunicarse con el microservicio Nestjs de Websockets y este no puede estar en red privada, ya que tiene que comunicarse con los terminales cliente de la cafetería, tuvimos que encontrar una manera de que autenticar nuestro microservicio, ya que no queremos que cualquier persona pueda hacer solicitudes POST a NEST mandando pedidos fraudulentos, en este caso programamos en el microservicio de pedidos dos componentes, uno de ellos es simple expone una variable donde va el token y nos permite injectar y actualizar el token y el otro tiene una tarea programada que se ejecuta al iniciar la aplicación y cuando ha pasado un 90% del tiempo de caducidad del token.

Esta tarea consiste en realizar una solicitud de login con un usuario con rol SYSTEM y actualizar el token que inyecta el otro componente.

```
@Slf4j
@RequiredArgsConstructor
@Getter
@EnableScheduling
@Component
public class RefreshSystemJwtScheduler {

    private final SpringAuthWebhook authWebhook;
    private final SystemJwt systemJwt;

    /**
     * Refresca el JWT de sistema antes de que expire
     *
     * @throws CustomException si hay un fallo de comunicación con el microservicio de autenticación
     */
    @Scheduled(fixedRateString = "${app.security.jwt.refresh-rate-ms}")
    private void refreshSystemJwt() throws CustomException {
        String jwtToken = this.authWebhook.getSystemJwtToken();
        this.systemJwt.setSystemJwt(jwtToken);
        log.info("Token de sistema refrescado");
    }

    /**
     * Ejecuta la tarea al iniciar el programa
     *
     * @throws CustomException si hay un fallo de comunicación con el microservicio de autenticación
     */
    @PostConstruct
    private void initSystemJwt() throws CustomException {
        log.info("Inicializando Token de sistema");
        refreshSystemJwt();
    }
}
```

```
@Component
@Getter
@Setter
public class SystemJwt {
    private String systemJwt;
}
```

5. Webhooks

Para obtener el token del microservicio de autenticación se utiliza un servicio que inyecta desde las variables del application.yml la url del microservicio de autenticación y el usuario y contraseña de este usuario con rol SYSTEM.

```
@Slf4j
@Service
public class SpringAuthWebhook {

    1 usage
    @Value("${app.webhook.spring-auth}")
    private String springAuthServiceUrl;

    1 usage
    @Value("${app.system-user-details.user}")
    private String systemUser;
    1 usage
    @Value("${app.system-user-details.password}")
    private String systemPassword;
```

Se emplea un método privado que construye un RestClient (Opción moderna y recomendada para hacer peticiones síncronas que ha reemplazado a RestTemplate).

```
private RestClient generateRestClient() {
    //forzamos HTTP_1_1
    HttpClient httpClient = HttpClient.newBuilder()
        .version(HttpClient.Version.HTTP_1_1)
        .connectTimeout(Duration.ofSeconds(5))
        .build();

    return RestClient.builder()
        .requestFactory(new JdkClientHttpRequestFactory(httpClient))
        .baseUrl(this.springAuthServiceUrl)
        .build();
}
```

Teniendo el cliente listo con la url base cargada, ejecutamos una petición.

```
/**  
 * Hace una petición al microservicio de autenticación con credenciales de system  
 * tras una serie de validaciones devuelve el token  
 *  
 * @return jwt de usuario con rol System  
 * @throws CustomException si hay un fallo de comunicación con el microservicio de autenticación  
 */  
1 usage  ± adminaitor  
public String getSystemJwtToken() throws CustomException {  
  
    LoginRequest request = new LoginRequest(systemUser, systemPassword);  
  
    log.info("Lanzando petición al microservicio de autenticación");  
    RestClient client = generateRestClient();  
    ResponseEntity<LoginResponse> response = client.post() RequestBodyUriSpec  
        .uri( uri: "/auth/login" ) RequestBodySpec  
        .contentType(MediaType.APPLICATION_JSON)  
        .body(request)  
        .accept(MediaType.APPLICATION_JSON)  
        .retrieve() ResponseSpec  
        .toEntity(LoginResponse.class);  
    log.info("Status code de la respuesta, Status code: {}", response.getStatusCode().value());
```

Este proceso es crítico ya que si falla no hay comunicación con NEST y no se procesan los pedidos, es por ello que el microservicio no arranca si no hay una respuesta o si el token obtenido no tiene rol SYSTEM.

```
if (response.getStatusCode().isError()) {  
    throw new CustomException("Ha habido un error interno de comunicación: AUTH_SERVICE_UNAVAILABLE, " +  
        "por favor, contacta con los administradores", HttpStatus.INTERNAL_SERVER_ERROR);  
}  
  
LoginResponse body = response.getBody();  
  
if (!body.authorities().contains(UserAuthority.SYSTEM)) {  
    throw new CustomException("Ha habido un error interno de comunicación: SYSTEM_USER_DONT_EXIST, " +  
        "por favor, contacta con los administradores", HttpStatus.INTERNAL_SERVER_ERROR);  
}  
return body.token();  
}
```

Este JWT es usado en el header Authorization en envíos a NEST.

```
public ResponseEntity<NestWebhookWebSocketEmitResponse> sendOrderToNest(OrderDTO order, String username) {
    RestClient client = generateRestClient();

    NestWebhookWebSocketEmitRequest nestWebhookWebSocketEmitRequest = NestWebhookWebSocketEmitRequest.builder()
        .id(order.getId())
        .comprador(username)
        .descripcion(order.getDescription())
        .pagado(order.isPaid())
        .productos(
            order.getProducts().stream() Stream<OrderProductDTO>
                .map(product -> new NestProductDetails(product.getName(), product.getQuantity())) Stream<NestProductDetails>
                .collect(Collectors.toSet())
        )
        .build();

    return client.post() RequestBodyUriSpec
        .uri( uri: "/websocket/emit" ) RequestBodySpec
        .header( headerName: "Authorization", ...headerValues: "Bearer " + this.systemJwt.getSystemJwt() )
        .contentType(MediaType.APPLICATION_JSON)
        .body(nestWebhookWebSocketEmitRequest)
        .accept(MediaType.APPLICATION_JSON)
        .retrieve() ResponseSpec
        .toEntity(NestWebhookWebSocketEmitResponse.class);
}
```

El cual NEST comprueba en un servicio y solo aprueba el envío pedido si el rol es SYSTEM.

```
@Controller('websocket')
export class WebSocketController {
    constructor(private readonly websocketService: WebSocketService) {}

    @Post('emit')
    @HttpCode(HttpStatus.OK)
    emitMessage(@Req() request: FastifyRequest, @Body() data: unknown) {
        // Verificar el JWT
        if (!this.websocketService.verifySystemJwt(request)) {
            throw new HttpException('Unauthorized', HttpStatus.UNAUTHORIZED); // 401
        }
        const result: Result = this.websocketService.processMessage(data);

        if (!result.success) {
            throw new HttpException(result.error as string, HttpStatus.BAD_REQUEST); // 400
        }

        return {
            data: result,
        };
    }
}
```

Extrae el token eliminando “Bearer ” lo verifica con la clave privada compartida y devuelve true si en los roles encuentra SYSTEM.

```
verifySystemJwt(request: FastifyRequest): boolean {
  //extraer el token del header
  const token = request.headers['authorization']?.split(' ')[1];
  if (!token) {
    return false;
  }

  //verificar la clave secreta
  const secret = this.configService.get<string>('VITE_JWT_SECRET');
  if (!secret) {
    throw new Error('Clave secreta de JWT no definida');
  }

  //verificar el token
  let tokenData: JwtUserDetailsAuthorities;
  try {
    tokenData = jwt.verify(token, secret) as JwtUserDetailsAuthorities;
  } catch (err) {
    console.error('Token inválido:', err);
    return false;
  }

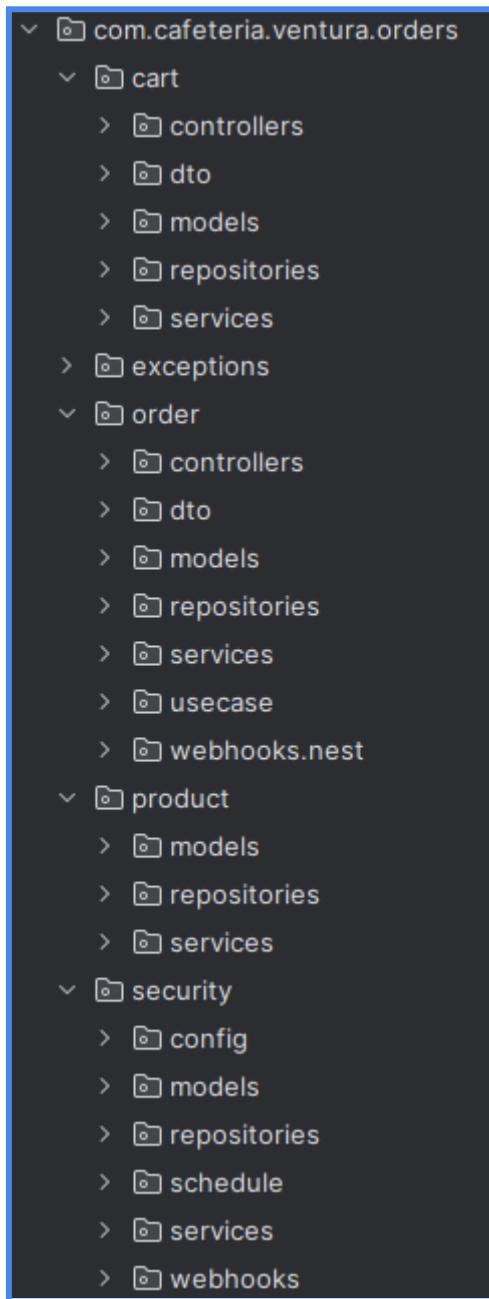
  return (
    tokenData.authorities?.some((authority) => authority === 'SYSTEM') ??
  );
}
```

ANEXO B - ARQUITECTURA DE PAQUETES

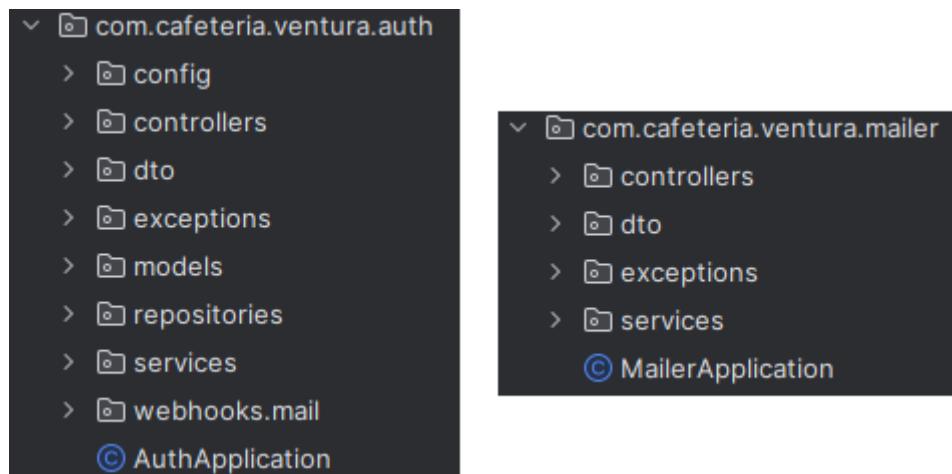
1. Backend

Hemos utilizado Screaming Architecture para hacer un código más mantenible y escalable, esta arquitectura se basa en que los paquetes te deben de gritar lo que hacen, en vez de tener un MVC tradicional con un paquete de controller, otro de service y otro de repository más el paquete de excepciones y utilidades, Screaming Architecture propone tener un paquete de lógica de negocio y dentro sus controllers, services, repositories...

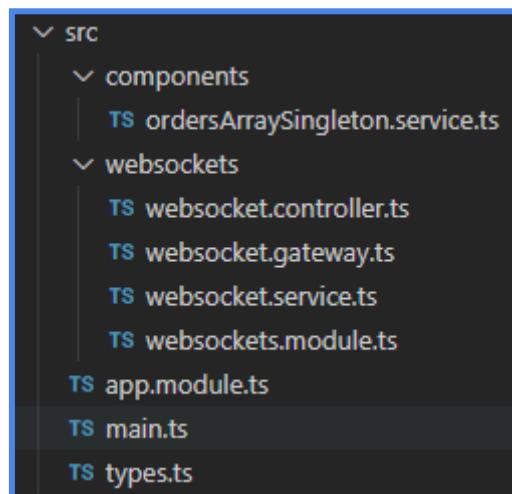
Hemos ido adaptando esta arquitectura combinándola con el uso de useCase propio de Clean Architecture, todo en su justa medida para tampoco pecar de sobre ingeniería en los que no merece la pena.



Dado que el microservicio de autenticación y el de mail no eran tan grandes hemos respetado el MVC tradicional.

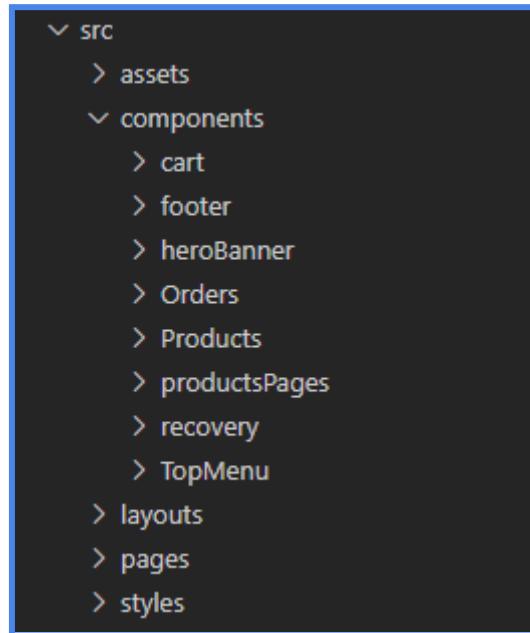


Para NEST he respetado la arquitectura recomendada.

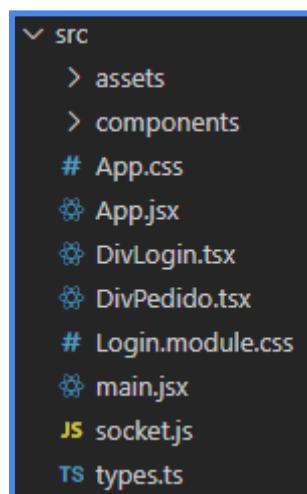


2. Frontend

Para la web principal se ha respetado la arquitectura que usa Astro Framework ya que los paquetes son semánticos y crear un componente en la carpeta pages crea a su vez una web.



Mientras que la web de react al ser una Single Page Application tan pequeña está casi todo a nivel de raíz de src.



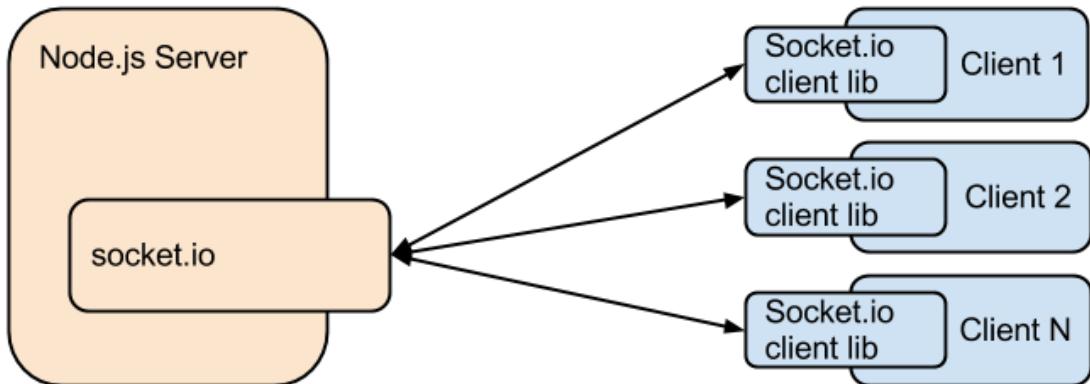


1. Por qué usar SOCKET.IO en vez de Websockets tradicionales

Socket.io es un protocolo que utiliza websockets por debajo pero no es compatible con el protocolo de conexión estándar de Websockets, es decir, tanto cliente como servidor deben utilizar las librerías oficiales del protocolo.

Utilizar Socket.io tiene varias ventajas, obtienes total compatibilidad con navegadores antiguos, en caso de que websockets esté restringido a nivel de proxy se cambia automáticamente a Long Polling con un rendimiento similar, trae características ya implementadas que a menudo se terminan necesitando en cualquier servicio de Websockets y de las cuales hacemos uso, como pueden ser las reconexiones automáticas, los grupos de mensajes y los middlewares de autenticación.

De esta manera terminamos desarrollando un sistema seguro con autenticación, que nos permite recibir pedidos en tiempo real en todos los clientes a la vez y actualizar la finalización de un pedido en tiempo real en todos ellos.



(Imagen de Medium)

2. Por qué Nestjs con Fastify en vez de Spring

Socket.io está desarrollado inicialmente para backends Node, aunque a día de hoy tiene una librería para Spring Framework la velocidad es hasta 4 veces menor, por eso no veíamos necesidad de que para un microservicio sin una lógica muy compleja, como es recibir pedidos, transmitirlos a los clientes, recibir aprobaciones, enviar la confirmación al resto y para finalizar hacer un update de un estado en la base de datos, sacar la librería del entorno en el que más rinde no tenía sentido, además utilizar el adaptador de Fastify en vez de Express hace que Nestjs se vuelva uno de los frameworks con mayor velocidad y menor consumo de recursos, lo que lo hace ideal para tareas como esta.

Otro de los puntos a favor es que Nestjs te obliga a desarrollar puramente en TypeScript y con inyección de dependencias, lo que hace que el salto desde Spring no sea tan abrumador.



3. Implementación en el Backend

Con apenas 100 líneas de código tienes todos los controladores necesarios para manejar la conexión, desconexión y middleware de autenticación exclusiva con tokens con rol de Admin.

```
@WebSocketGateway({
    // Cors del cliente React
    cors: {
        origin: '*', //TODO: Cambiar a la url cuando se defina donde esta el cliente
        methods: ['GET', 'POST'],
        allowedHeaders: ['Content-Type'],
    },
})
export class WebsocketGateway
    implements OnGatewayConnection, OnGatewayDisconnect
{
    @WebSocketServer()
    server: Server;

    constructor(
        private readonly websocketService: WebSocketService,
        private readonly ordersArray: OrdersArrayService,
    ) {}

    afterInit(server: Server) {
        this.websocketService.setServer(server);
        // Verifica que el token JWT sea válido y tenga el rol de ADMIN
        server.use((socket, next) => {
            try {
                const valid = this.websocketService.verifyAdminJwt(socket);
                if (!valid) {
                    return next(new Error('Authentication error'));
                }
                next();
            } catch (error) { // 'error' is defined but never used.
                next(new Error('Authentication error')); //error al cliente
            }
        });
    }

    handleConnection(client: Socket) {
        console.log(`Cliente conectado: ${client.id}`);
        //envia todos los componentes almacenados al iniciar
        setTimeout(() => {
            this.ordersArray
                .getOrders()
                .forEach(order =>
                    this.websocketService.processInitialMessage(client, order),
                );
        }, 1000);
    }

    handleDisconnect(client: Socket) {
        console.log(`Cliente desconectado: ${client.id}`);
    }

    @SubscribeMessage('removeOrder')
    handleRemoveOrder(@MessageBody() id: number) {
        return this.websocketService.processRemoveOrder(id);
    }
}
```

Además implementamos un componente que es un array de objetos que nos permite enviar todos los pedidos pendientes a las nuevas conexiones a la aplicación.

```
@Injectable()
export class OrdersArrayService {
  private orders: Pedido[] = [];

  getOrders() {
    console.log('contenido del array:\n', this.orders);
    return this.orders;
  }

  addOrder(order: Pedido) {
    this.orders.push(order);
  }

  removeOrderById(id: number) {
    this.orders = this.orders.filter((order) => order.id != id);
  }
}
```

Reutilizamos la comprobación de token SYSTEM para construir de forma casi idéntica la de token ADMIN.

```
verifyAdminJwt(client: Socket): boolean {
  //extraer el token del objeto de autenticación del cliente
  console.log('Verificando token JWT de ADMIN - ' + client.id);
  const token = client.handshake.auth?.token as string | undefined;
  if (!token) {
    return false;
  }

  //verificar la clave secreta
  const secret = this.configService.get<string>('VITE_JWT_SECRET');
  if (!secret) {
    throw new Error('Clave secreta de JWT no definida');
  }

  //verificar el token
  let tokenData: JwtUserDetailsAuthorities;
  try {
    tokenData = jwt.verify(token, secret) as JwtUserDetailsAuthorities;
  } catch (err) {
    console.error('Token inválido:', err);
    return false;
  }
  console.log('Token verificado:', tokenData);
  return (
    tokenData.authorities?.some((authority) => authority === 'ADMIN') ?? false
  );
}
```

Manejamos el envío y procesamiento de estos pedidos en nuestro servicio usando una ORM prisma para actualizar el estado en base de datos.

```
processMessage(data: unknown): Result {
    console.log('Procesando mensaje:\n', data);

    try {
        const message: Pedido = PedidoSchema.parse(data);
        this.server.emit('mensajeServer', message);
        this.ordersArray.addOrder(message);
        return { success: true } as Result;
    } catch (error: unknown) {
        console.error('Error al procesar el mensaje:', error);
        const errorMessage =
            error instanceof Error ? error.message : 'Error desconocido';
        return { success: false, error: errorMessage } as Result;
    }
}

processRemoveOrder(id: number): Result {
    console.log('procesando pedido concluido:\nID:', id);

    //hacer query a db actualizando el estado
    this.prisma.orders
        .update({
            where: { id_order: id },
            data: { state: 'FINALIZADO' },
        })
        .then(() => {
            console.log(`Pedido ${id} actualizado a FINALIZADO en la base de datos`);
            this.server.emit('removeOrderServer', id);
            this.ordersArray.removerOrderById(id);
        });

    return { success: true } as Result;
}

processInitialMessage(client: Socket, order: unknown): Result {
    console.log('Procesando mensaje inicial:\n', order);

    try {
        const message: Pedido = PedidoSchema.parse(order);
        client.emit('mensajeServer', message);
        return { success: true } as Result;
    } catch (error: unknown) {
        console.error('Error al procesar el mensaje:', error);
        const errorMessage =
            error instanceof Error ? error.message : 'Error desconocido';
        return { success: false, error: errorMessage } as Result;
    }
}
```

4. Implementación en el frontend

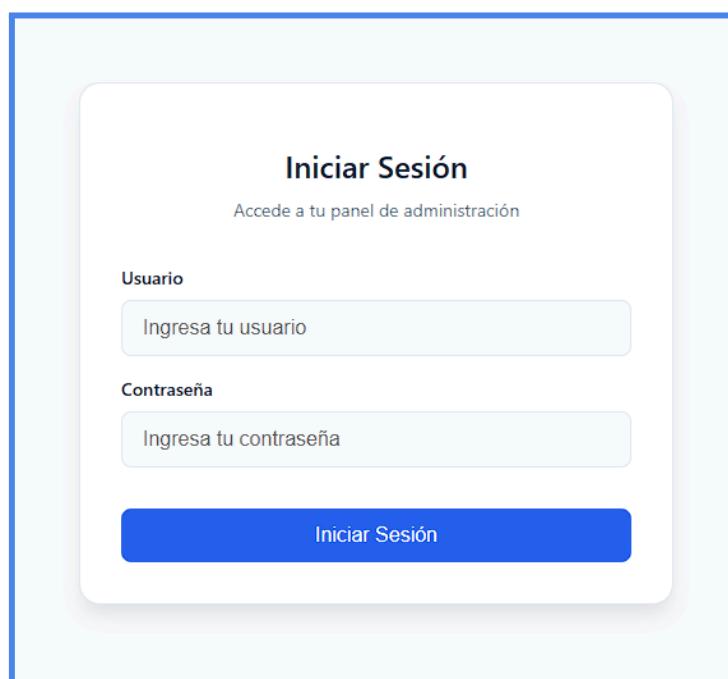
Para el frontend decidimos crear una Single Page Application (SPA) utilizando React debido al Virtual DOM y la optimización que tiene gracias a este a la hora de crear y actualizar componentes, además de la compatibilidad que tiene con la propia librería y la mantenibilidad que aporta el código cliente usando hooks y componentes.

No utilizamos Astro en esta página como en la otra debido a que la gran mayoría de contenido se iba a renderizar de forma dinámica con Client Side Rendering.

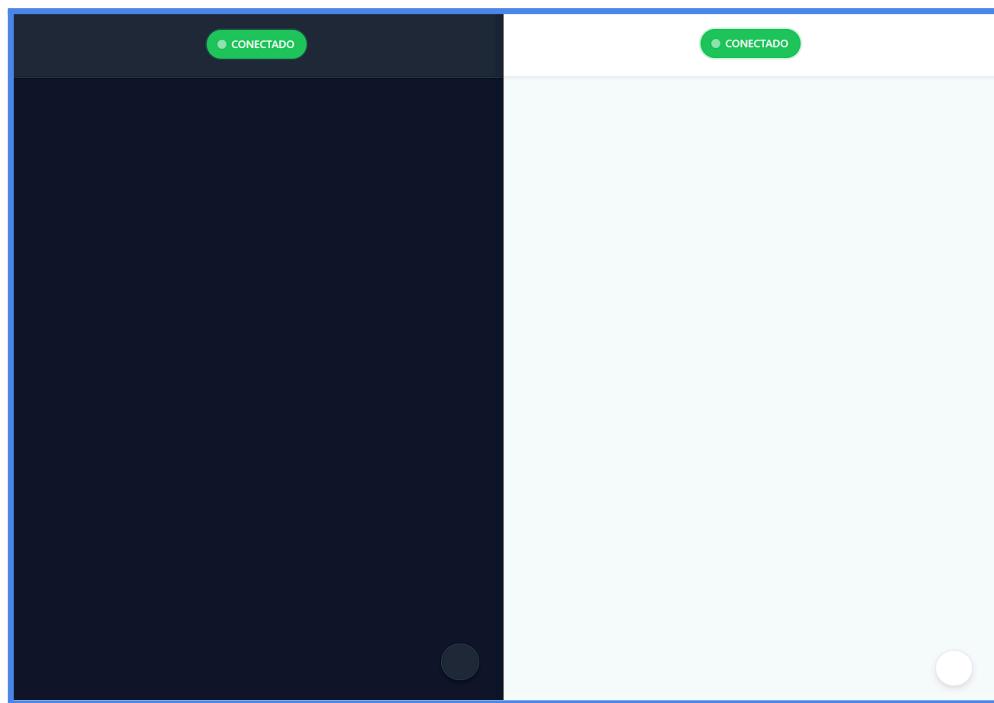
En un archivo de JavaScript puro definimos cómo se realiza la conexión al socket y los reintentos.

```
1 import { io } from "socket.io-client";
2
3 const URL = import.meta.env.VITE_URL_NEST;
4
5 let socket = null;
6
7 export const getSocket = () => {
8   if (!socket) {
9     socket = io(URL, {
10       auth: {
11         token: sessionStorage.getItem('token') || '',
12       },
13       reconnectionAttempts: Infinity, // reconexión infinita
14       reconnectionDelay: 1000, // cada segundo
15     });
16
17     socket.on('connect_error', (err) => {
18       console.error(`Connection error: ${err.message}`);
19     });
20   }
21
22   return socket;
23 };
```

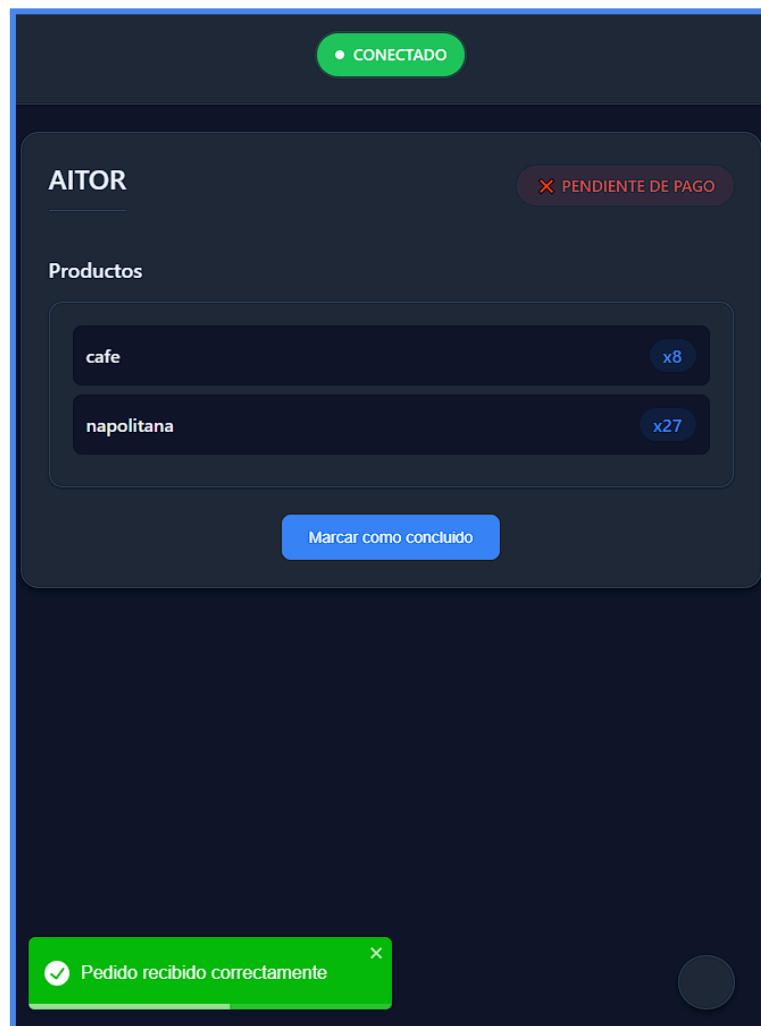
Como vemos este archivo extrae un token del sessionStorage que conseguiremos al lanzar una query al microservicio de pedidos, sólo si este tiene un rol de ADMIN.



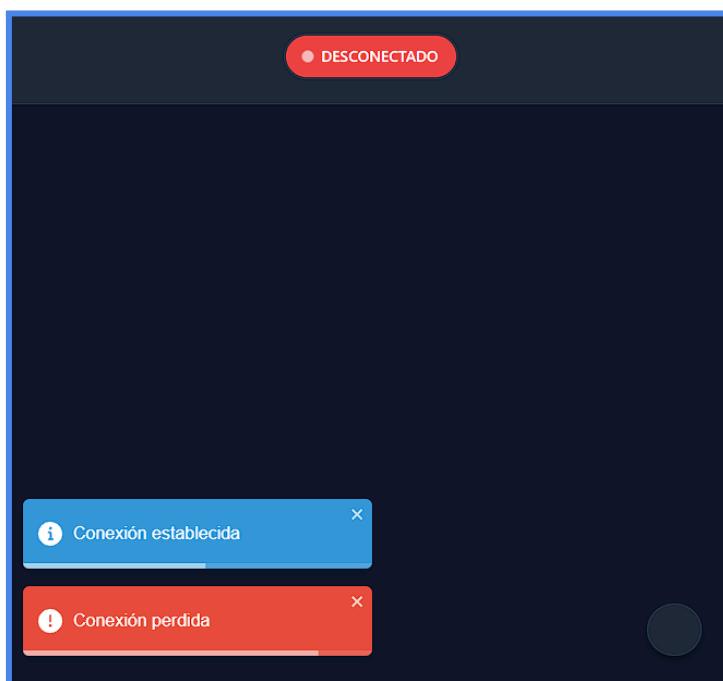
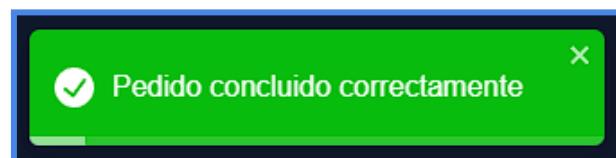
Una vez se inicia sesión aparece este menú de pedidos junto a un botón que indica si estamos o no conectados al servicio y un botón para alternar entre modo claro y modo oscuro.



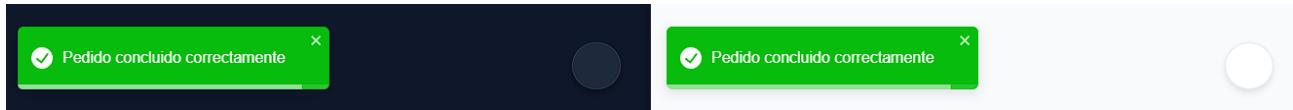
Cuando recibimos un pedido aparece un toast informandonos.



Estos toast también nos informan de pedidos concluidos, conexiones y desconexiones.



Los estados de conexiones son propios de cada dispositivo pero como los pedidos llegan y se concluyen en todos a la vez, el resto de terminales cliente pueden ver que otro compañero ha concluido un pedido y que este desaparece de la interfaz.



Estos cuatro eventos ocurren dentro de un hook useEffect que se monta al hacer login y se monta y desmonta la suscripción de estos eventos según el estado de conexión.

```
useEffect( () => {
  // no monta el socket si no está logueado
  if (!isLogged) {
    return;
  }

  const socket = getSocket();

  setIsConnected(socket.connected);

  function onConnect() {
    toast.info("Conexión establecida");
    setIsConnected(true);
  }

  function onDisconnect() {
    toast.error("Conexión perdida");
    setIsConnected(false);
  }

  function onMessageEvents(newMessage) {
    //crea un nuevo array con el contenido del anterior y el nuevo valor (react requiere que sea inmutable)
    setMensajeEvents(previous => [...previous, newMessage]);
    toast.success("Pedido recibido correctamente");
  }

  // elimina el pedido correspondiente a id de la visualización
  function onRemoveOrderServer(id) {

    //setea fading=true en el elemento a eliminar
    setMensajeEvents( (prevOrders) => {
      return prevOrders.map( (order) => order.id === id ? { ...order, fading: true} : order);
    });

    //espera un segundo y elimina el elemento del array
    setTimeout( () => {
      setMensajeEvents(previousOrders => previousOrders.filter(order => order.id != id));
    }, 1000);

    // muestra una alerta exitosa
    toast.success("Pedido concluido correctamente");
  }

  //registra en cada evento cada función
  socket.on("connect", onConnect);
  socket.on("disconnect", onDisconnect);
  socket.on("mensajeServer", onMessageEvents); //evento mensajeServer crear elemento
  socket.on("removeOrderServer", onRemoveOrderServer); // evento concluir pedido

  //limpia los eventos al desmontarse
  return () => {
    socket.off("connect", onConnect);
    socket.off("disconnect", onDisconnect);
    socket.off("mensajeServer", onMessageEvents);
    socket.off("removeOrderServer", onRemoveOrderServer); // evento concluir pedido
  }
}, [isLogged]);
```

ANEXO D - ASTRO

1. Por qué usar Astro

Astro es un generador de páginas estáticas agnóstico a frameworks de js, ¿Qué significa esto? astro permite el uso de javaScript en tiempo de compilación para la generación de páginas web estáticas es decir, compila clientes desde el lado del servidor como podría hacer PHP pero salvando las distancias.

Cuando se utiliza PHP estamos utilizando Server Side Rendering, es decir, un cliente hace una petición, se renderiza una página en el servidor para ese cliente y se envía, Astro hace una cosa totalmente distinta, usa javaScript para compilar una sola vez una web para servir a todos los clientes, esto nos aporta velocidad y menos gasto de recursos sin renunciar al SEO que nos aportaba el SSR, además al ser páginas completamente estáticas no tenemos que preocuparnos de la eficiencia del JS de nuestro servidor a la hora de compilar ya que se traduce en HTML.

Astro también nos permite utilizar una arquitectura de componentes como la de muchos frameworks de frontend pero con la particularidad de que estos componentes pueden ser solo de HTML y CSS, todo el JS que se ejecute entre la separación de tres guiones (frontmatter) será ejecutado en compilación, de igual manera que todo el que se ejecute entre llaves {}.

```
---
import Layout from '../layouts/Layout.astro';
import TopMenu from '../components/TopMenu/TopMenu.astro';
import HomeHero from '../components/heroBanner/HomeHero.astro';
import Products from '../components/Products/Products.astro';
import Footer from '../components/footer/Footer.astro';
---

<Layout>
  <TopMenu />
  <HomeHero />
  <main>
    <Products />
  </main>
  <Footer />
</Layout>

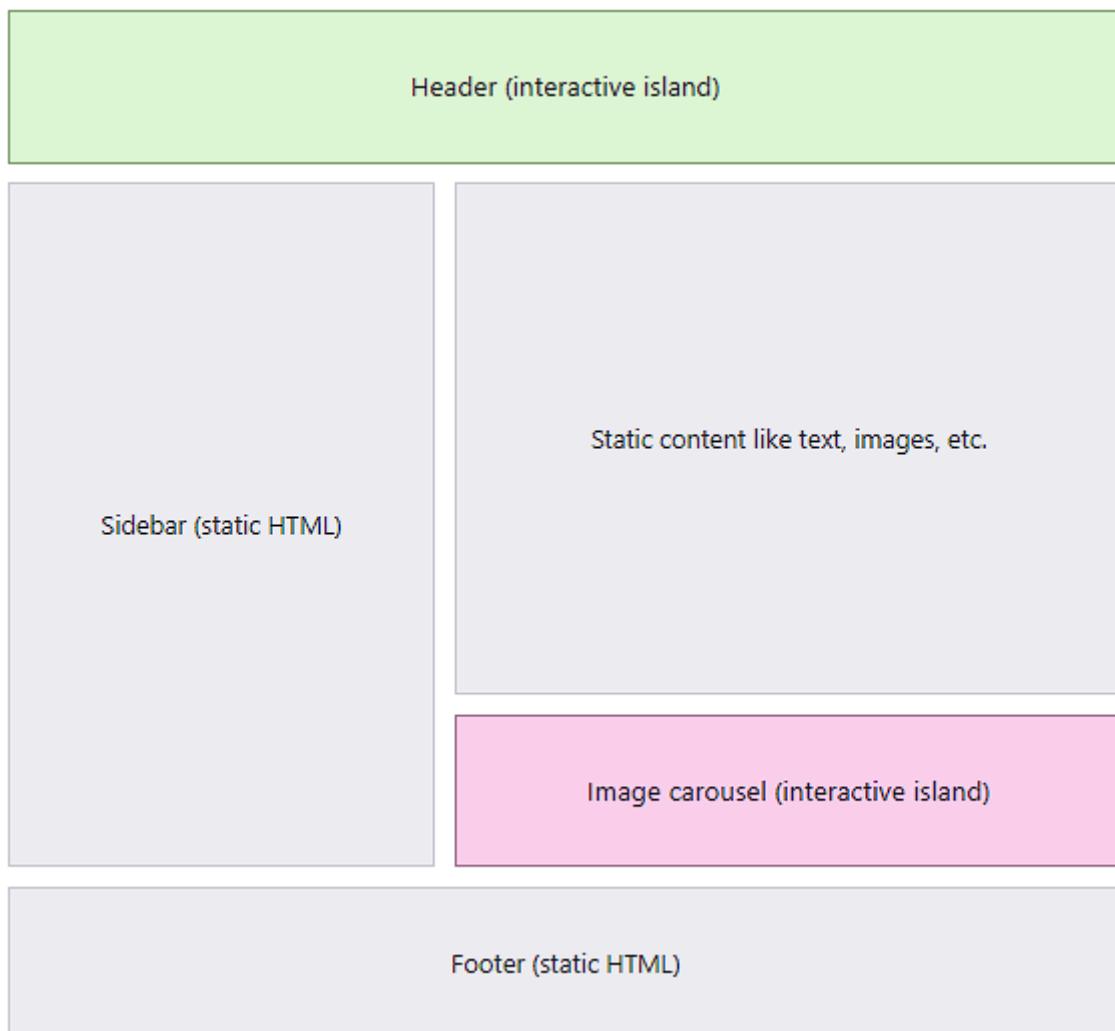
<style>
  body {
    background-color: #aliceblue;
  }

  main {
    min-height: 100vh;
  }
</style>
```

2. Hidratación de JavaScript en Islas

Antes hemos comparado Astro con el SSR de un PHP pero también lo podemos comparar con el Client Side Rendering que tiene, React, Angular, Vue, Svelte y muchos otros más, hasta ahora el CSR tenía ventajas y desventajas muy claras, la principal ventaja es que eliminás mucho gasto del servidor y las principales desventajas son que el SEO hasta hace poco no sabía bien cómo indexar la página, al final estás mandando instrucciones al cliente en forma de JS sobre como montar la UI, lo que lo hace más lento y hace que algunos clientes abandonen o se desesperen si tienen terminales poco potentes.

Astro introduce un modelo de hidratación de JavaScript en forma de islas agnósticas a cualquier framework, es decir, renderiza de forma estática todo lo que no necesite ser dinámico y aísla en islas componentes dinámicos de JavaScript o cualquiera de sus frameworks, esto hace que pueda construir una UI completamente estática pero que cargue un historial de pedidos de una petición al servidor.



En nuestro caso utilizamos el framework React para construir esta interactividad junto a alguna isla de javaScript puro también.

```
---
```

```
import OrdersFetch from "./OrdersFetch";
const apiOrdersUrl = import.meta.env.VITE_API_ORDERS_URL;
---
```

```
<main>
  <OrdersFetch apiOrdersUrl={apiOrdersUrl} client:only="react" />
</main>
```

```
<style>
  main {
    margin: 30px;
    min-height: 100vh;

    display: flex;
    flex-direction: column;
    gap: 10px;
    padding: 10px;
    box-sizing: border-box;
  }
}
```

```
export default function OrdersFetch({ apiOrdersUrl }: { apiOrdersUrl: string }) {
  const sessionStorageToken = sessionStorage.getItem("token");
  const [loading, setLoading] = useState(true);
  const [orders, setOrders] = React.useState<OrdersResponse | null>(null);
  const [page, setPage] = useState("");

  useEffect(() => {
    // extrae el parámetro de la url
    const urlParams = new URLSearchParams(window.location.search);
    setPage(urlParams.get("page") || "");
    console.log(page);
  }, []);

  useEffect(() => {
    const fetchOrders = async () => {
      // ¿Existe el token de sesión? Si existe, se ejecuta el fetch.
      if (sessionStorageToken) {
        // Se establece el estado de carga a true para mostrar un loading.
        setLoading(true);
        // Intentar ejecutar el fetch.
        try {
          // Se ejecuta el fetch a la API de pedidos usando el token de sesión.
          const response = await fetch(apiOrdersUrl + "/order/history" + "?page=" + page, {
            headers: {
              authorization: `Bearer ${sessionStorageToken}`,
            },
          });
          // Si el fetch devuelve un error 403, se elimina el token de sesión.
          if (response.status === 403) {
            // Se elimina el token de sesión.
            sessionStorage.removeItem("token");
            // Se muestra un mensaje de error.
            console.error("Token eliminado debido a un error de autenticación.");
            return;
          }
          // Se convierte la respuesta a un objeto JSON.
          const data = await response.json();
          // Se muestra el objeto JSON en la consola.
          console.log(data);
          // Se establece el estado de los pedidos.
          setOrders(data);
        } catch (error) {
          // Se muestra un mensaje de error en la consola.
          console.error("Error fetching orders:", error);
        } finally {
          // Se establece el estado de carga a false para ocultar el loading.
          setLoading(false);
        }
      }
    };
    // Se ejecuta el fetch.
    fetchOrders();
  }, [
    // Si "page", la "url" o el "token" cambian, se ejecuta el fetch otra vez.
    [page, apiOrdersUrl, sessionStorageToken];
  ]);
}
```

De esta manera también utilizamos componentes interactivos y reutilizables para renderizar información extraída de una petición al servidor.



Cafetería Ventura Rodríguez

≡

IDENTIFICADOR: 133

12 de junio de 2025, 19:20



cafe
1.80€
4 unidades

DESCRIPCIÓN:

PRECIO TOTAL: 7.20€

PAGADO: no

IDENTIFICADOR: 132

12 de junio de 2025, 19:19



cafe
1.80€
4 unidades

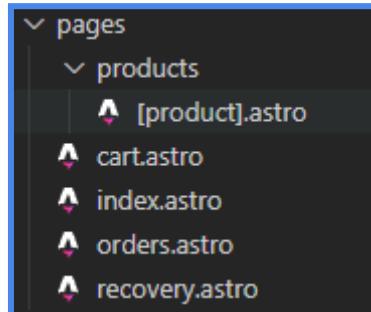
DESCRIPCIÓN:

PRECIO TOTAL: 7.20€

PAGADO: no

3. PrismaORM y generación de páginas dinámicas

Astro te permite generar páginas estáticas de forma dinámica definiendo un archivo como array de nombre dentro de un directorio.



Dentro de este componente debemos definir una función asincrona de astro con el nombre de `getStaticPaths`, esta función definirá cómo se crea cada página, en mi caso utiliza un select de todos los productos de la tabla de productos en la base de datos.

```
---
import TopMenu from "../../components/TopMenu/TopMenu.astro";
import LayoutProduct from "../../layouts/LayoutProduct.astro";
import ProductContainer from "../../components/productsPages/ProductContainer.astro";
import Footer from "../../components/footer/Footer.astro";
import { PrismaClient } from "@prisma/client";

export async function getStaticPaths() {

    const prisma = new PrismaClient();
    const products = await prisma.products.findMany(); //trae todos los productos de la tabla productos

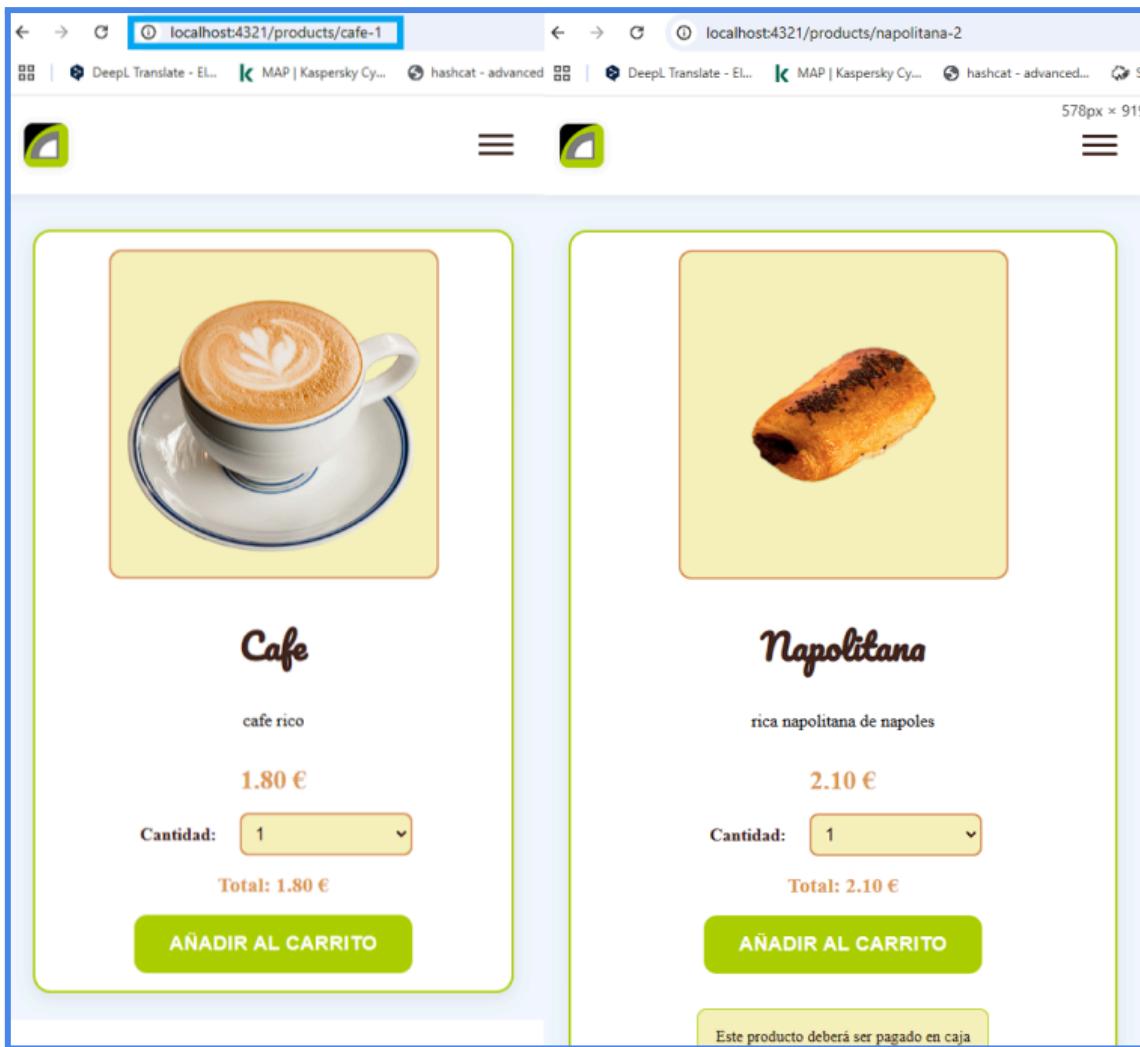
    return products.map( product => ({
        params: { product: product.product_name + "-" + product.id_product },
        props: { product }
    }));
}

const { product } = Astro.props;
const { id_product, is_unlimited, product_name, product_price, url_image, product_description } = product;
---

<LayoutProduct title={product_name}>
    <TopMenu />
    <ProductContainer
        id_product={id_product}
        is_unlimited={is_unlimited}
        product_name={product_name}
        product_price={product_price}
        url_image={url_image}
        product_description={product_description}
    />
    <Footer />
</LayoutProduct>
```

Fuera del frontmatter definiremos qué estructura tendrán todas las páginas.

Si nos fijamos en las Url vemos que estas se generan de forma dinámica como definimos en la propiedad params en getStaticPaths.



Para obtener estos campos de la base de datos he usado prisma ORM que te permite generar las funciones del repositorio escribiendo un archivo .prisma o incluso generando los modelos de la propia base de datos con “npx prisma db pull” y con “npx prisma generate” para generar los repositorios.

```
schema.prisma
```

```
frontend > cafeteria-app > astro-pedidos > prisma > schema.prisma > ...
  Generate
  1 generator client {
  2   provider = "prisma-client-js"
  3 }
  4
  5 datasource db {
  6   provider = "mysql"
  7   url      = env("DATABASE_URL")
  8 }
  9
 10 model products {
 11   is_unlimited    Boolean          @db.Bit(1)
 12   product_price   Float            @db.Float
 13   id_product     BigInt           @id @default(autoincrement())
 14   product_name   String           @db.VarChar(45)
 15   url_image      String?         @db.VarChar(255)
 16   product_description String?     @db.VarChar(255)
 17 }
```

Si además en la página principal mapeamos esos productos y añadimos un href tenemos enlaces dinámicos a esas páginas.

```
<section class="products-container">

{ products.map( product => (
  <article class="product">
    <a href={"/products/" + product.product_name + "-" + product.id_product}>
      <div class="image-container">
        <img
          src={product.url_image ?? "/favicon.png"}
          alt={"imagen " + product.product_name} class="product-image"
          transition:name={product.product_name + "-" + product.id_product}>
        <AddToCart client:only="react"
          productId={product.id_product}
          apiOrdersUrl={apiOrdersUrl} />
      </div>

      <p class="product-name">{product.product_name}</p>
      <p class="product-price">{product.product_price.toFixed(2)} €</p>
    </a>
  </article>
))}

</section>
```



4. View Transitions

Astro nos permite utilizar la nueva API de JS de View Transitions desde un el componente ClientRouter que podemos importar en nuestro layouts.

```
---  
import { ClientRouter } from "astro:transitions";  
---  
<!doctype html>  
<html lang="es">  
  <head>  
    <meta charset="UTF-8" />  
    <meta name="viewport" content="width=device-width" />  
    <link rel="icon" type="image/png" href="/favicon.png" />  
    <link rel="preconnect" href="https://fonts.googleapis.com">  
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>  
    <link href="https://fonts.googleapis.com/css2?family=Open+Sans:ital,wght@0,300..800;1,300..800&family=Pacifico&family=Quicksand">  
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.5.0/css/all.min.css" crossorigin="anonymous">  
    <meta name="generator" content={Astro.generator} />  
    <title>Cafetería Arquitecto Ventura Rodríguez</title>  
    <ClientRouter /> <!-- Para que funcionen las transiciones de navegación entre páginas -->  
  </head>  
  <body>  
    <slot />  
  </body>  
</html>
```

De esta manera ya tenemos transiciones suaves entre las distintas páginas, además con la propiedad transition:name podemos mover elementos con mismo nombre entre varias páginas de forma automática, creando un efecto que capta la atención del usuario.

```
<img  
  src={product.url_image ?? "/favicon.png"}  
  alt={"imagen " + product.product_name} class="product-image"  
  transition:name={product.product_name + "-" + product.id_product}>
```

ANEXO E - PAGINACIÓN

1. Backend

Utilizamos paginación cuando tenemos una consulta muy grande a base de datos y no queremos traer todos los registros de golpe, además usarla de forma correcta tiene algunas ventajas como el acceso a N página de registros de forma permanente desde una URL.

Para utilizar paginación en Spring debemos hacer que nuestro controlador devuelva un objeto Page, este objeto es como una lista pero contiene datos de la paginación como el número de páginas, el tamaño, si estamos en la última página.

```
private static final int NUMBER_OF_PAGES = 10;
no usages  ↳ adminaitor
@GetMapping("/history")
public ResponseEntity<Page<OrderDTO>> getOrderUserHistory(
    @RequestParam(defaultValue = "0") int page,
    @AuthenticationPrincipal UserDetails userDetails) throws CustomException {
    //ordena la paginación por los más recientes
    Pageable pageable = PageRequest.of(page, NUMBER_OF_PAGES, Sort.by( ...properties: "id").descending());
    Page<OrderDTO> orders = orderService.getOrdersByUsername(userDetails.getUsername(), pageable);
    return ResponseEntity.ok(orders);
}
```

Para ello debemos crear un objeto de la interfaz Pageable que contendrá la página que queremos buscar sacada de los RequestParam, el número máximo de objetos por página y cómo debe estar ordenada la búsqueda.

En nuestro repositorio debemos pedir el tipo de devolución como Page genérico de nuestro objeto, además en este caso hago uso de la anotación @EntityGraph que soluciona el problema del n+1 en la base de datos.

```
@Repository
public interface OrderRepository extends JpaRepository<OrderEntity, Long> {

    /**
     * Trae paginadamente toda la consulta
     * orderList {
     *   order {
     *     product {
     *       details
     *     }
     *     ...
     *   }
     *   ...
     * }
     *
     * @param user usuario que tiene el historial
     * @param pageable parámetros de paginación
     * @return historial de pedidos
     */
    2 usages  ↳ adminaitor
    @EntityGraph(attributePaths = {"orderHasProducts", "orderHasProducts.product"})
    Page<OrderEntity> findAllByUser(UserEntity user, Pageable pageable);
}
```

2. Frontend

Al hacer Get obtendremos un objeto content con los detalles de nuestros pedidos.

```
1      "content": [
2          {
3              "id": 188,
4              "description": "",
5              "dateTime": "12 de junio de 2026, 21:01",
6              "products": [
7                  {
8                      "productId": 2,
9                      "name": "napolitana",
10                     "price": 2.1,
11                     "imageUrl": "/napolitana.jpg",
12                     "quantity": 2
13                 }
14             ],
15             "paid": false
16         },
17     ]
```

Junto a otros objetos y campos que nos sirven para saber cómo representar al cliente.

```
168     "pageable": {
169         "pageNumber": 0,
170         "pageSize": 10,
171         "sort": {
172             "sorted": true,
173             "unsorted": false,
174             "empty": false
175         },
176         "offset": 0,
177         "paged": true,
178         "unpaged": false
179     },
180     "totalPages": 1,
181     "totalElements": 10,
182     "last": true,
183     "numberOfElements": 10,
184     "size": 10,
185     "number": 0,
186     "sort": {
187         "sorted": true,
188         "unsorted": false,
189         "empty": false
190     },
191     "first": true,
192     "empty": false
193 }
```

Mientras que el content nos es útil para mostrar el contenido en nuestra web, esta otra parte del JSON nos dice si podemos pedir o no otra página o incluso que flechas mostrar.

```
1 import styles from "./OrderElement.module.css";
2
3 interface PageMenuProps {
4     totalPages: number;
5     first: boolean;
6     last: boolean;
7     currentPage: number;
8 }
9
10 export default function PageMenu({ totalPages, first, last, currentPage }: PageMenuProps) {
11
12     const url = new URL(window.location.href);
13
14     const nextPage = (page: string) => {
15         url.searchParams.set("page", page);
16         window.location.href = url.toString();
17     }
18
19     return (
20         <div className={styles.pageControlContainer}>
21             {!first &&
22                 <button
23                     className={styles.pageControlButton}
24                     onClick={() => nextPage(currentPage - 1 + "")}>{"<"}
25                 </button>}
26             <p>Página {currentPage + 1} de {totalPages}</p>
27             {!last &&
28                 <button
29                     className={styles.pageControlButton}
30                     onClick={() => nextPage(currentPage + 1 + "")}>{">"}
31                 </button>
32             }
33         </div>
34     );
35 }
```

