

Creación de un dashboard para usuarios del ticket digital de Mercadona con visualización gráfica de datos: evolución de precios por producto, gastos por categoría de alimentación y ventanas temporales de gastos.

Santiago Sánchez Sans

Ciclo formativo en desarrollo de aplicaciones web

Memoria del Proyecto de DAW

IES Abastos. Curso 2024/25. Grupo 7S. 25 de Mayo de 2025

Tutor Individual: Carlos Furones

Agradecimientos

Quiero agradecer a mi padre y a mi madre por su apoyo incondicional en la realización de este grado superior a tiempo completo.

Mensaje para el lector

La redacción íntegra de esta memoria se ha hecho a mano. No se ha utilizado inteligencia artificial para la redacción de ninguna frase, ni revisión gramatical ni ortográfica. Se invita al lector valorarla en su idiosincrasia, con sus virtudes y defectos, y considerarla el resultado del esfuerzo y trabajo constante a lo largo de los meses de marzo a mayo de 2025 (en paralelo con la FCT a tiempo completo del grado superior).

Índice general

1. Identificación de objetivos	1
1.1. ¿Qué es el ticket digital de Mercadona?	1
1.2. Identificación de necesidades	1
1.3. Objetivos del proyecto	2
2. Diseño del proyecto	3
2.1. Análisis de la realidad local	3
2.2. Requisitos Funcionales	4
2.2.1. Requisitos de la aplicación	4
2.2.2. Requisitos de los usuarios	5
2.3. Stack tecnológico	5
2.3.1. Front-End: HTML, CSS y Javascript	5
2.3.2. Back-end: Java (Spring Boot) y Python (FastAPI)	6
2.3.3. Cloud: Google API Client	6
2.3.4. BBDD: MySQL y MongoDB	6
2.4. Secuenciación de tareas	7
2.5. Diagrama de sistemas de la aplicación	8
2.5.1. Camino recorrido durante el registro de usuario	9
2.5.2. Camino recorrido durante el inicio de sesión	10
3. Desarrollo del proyecto	11
3.1. GitHub del proyecto	11
3.2. Entornos de desarrollo	11
3.3. Despliegue	12
3.4. Desarrollo back-end (Spring Boot, Java)	13
3.4.1. Contenerización	13
3.4.2. Estructura de la aplicación	13
3.4.2.1. src/main/java: las clases del proyecto	15
3.4.2.2. src/main/resources: archivos de configuración	20
3.4.2.3. pom.xml: dependencias de maven	20
3.4.3. Autenticación y Autorización	21
3.4.3.1. método utilizado: JWT	21
3.4.3.2. ¿Qué compone un JWT?	23
3.4.3.3. Implementación de JWT en java SpringBoot	23
3.4.3.4. Enviar por primera vez el Access Token hacia el front-end (registro)	26
3.4.3.5. Recibir en el back-end el JWT enviado desde el front-end y con él securizar un endpoint de la API mediante interpretación de claims y verificación de firma.	27

3.4.4. Validación de datos (endpoints back-end)	32
3.4.5. Hasheado de contraseñas	35
3.5. Desarrollo back-end (FastAPI, Python)	36
3.5.1. Contenerización	36
3.5.2. estructura de la aplicación	39
3.5.3. Solicitud de subida y parseo de datos	40
3.5.3.1. PARTE 1: FastAPI recibe la POST request en el endpoint /api/subir-tickets-pdf con los tickets adjuntos desde el cliente y con el token de acceso con permisos a 0	40
3.5.3.2. PARTE 2: FastAPI parsea todos los tickets con un algoritmo de extracción	41
3.5.3.3. PARTE 3: FastAPI manda a MongoDB los datos de los tickets parseados	45
3.5.3.4. PARTE 4: FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot y, después de persistir el cambio en los permisos, Spring Boot le manda de vuelta un nuevo token (con permisos a 1). FastAPI transmite al cliente los datos de los tickets	45
3.5.4. Gestión de solicitudes: token de acceso	46
3.5.5. Validación de datos (“/api/subir-tickets-pdf”)	47
3.6. Desarrollo del front-end	48
3.6.1. Contenerización	48
3.6.2. Estructura de la aplicación	49
3.6.3. Enrutamiento de vistas	49
3.6.4. Manejar vistas en función de Autenticación y autorización	53
3.6.4.1. Protegiendo vistas “privadas” ante usuarios no “logueados”: cuando el token ha expirado	54
3.6.4.2. Protegiendo vistas “públicas” para usuarios “logueados”: cuando el token no ha expirado	56
3.6.4.3. Protegiendo vistas “privadas” ante usuarios “logueados”: cuando el token no ha expirado.	57
3.6.4.4. Salir voluntariamente de las páginas privadas: botón “cerrar sesión”	59
3.6.5. Recibir el Access Token desde el back-end	60
3.6.6. Validación de datos (Formularios entrada)	63
3.6.6.1. Validación del correo	63
3.6.6.2. Validación de la contraseña	66
3.6.7. Diseño (UX/UI): páginas publicas	67
3.6.8. Diseño (UX/UI): páginas privadas	69
3.6.8.1. Diseño del dashboard	69
3.6.8.2. Diseño del “pas4_concedirAccesGmail.html”	75

3.6.9.	Arquitectura (JavaScript): páginas privadas	80
3.6.9.1.	Arquitectura del dashboard	80
3.6.9.2.	Arquitectura pas4_concedirAccesGmail.html	81
3.6.10.	Diseño de iconos	85
3.6.10.1.	Áreas de producto	85
3.6.10.2.	Icono mercApp pequeño	86
3.7.	Desarrollo Cloud: configuración google API client	87
3.8.	Bases de datos	94
3.8.1.	Base de datos mySQL	94
3.8.2.	Base de datos mongoDB	95
4.	Evaluación y Conclusiones Finales	96
7.	Bibliografía	97
5.	ANEXO	100
5.1.	Flujo de trabajo habitual en git	100
5.2.	Diferencias de seguridad: JWT vs SESSID en cookies seguras	101
5.3.	Clases para crear y verificar JWTs	102
5.3.1.	Clase JWT	102
5.3.2.	Clase Refresh Token	102
5.3.3.	Clase Access Token	102
5.4.	Clases de seguridad	103
5.4.1.	Clase ConfiguracioSeguretat.java	103
5.4.2.	Controlador con restricciones aplicadas	104
5.5.	Diagrama réplica netflix	105
5.6.	Diagrama enrutamiento mercApp	106
5.7.	Aspectos ventajosos de separar front-end y back-end (SoC)	106
5.8.	Captura proyecto desarrollo interfaces	107
5.9.	Creación del icono mercapp pequeño: IA con Gimp	108
5.10.	Google Cloud: descargar tickets digitales mediante cloud	109
5.11.	Google cloud: cálculo de unidades de cuota	110
5.12.	Correspondencia entre páginas del sistema de registro de NetFlix y el sistema de registro de mercApp	112

Identificación de objetivos

1.1. ¿Qué es el ticket digital de Mercadona?

Mercadona implementa un sistema de tickets digitales que vinculan la tarjeta de débito a un correo electrónico. Cualquier usuario del supermercado que quiera utilizar el ticket digital solamente deberá facilitar estos dos datos y el supermercado le enviará por correo electrónico los tickets de las posteriores compras hechas en cualquier establecimiento de Mercadona.

Las ventajas para el usuario y para el supermercado de tener un ticket digital son evidentes: el cliente no perderá los tickets de cara a devoluciones, no deberá esperar a su impresión después del pago y no se verá expuesto a la tinta del texto del ticket físico: que al menos por allá en 2019 la comisión europea ya alertaba de su peligrosidad [1] a partir de un estudio de la Universidad de Granada que hallaba alto contenido de Bisfenol-A¹ en los tickets de compra de distintos países [2][3].

Las ventajas de la adopción masiva del ticket digital para el supermercado y el trabajador también son claras: se evita el derroche de papel, se impide que los cajeros estén expuestos a químicos que constituyan un riesgo laboral y, finalmente, se consigue acortar los tiempos de cola mejorando la experiencia de los clientes y asegurando su regreso futuro.

1.2. Identificación de necesidades

Los tickets de cada usuario del ticket digital de Mercadona se acumulan de forma recurrente en su correo electrónico. A pesar de contener información valiosa de cara a la planificación de gastos, este formato digital solamente responde a ventajas operativas para el supermercado y cliente del mismo; pero no ha mejorado todavía la asimilación ni la interpretación de los datos por parte del cliente: este no puede visualizar lo que ha gastado a lo largo de un período temporal, ni la evolución de los precios de los productos que adquiere, ni los supermercados en los que ha comprado, ni las veces que lo ha hecho, etc.

Con un formato estructurado como el que ya tienen a día de hoy los tickets

¹Es un químico que es un disruptor del sistema endocrino.

digitales podemos sacar muchísima información y presentársela al cliente de forma clara y visual: los asuntos de los correos que contienen los tickets tienen un formato estándar y predecible, y dentro de cada correo electrónico se encuentra un solo PDF con el desglose de la compra (producto, unidades vendidas, establecimiento, etc.) que espera ser minado y analizado.

1.3. Objetivos del proyecto

Este proyecto quiere responder a estas necesidades. Para ello se plantea la Creación de un *dashboard* o “cuadro de mando” en forma de aplicación web para que un usuario del ticket digital de Mercadona pueda visualizar la evolución de precios de los productos adquiridos, el coste promedio de sus compras por períodos temporales y sus distribuciones de gastos a partir de los tickets digitales guardados en una base de datos.

A grandes rasgos, los **Objetivos principales** del proyecto son proporcionar al usuario del ticket digital una herramienta que muestre en gráficos visuales:

- **La evolución de precios** (inflación) a lo largo del tiempo en los productos habitualmente obtenidos en el mismo establecimiento².
- **Evolución del gasto** total del usuario a lo largo del tiempo por períodos temporales.

Más concretamente los subobjetivos los mostramos en forma de requisitos en el apartado [2.2.1](#)

²La evolución de precios se mostrará solamente para un mismo centro de Mercadona, dado que distintos centros pueden cambiar los nombres de los productos (por ejemplo, en Cataluña...).

Diseño del proyecto

Como veremos en el apartado [2.1](#), el motivo por el que se ha usado principalmente Java como lenguaje de back-end y Spring Boot como framework está vinculado con el análisis de la realidad local: tanto en la probabilidad de inserción laboral futura como en economía de tiempo durante la realización de las prácticas.

Asimismo, de los objetivos principales de los que hemos hablado en la sección [1.3](#) hemos derivado una serie de requisitos funcionales de la aplicación, que se verán en el apartado [2.2.1](#)

2.1. Análisis de la realidad local

Esta aplicación tiene muchísimo contenido de back-end. Por ello, la elección del lenguaje de programación para este era importante y obedece a criterios puramente estratégicos.

En primer lugar, se ha utilizado el lenguaje de programación Java y el framework Spring Boot porque en el equipo de desarrollo de software en el que me he integrado para las prácticas en Lâberit me estoy formando justamente en este lenguaje y framework.

En segundo lugar, el tiempo de formación inicial en estas prácticas ha entrañado un curso de Spring Boot excesivamente introductorio, y se estimó que la única forma de ganar conocimientos reales era desarrollar una aplicación de back-end completa y segura con el framework, desde cero. De este modo, el salto futuro a una posición más integrada en el equipo de prácticas debería ser más probable; y aprovechando el solapamiento existente en el desarrollo del sistema de gestión de usuarios de mercApp y los contenidos del curso de Spring Boot mandado desde Lâberit han permitido realizar parte de este back-end durante el primer mes de prácticas en la empresa.

En tercer lugar, la elección de Java como lenguaje de back-end en lugar de PHP viene motivada porque otras empresas de la zona utilizan el framework Spring Boot (Mercadona tech, una de ellas). No es de extrañar que esto pase, dado que es el lenguaje que Imma Cabanes nos enseñó en primer curso en Abastos y también el lenguaje que se enseña en primer curso a los estudiantes del grado de ingeniería informática de la Universitat Politècnica de Valencia. Ello implica que el ecosistema

tecnológico valenciano se nutre de forma orgánica de desarrolladores Java salidos de la academia.

En cuarto lugar, apostar por Java como lenguaje de back-end es un caballo ganador en forma de crecimiento profesional al ser un lenguaje sólido y de largo recorrido no solo en Valencia sino en muchos países de Europa¹. No es un lenguaje que vaya por modas y ejercitarlo puede permitir iniciar una carrera enfocada en una tecnología que no se prevé que desfallezca en un futuro cercano.

En quinto y último lugar, se escogió Python para la realización de la parte de extracción de tickets porque es el lenguaje de programación con el que aprendí algorítmia por primera vez, hará justo ahora 10 años, y con el que en 2017 hice un trabajo final de máster con modelos predictivos usando la librería Sklearn². Por todo ello, y por su simplicidad, es mi lenguaje de scripting por elección; por no mencionar que con este lenguaje ya he parseado PDFs en el pasado. Además, hay una probable oportunidad laboral que requiere conocer este framework; con lo cual hay otro motivo más, si cabe, para aprenderlo.

2.2. Requisitos Funcionales

NOTA: Los requisitos presentes en el siguiente subapartado se suman a los requisitos que de forma tácita se sobreentiende que debe tener una aplicación de un proyecto final de grado superior; es decir: tener un front-end, un back-end con sistema de registro de usuarios, un login con buenas prácticas en materia de seguridad y una base de datos.

2.2.1. Requisitos de la aplicación

REQUISITO A: Mostrar *evolución de los precios* de los productos unitarios adquiridos con más frecuencia (visualizable en un gráfico donde en X tendremos el tiempo y en Y el precio en euros). Para los productos de precios muy variables (productos a granel, como frutas, etc.), se mostrará la evolución del precio por kg a lo largo del tiempo.

REQUISITO B: Mostrar *gasto total en distintas ventanas temporales* del usuario: períodos de 1, 3, 6 meses y un año; independientemente del centro de Mercadona en el que se compre (todos juntos).

REQUISITO C: Al lado de este mismo coste total mostrado en REQUISITO

¹Este fue uno de los motivos que me hizo contactar con Lâberit para hacer las prácticas: poder tocar back-end con Java y Spring Boot.

²El lector puede ver el repo de GitHub con el código y el trabajo final de máster [aquí](#).

B, se incluirá un ***diagrama de sectores*** desglosando porcentaje de dinero gastado en 13 categorías: verdura y hortalizas, frutas, huevos y lácteos, agua y bebidas, aceite y especias, carne, pescado, hogar e higiene personal, Pan y pastelería, pasta, arroz y legumbres, Snacks y dulces, Mascotas, sin clasificar³.

REQUISITO D⁴: Podremos permitir que los PDFs descargados del correo del usuario se almacenen en una carpeta local del mismo para que pueda verificar la extracción de los datos.

REQUISITO E⁵: El sistema front-end y back-end de registro permitirá redirigir a los usuarios rápidamente a un registro de forma inteligente. Nos inspiraremos en el sistema de registro e iniciar sesión de NetFlix.

2.2.2. Requisitos de los usuarios

El correo electrónico y la contraseña de la cuenta de Gmail de alguien que sea usuario del ticket digital de Mercadona y tenga decenas de tickets digitales por analizar, con compras estables y productos recurrentes.

NOTA: En la demo se proporcionarán ya muchos tickets digitales (tickets míos, que cederé para mostrar la utilidad de la aplicación). No será necesario recurrir a la extracción de datos de otro usuario de ticket digital. Se mostrarán un mínimo de tickets digitales en un mismo centro de Mercadona para poder evidenciar la evolución de precios y gastos.

2.3. Stack tecnológico

2.3.1. Front-End: HTML, CSS y Javascript

Se han usado HTML, *vanilla* CSS y *vanilla* JavaScript. Excepciones al uso de los lenguajes puros en el front-end son una librería javascript para la visualización de gráficos, [chart.js⁴](#); y una librería de css que permite aplicar transiciones, [animate.css⁵](#) conjuntamente con [wowjs⁶](#) que expande las capacidades de animate.css. Las tres fueron vistas en la asignatura de Desarrollo de Interfaces Web.

Como hemos visto en los requisitos de la aplicación, en el sistema de registro e iniciar sesión se ha hecho una réplica mediante desarrollo inverso de los procesos que

³Para ello, dado que no tenemos categorizados todos los productos de Mercadona ni podríamos hacerlo por falta de una lista exhaustiva, si nos da tiempo, se usará un modelo predictivo con word embeddings (módulo Spacy) y cosine similarity (sklearn) para encontrar distancias pequeñas entre las descripciones de los tickets y las categorías, facilitando así la clasificación.

⁴Requisito añadido después de la presentación del proyecto.

⁵Requisito añadido después de la presentación del proyecto.

Netflix utiliza a tal efecto, adaptándola a nuestro caso particular (puede verse como se ha aprovechado ello en la sección [3.6.3](#)). En este paso se ha dedicado muchísimo tiempo y es quizás la parte más importante de este proyecto en cuanto a vínculos directos con los objetivos del CFGS de DAW.

El diseño de las páginas será responsive, hecho con *media queries* de CSS, a excepción de los casos en que las librerías de gráficos utilizadas no lo permitan. Se insta al lector a modificar el tamaño de la ventana en todas las vistas del proyecto para valorar las soluciones empleadas.

2.3.2. Back-end: Java (Spring Boot) y Python (FastAPI)

- Back-end con Java (Spring Boot) para gestionar los usuarios: guardarlos en bbdd y hacer autenticación y autorización de usuarios mediante token JWT: con este framework persistiremos datos en la BBDD mySQL.

- El Back-end con Python (FastAPI) se usará para gestionar los tickets y sus datos: *en primer lugar*, parseará el contenido de los tickets en PDF mediante pyPDF; *en segundo lugar*, guardará (y luego leerá) de MongoDB los datos extraídos por paraseo de los tickets digitales mediante la librería *pymongo*, asegurando así búsquedas eficientes; *en tercer lugar*, si queda tiempo antes de la presentación final, se usarán módulos de Python como sklearn, spacy y Numpy para procesar con lenguaje natural (NLP) cada producto y categorizarlo automáticamente en una de estas [12 categorías](#) de alimentación distintas (en lugar de hacer la clasificación manual); *en cuarto lugar*, incluirá la capacidad de autenticar y autorizar conexiones entrantes pero no de generar nuevos tokens de acceso.

2.3.3. Cloud: Google API Client

Google API Client, y más específicamente la **Gmail API**, permitirá a cada usuario descargarse en cuestión de uno o dos minutos hasta 500 tickets digitales de su gmail directamente a su ordenador mediante Oauth2. Esto lo haremos con un script en JavaScript.

2.3.4. BBDD: MySQL y MongoDB

Para guardar los datos de los usuarios se debe usar un sistema de gestión de base de datos relacional. Hemos escogido MySQL dado que es el que hemos visto en el grado superior y ya lo conocemos.

Sin embargo, los productos de Mercadona no los conocemos de antemano ni tenemos una lista exhaustiva de los mismos. Además, el número de productos que

se pueden encontrar en un ticket varía en cada compra, por lo que no podemos usar una base de datos relacional tradicional como MySQL o PostgreSQL porque se trata de información no estructurada. En su lugar, usaremos MongoDB, una BBDD NoSQL que almacena datos en formato JSON y permite, además, búsquedas eficientes.

Para optimizar el backend, intentaremos que un usuario pueda consultar repetidamente sus compras sin sobrecargar el servidor. Cuando inicie sesión y consulte sus datos de tickets, estos se descargará de mongoDB y se almacenarán en el localStorage del cliente (navegador). En consultas posteriores, los datos se obtendrán directamente de localStorage sin necesidad de hacer peticiones al servidor, hasta que expire el token de acceso del usuario: caso en el que, ahora sí, se borrarán los datos del localStorage. Esta es la implementación ideal; pero si hay falta de tiempo haremos extracción directa de la base de datos cada vez que el usuario actualice la página del dashboard (que, en mi opinión, no es lo ideal).

2.4. Secuenciación de tareas

El desarrollo del proyecto ha empezado entorno al **7 de marzo de 2025** (el día después de la sesión informativa sobre proyectos). Desde ese día hasta el día **21 abril** se ha invertido tiempo en la parte del front-end para gestionar el registro e inicio de sesión (páginas públicas) del proyecto, en comisión y en paralelo con el sistema de autenticación, autorización y securización de las APIs vinculadas a la tabla Usuari en el back-end (el entramado más o menos complejo que mostramos en la figura 3.23 principalmente) y gran parte del redactado de la parte de Spring Boot del proyecto (ver apartado 3.4).

Durante el horario de las prácticas de Lāberit (iniciadas el 10 de marzo) hasta el viernes 11 de abril se ha diseñado el back-end en Java Spring Boot en paralelo a la realización del curso introductorio de Spring Boot mandado por la empresa (ver imagen 3.2, recuadro en rojo, verde y azul).

Fuera del horario de prácticas desde el **21 de abril** hasta el **24 de mayo** (día previo a la entrega de la memoria) se intentará terminar las dos páginas privadas:

- `pas4_ConcedirAccesGmail.html`: que da acceso a tickets digitales (donde se hará la integración con API de Google).
- `dashboard.html`: que obtiene datos de la API de FastAPI que a su vez extraerá los datos de MongoDB.

2.5. Diagrama de sistemas de la aplicación

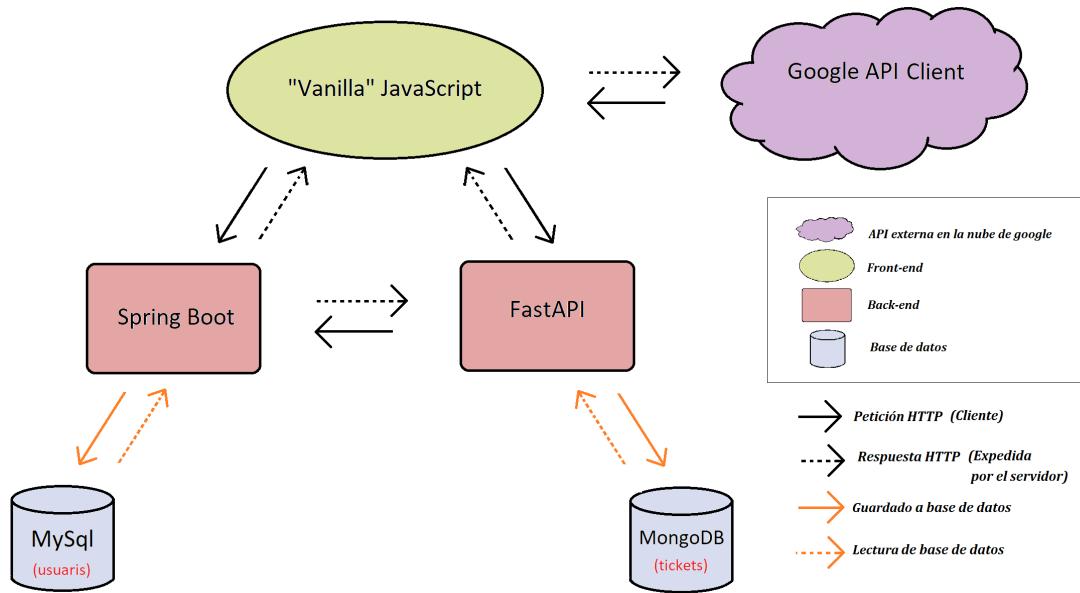


Figura 2.1: Diagrama de sistemas simplificado de la aplicación mercApp. Tenemos el back-end de Spring Boot, donde se mandan a guardar y leer de mySQL los datos de los usuarios y gracias al cual se gestiona el enrutamiento de vistas públicas. Tenemos también el back-end de fastAPI, que parsea los tickets digitales y los pasa a formato estructurado JSON para persistirlos en MongoDB. También hacemos uso de Google API Client, que pertenece a *Google Cloud*, y que permite la extracción de tickets del Gmail del usuario hacia su ordenador, a petición suya a través del navegador. El front-end con “Vanilla” JavaScript muestra las vistas al usuario en función del token de acceso que expide Spring Boot.

2.5.1. Camino recorrido durante el registro de usuario

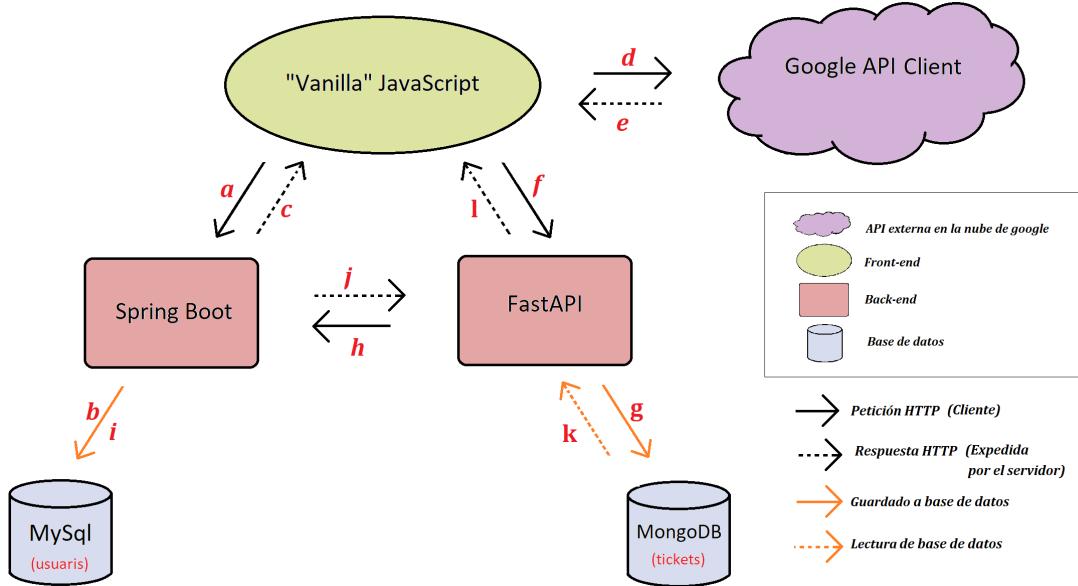


Figura 2.2: Diagrama de sistemas simplificado con los caminos activados durante el registro de un usuario (no superusuario). Mostramos acontecimientos en orden desde que este introduce su correo electrónico y su contraseña por primera vez hasta que este tiene acceso al dashboard de visualización, siempre que haya éxito en todas las partes del proceso: a) Usuario pone correo y contraseña. b) guardamos correo y hash de la contraseña. c) Mandamos al front-end un token de acceso con permisos = 0 y le pedimos al usuario que nos conceda acceso a sus tickets digitales. d) usuario se autentica con la Gmail API de google API Client, con OAuth2, y automáticamente se pide la descarga de sus tickets digitales e) tickets digitales en PDF regresan al navegador y se descargan al ordenador del usuario. f) cuando el usuario lo quiera sube a fastAPI los tickets de su ordenador para que el servidor los parsee a formato estructurado g) Los tickets con formato estructurado se guardan en MongoDB h) FastAPI pasa el token de acceso con permisos = 0 hacia Spring Boot y le pide uno con permisos = 1. i) Spring Boot guarda permisos a 1 a la BBDD mySQL j) Spring Boot regresa a fastAPI el token de acceso recién expedido con permisos = 1. k) FastAPI obtiene los datos de los tickets de usuario de MongoDB l) FastAPI manda el token de acceso con permisos = 1 y los datos extraídos de MongoDB (información de los tickets) de vuelta al front-end para que JavaScript los muestre automáticamente en el dashboard de visualización.

2.5.2. Camino recorrido durante el inicio de sesión

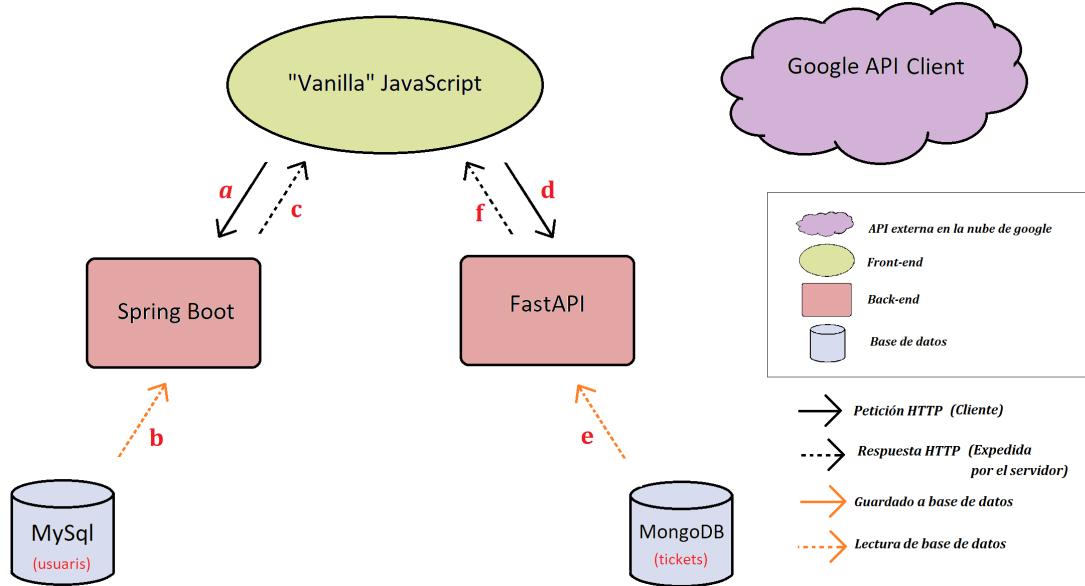


Figura 2.3: Diagrama de sistemas simplificado con los caminos activados durante el *inicio de sesión* de un usuario (no superusuario) que ya fue registrado, ya facilitó sus tickets digitales para su extracción o minado -parseo- y estos ya fueron extraídos y persistidos en MongoDB. Mostramos acontecimientos en orden desde que este introduce su correo electrónico y su contraseña hasta que visualiza el dashboard, siempre que haya éxito en todas las partes del proceso: a) Usuario escribe y manda correo y contraseña. b) Obtenemos usuario y hash de la BBDD y en Spring Boot vemos que son coincidentes con los que escribió el usuario por el front-end. c) Mandamos al front-end el token de acceso con permisos a 1 y automáticamente el front redirigirá a *dashboard*. d) Se pide a FastAPI la información de los tickets. e) Se toman los tickets de la bbdd mongoDB. f) Se mandan de fastAPI al front-end (cliente, navegador) los datos de los tickets para ser visualizados en el dashboard.

Desarrollo del proyecto

3.1. GitHub del proyecto

Para desarrollar este proyecto se ha trabajado con GitHub y git. Para su desarrollo se ha seguido la estrategia de crear ramas de característica (puede verse anexo 5.1 para ver el flujo de trabajo habitual) y, una vez satisfactoria, hacer un merge en la rama main en local. Los cambios de la rama main local se han ido subiendo al repo remoto en la rama main.

Las instrucciones para correr los componentes del proyecto en sus respectivos entornos de desarrollo están en el apartado 3.2. Las instrucciones para hacer un despliegue de la aplicación mediante contenedores Docker se encuentran en el apartado 3.3.

Link al repositorio → <https://github.com/blackcub3s/mercApp>

3.2. Entornos de desarrollo

Para el back-end de Java con SpringBoot se ha utilizado el editor Java **IntelliJ Idea community edition** que expone el backend en el puerto **8080**¹: se han utilizado algunas extensiones del editor para correr el proyecto. Por ejemplo, es necesario la extensión para Lombok: sin ella IntelliJ no encontrará getters o setters y fallará.

Para el front-end (archivos estáticos: HTML, CSS y JavaScript) se ha utilizado **VScode**, con la extensión live server para poder hacer llamadas al back-end directamente desde el puerto **5500**².

Para el back-end de Python con FastAPI se ha utilizado el servidor embedido en el propio framework, expuesto en el puerto **8000**³.

Para la base de datos mySQL se usó workbench y se expuso el puerto **3306**⁴. Para MongoDB se usó MongoDB compass exponiendo en el **27017**⁵.

¹NOTA: Para correr el proyecto back-end con *Spring Boot* abrir con intelliJ la carpeta app!

²El proyecto *front end* con *vanilla javascript* debe abrirse con vscode en la carpeta **app** y ejecutarlo en live server, si no **no** funcionará correctamente!

³Se recomienda ejecutarlo desde la carpeta **/app** mediante *fastApi dev controlador.py*

⁴Es la versión MySQL 8.4.4 la que expone el puerto

⁵Es MongoDB Community Server el que expone el puerto.

3.3. Despliegue

Los componentes del apartado 3.2 de entornos desarrollo se han *dockerizado* de modo que cada uno de ellos sea un microservicio, a excepción (por ahora), de las bases de datos. Hemos sido cuidadosos de que los microservicios en Docker corran en los mismos puertos que los servidores embedidos con los que hemos hecho el desarrollo de la aplicación mercApp: por claridad y porque múltiples archivos dependen ya de ellos.

Para crear la imagen de cada microservicio hemos creado un Dockerfile. Luego, para crear los contenedores y crear de nuevo las imágenes en caso que haya cambios en los archivos, para cada uno de estos microservicios, hay un script denominado *creaImatge_i_arrancaContenedor.sh*⁶ que reunirá todos los subcomandos de docker necesarios para el ciclo de vida de la creación de imagen (build) y subsecuente instanciado de contenedor (create, start, stop, rm). Todo ello haciendo solamente desde la terminal de Linux o desde la terminal de bash en Windows:

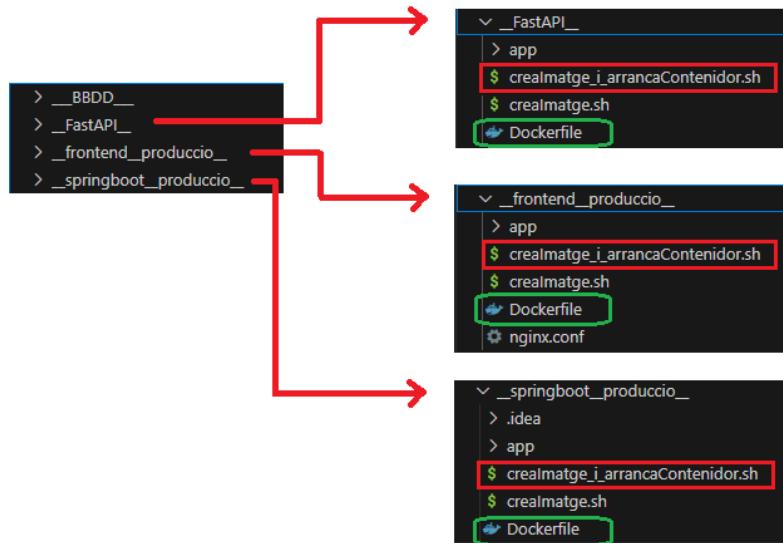
```
bash creaImatge_i_arrancaContenedor.sh
```

Así, no tenemos que preocuparnos de parar un contenedor activo, borrarlo y luego crear de nuevo la imagen del que deriva (ver figura 3.1)⁷.

⁶La explicación pormenorizada de sus comandos, para el caso de FastAPI por ejemplo, puede verse en el apartado 3.5.1.

⁷A futuro planteamos hacer el despliegue con un sistema de orquestación de contenedores como es Kubernetes, subiendo las imágenes a un registry; por ahora no lo haremos y quien quiera desplegarlo en su ordenador para probarlo puede hacerlo fácilmente creando las imágenes e instanciando los contenedores con los scripts en Bash en rojo. No se hará porque las barras bajas de los nombres de las carpetas probablemente darán problemas: la memoria tiene ya muchos links al GitHub que dependen que estas rutas estén bien y por ahora no se tocará.

Figura 3.1: En rojo los archivos que crean una imagen y arrancan su contenedor, para cada microservicio. NOTA: Importante correr el script en bash con la terminal en el *mismo nivel* del árbol de directorios al que pertenece el Dockerfile.



3.4. Desarrollo back-end (Spring Boot, Java)

3.4.1. Contenerización

El archivo Dockerfile y los scripts en Bash para la creación de imágenes y contenedores se encontrarán ⁸ [aquí](#). Estos archivos se ubican, igual que lo que pasa con los demás microservicios del proyecto mostrados en la figura 3.1, un nivel por encima de la carpeta */app*. Si el lector quiere profundizar en el tema de generar imágenes y contenedores con bash puede ir directamente al apartado 3.5.1 donde se explica la contenerización mediante el caso particular del microservicio de FastAPI.

3.4.2. Estructura de la aplicación

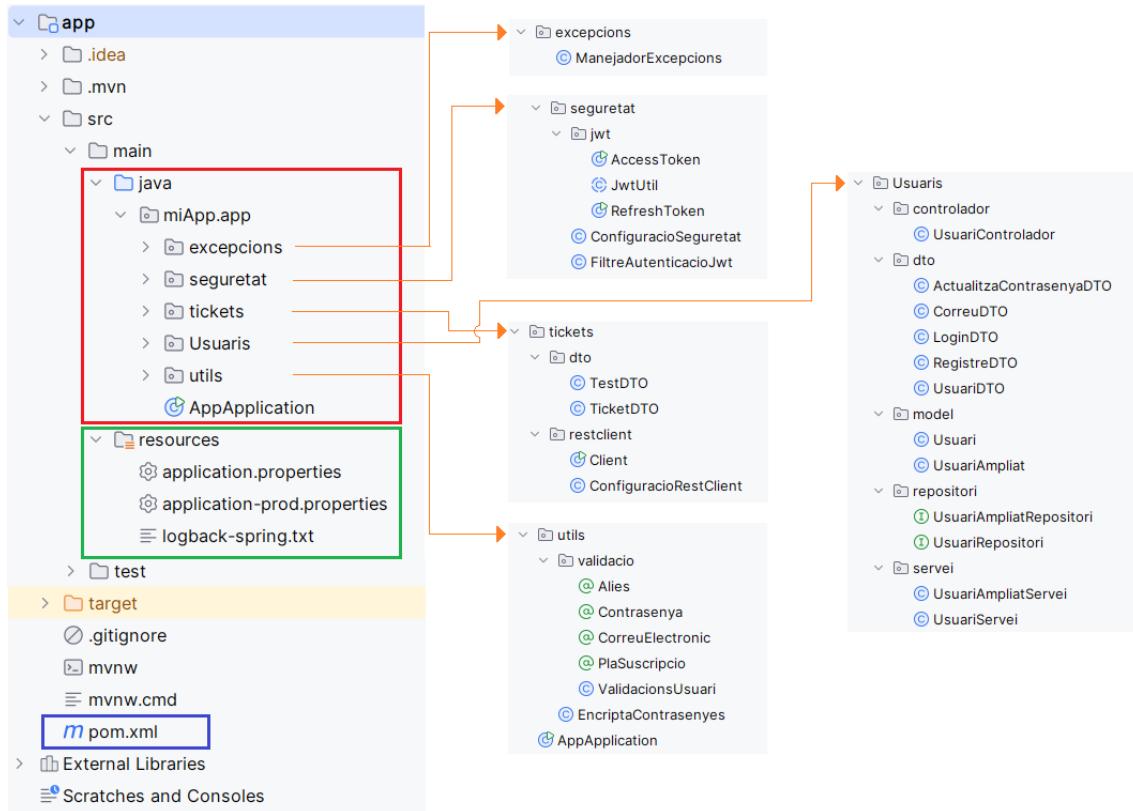
La parte de mercApp programada con Java (Spring Boot) se puede ver en el repositorio de GitHub del proyecto abriendo esta carpeta: [link](#). Recomendamos al lector que abra y corra el proyecto con IntelliJ abriendo esta misma carpeta enlazada en GitHub, pero en local. Al hacerlo, el lector podrá ver entonces en el explorador de directorios de IntelliJ *tres* rutas importantes, que son las que se encuentran por defecto en cualquier proyecto Spring Boot:

⁸En el momento de escribir esta memoria el Dockerfile para el microservicio de Java había que programarlo. Si todavía no está disponible en el link proporcionado el lector puede dirigirse a los apartados Contenerización de los demás microservicios: FastAPI y Nginx (front-end).

- **src/main/java** → En esta ruta encontramos los *packages* y las clases del proyecto (ver figura 3.2 recuadro en rojo).
- **src/main/resources** → Dentro de esta carpeta nos encontramos con el archivo **application.properties**. Se trata de un archivo de configuración donde se define, por ejemplo, el conexiónado con la base de datos (ver figura 3.2 recuadro en verde) o el archivo **logback-spring.xml** que, aunque no está definido por defecto en una aplicación Spring Boot, si lo añadimos nos guarda los logs del proyecto en **logs/spring.log** en lugar de salir impresos por la terminal⁹. Aquí aparece como un **.txt** en lugar de un **.xml** porque por ahora no queremos que se guarden los errores.
- **pom.xml** → Es un archivo donde encontramos el árbol de dependencias de Maven¹⁰ (ver figura 3.2, recuadro en azul): cada vez que añadimos una dependencia estamos añadiendo nuevas funcionalidades a nuestro proyecto SpringBoot a través de una descarga automatizada del repositorio central de Maven.

Figura 3.2: Estructura del back-end de Spring Boot en la aplicación mercApp.

(Cajas de color → {clases del proyecto, archivos de configuración, dependencias de Maven})



⁹La ruta se crea automáticamente

¹⁰Maven es una herramienta de automatización para la construcción de proyectos Java. Gestiona todas las dependencias, que se descargan desde un repositorio central muy vivo donde hay millones de nuevos paquetes publicados anualmente ([ver link](#)). Maven permite empaquetar el proyecto en un **.jar** o **.war** fácilmente, entre otras cosas.

3.4.2.1. `src/main/java`: las clases del proyecto

Dentro de `src/main/java` tenemos la ruta `miApp.app/utils/validacio` donde residen las clases de anotación utilizadas para permitir que los campos de formulario de entrada se validen en el back-end (nótese que, con esto, hemos establecido redundancia de validaciones por si se diese el caso que un usuario maliciosamente intentase hacer llamadas directas a la API, esquivando así las validaciones ya definidas en el front-end): los archivos y su explicación quedan referenciados en detalle en el apartado [3.4.4](#).

Dentro de `src/main/java` tenemos también la ruta `miApp.app/seguretat/jwt` donde tenemos clases que generan tokens de acceso y de refresco a partir de una clave secreta que solo está en el servidor, que referenciamos en detalle dentro de [3.4.3.3](#); Análogamente, las clases que permiten implementar la autenticación y la autorización a partir de los tokens generados por las clases anteriores, las referenciamos también en detalle dentro de [3.4.3.5](#).

Dentro de `src/main/java` tenemos también la ruta `miApp.app/seguretat` encontramos la clase `EncriptaContrasenyes.java`, que utilizamos en el servicio de usuaris para hashear las contraseñas en la base de datos. Esto lo explicamos en detalle dentro de [3.4.5](#).

Dentro de `src/main/java` tenemos también la ruta `miApp.app/tickets/`. Ahí dentro tenemos unas clases diseñadas para expansiones futuras que no han sido utilizadas por ahora: por ejemplo, dentro del package `restclient` tenemos una clase habilitada que permite hacer llamadas HTTP como cliente hacia el back-end de FastAPI¹¹ (sí, el back-end de Java Spring Boot puede actuar como cliente hacia otro back-end mediante la librería RestClient) .

Dentro de `src/main/java` tenemos la ruta `miApp.app/Usuaris` con varios subpackages. Ahí dentro merece la pena mencionar que tenemos todas las clases con las que se recibe y procesa datos del cliente para hacer la gestión de usuarios. Las que nos interesan son las siguientes: `UsuariControlador.java`, `UsuariServei.java`, `UsuariRepositori.java` y `Usuari.java`¹². A continuación vamos a mostrar, a modo de ejemplo, cuál es la utilidad de todas esas clases cuando hacemos una llamada a un endpoint del `UsuariControlador.java` (concretamente el endpoint `/api/login`, que veremos que es el endpoint que interviene en el inicio de sesión): más adelante veremos que este endpoint cobra un papel importante en el enrutamiento front-end de la aplicación (que mostraremos exhaustivamente en [3.23](#)) y que su utilidad es

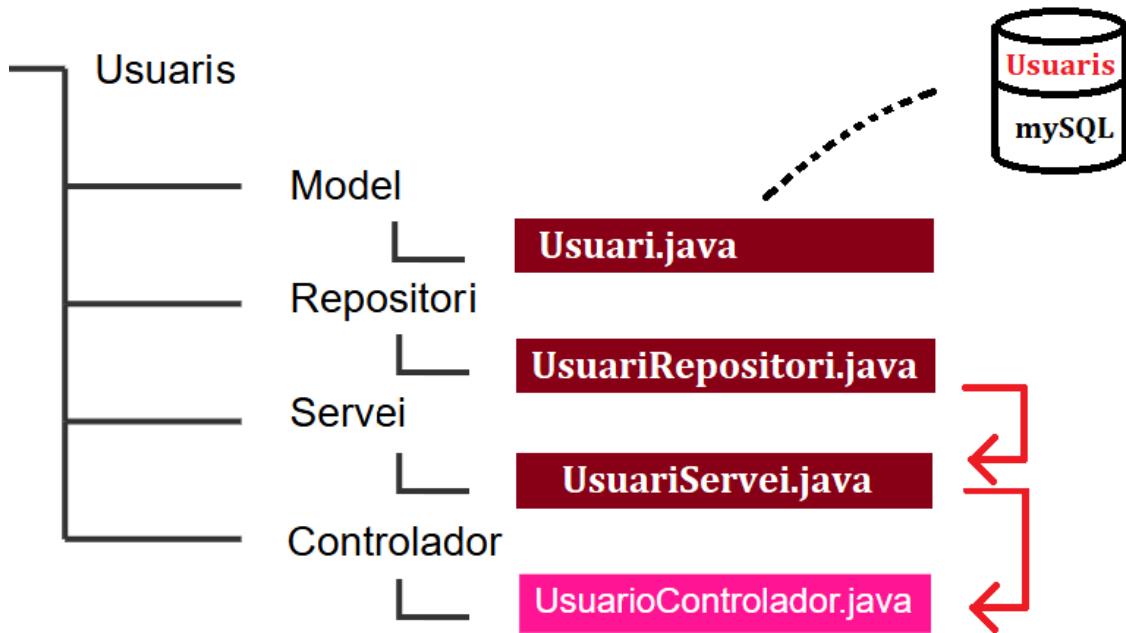
¹¹Ambos back-ends sí se comunicarán, pero la comunicación se establecerá finalmente en sentido opuesto: desde el back-end de Python FastAPI **hacia** el back-end de Spring Boot para que este último emita tokens, como ya explicaremos en la *parte 4* del apartado [3.5.3](#) dedicado a FastAPI.

¹²No nos interesa por ahora hablar mucho de `UsuariAmpliat.java` y de `UsuariAmpliatRepositori.java` porque se han creado como plantilla para la expansión de datos de usuarios si a futuro seguimos desarrollando esta aplicación.

crucial para la redirección a una de las dos páginas privadas.

Antes que nada, empero, mostramos una figura explicativa a nivel gráfico de lo que pasa en este proyecto Spring Boot y, en teoría, en todos: luego iremos desgranando cada componente (ver figura 3.3). De esta figura empezamos destacando la labor del Controlador: en Spring Boot (y probablemente en otros frameworks de back-end), los *endpoints* o puntos de entrada de solicitudes HTTP, van a ser definidos dentro de lo que llaman una clase “controller” o controlador. En este caso, la clase controlador que nos ocupa está en el archivo `UsuariControlador.java`.

Figura 3.3: Inyección de dependencias en nuestro proyecto Spring Boot bajo el modelo-vista-controlador (MVC) dentro del package `Usuaris`. Spring Boot añade otra capa más al modelo que ya conocíamos de desarrollo web entorno servidor: el repository (repositorio). En el `UsuariControlador.java` están los endpoints que tienen que ver con los usuarios. Estos endpoints dependen de la capa de servicio (`UsuariServei.java`), que es la que implementa la lógica de negocio. La lógica de negocio depende, a su vez, de las operaciones en base de datos, que se implementan en el repositorio (`UsuariRepositori.java`). El “model” es otra capa opcional que se implementa en Java y cuya función es mapear cada atributo de una clase Java a una columna de una tabla de la base de datos



Habiendo introducido a las clases más importantes del proyecto en la figura 3.3, ahora, con un ejemplo, vamos a ver como estas clases interaccionan entre sí cuando un usuario llama al endpoint de inicio de sesión `/api/login` con un cliente (navegador, Postman) y obtiene, como response, un JSON como el siguiente o como el que se muestra en el recuadro verde de la figura 3.4:

```
{
    "correoElectronico" : "acces@gmail.com",
    "contrasenya" : "12345678Mm\_"
}
```

Figura 3.4: Captura de Postman: en verde la petición POST al controlador `/api/login` | En rojo, la respuesta de la llamada exitosa a ese controlador, resultante en la obtención de un token de acceso y otros datos.

POST <http://localhost:8080/api/login>

Params Authorization Headers (8) **Body** Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL **JSON**

```
1  {"correoElectronico" : "acces@gmail.com", "contrasenya" : "12345678Mm\_"}
```

Body Cookies Headers (14) Test Results

{ } JSON ▾ Preview ⚡ Visualize

```
1  {
2      "usuari": {
3          "alias": "blackcub3s",
4          "permisos": 1,
5          "idUsuari": 2
6      },
7      "existeixUsuari": true,
8      "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MSwiZWVc3VhcmkiO:
9      "teAccesArecursos": true,
10     "contrasenyaCorrecta": true
11 }
```

La petición que hace el front-end, pues, lleva el correo y la contraseña; y la respuesta que recibe este front-end o, en este caso Postman, está marcada en la figura anterior con el recuadro en rojo. Como veremos en la figura 3.5, el controlador `/api/login` está activado por una función denominada `login` que pide obtener el body de una solicitud POST: la indicación de esperar el cuerpo o body de la solicitud se hace mediante la anotación `@RequestBody` (que va a guardar el cuerpo dentro de la variable pasada por parámetro, en este caso `dto`¹³), mientras que `@PostMapping` nos indica que el controlador solamente procesa solicitudes HTTP de tipo POST.

Asimismo, el back-end de Spring Boot no va a permitir conexiones entrantes de

¹³En el apartado de validaciones hablaremos sobre los DTOs o Data Transfer Objects, y para qué sirven.

un origen distinto al que está corriendo; con lo cual si queremos permitir que el endpoint del back-end `/api/login` sea expuesto al front-end (que corre en un puerto distinto) necesitaremos indicarle a Spring Boot de qué lugar va a estar este front-end haciendo la petición: esto lo hacemos con la anotación `@CrossOrigin`, pasándole la IP y el puerto de la aplicación front-end (5500¹⁴) para que el controlador habilite conexiones entrantes desde ese origen.

Figura 3.5: Descripción de alguna de las funciones activadas ante un inicio de sesión que empieza por un usuario lanzando una petición POST a `/api/login`. Se utilizan todas capas del modelo: controlador, servicio, repositorio y modelo (clase `Usuari`).



Una de las partes fuertes de Spring Boot es que permite mapear objetos de la base de datos MySQL a objetos Java: esto se consigue en las clases del *model*, aquellas que llevan la anotación `@Entity`. En nuestro caso tenemos la clase `Usuari.java`, con la que podemos recoger en sus atributos una fila de la tabla `Usuari` de la base de datos MySQL. Esto es posible gracias a JPA (Java Persistence API), que define algo llamado ORM u *Object Relational Mapping*. En teoría, si se utiliza bien el ORM ya no es necesario hacer consultas con lenguaje SQL a la base de datos porque JPA ya lleva una interfaz de la que extendemos en el `UsuariRepository.java` que nos da acceso a funciones que por detrás ya implementan consultas SQL, dandonos la

¹⁴Esto es, básicamente, el front-end de vscode o el puerto de Nginx que hemos configurado en el Dockerfile para el despliegue en contenedores como veremos luego en el apartado 3.6.1 de contenerización.

prerrogativa de no salir del lenguaje Java y no tener que adentrarnos en el terreno del SQL. Es por esta razón que en UsuariRepositori.java se ha definido una interfaz que hereda de la interfaz JpaRepository. La definición de UsuariRepositori debe hacerse de forma muy específica y cuidadosa para que funcione: hay que pasarle primero el objeto con el que mapeamos la tabla mySQL (un objeto con la anotacion @Entity, de los que esté dentro del *model*, en este caso Usuari y definido como en la figura 3.6) y, segundo, pasarle la clave primaria de la tabla Usuaris: [ver línea de código](#); en nuestro caso hemos usado funciones predefinidas como *findById()*, que permiten obtener a un usuario a partir de su clave primaria desde la tabla mySQL mapeada; pero, a pesar de ello, muchas de las consultas se han definido manualmente con la anotación *@Query*, como vemos en [UsuariRepositori.java](#) en la figura 3.5, porque no siempre se ha encontrado una manera de usar las funciones de JPA correctamente.

A continuación, en la figura 3.6 explicamos la clase *Usuari.java*, perteneciente al *model* de la aplicación: es indispensable definirla así para el ORM.

Figura 3.6: Detalle de la correspondencia existente entre objeto Java (Usuari) y entidad de base de datos (tabla usuari, mySql). || *@Column* nos indica en el parámetro *name* el nombre exacto que tiene la columna de la base de datos a la que mapea el atributo contiguo a la anotación. | *@Table* mapea con ORM tabla usuari con clase Usuari. | *@Entity* habilita las anotaciones anteriores.

```
// (recordem que estem al paradigma ORM que és el que ofereix JPA). En aquest cas JPA no crea
// la taula de la base de dades ja que hem posat "spring.jpa.hibernate.ddl-auto=validate" a
// application.properties".
//
// En aquesta classe també creo els
// getters, setters i la funció to string per poder imprimir la informació de l'usuari.

@Entity
@Table(name = "usuari")      //indiquem el nom de la taula a sql
@NoArgsConstructor
@AllArgsConstructor
@Data
public class Usuari {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)      //ES GENERA INCREMENTALMENT
    @Column(name = "id_usuari")
    private Integer idUsuari;

    @Column(name = "correu_electronic", nullable = false) //FEM QUE SIGUI NOT NULL
    private String correuElectronic;

    @Column(name = "hash_contrasenya", nullable = false)
    private String hashContrasenya;

    @Column(name = "aliases")
    private String aliases;

    @Column(name = "permisos", nullable = false)
    private Byte permisos;

}
```

Para este proyecto solamente se utilizan tres endpoints de Spring Boot en las páginas públicas (como veremos en el diagrama 3.23). Sin embargo, el proyecto tiene una API REST bastante completa. Durante la realización de las prácticas en la empresa donde he estado estos tres meses y en paralelo al curso, una API rest completa fue creada; ésta se ha incluido también en el back-end de Spring boot aunque no se ha utilizado para el proyecto, todavía; pero podrá servir a futuro para expandir la aplicación, si se desea. Aquí un detalle de todos los controladores hechos y no utilizados, pensados para ser livianos, dejando toda la lógica de negocio en el Service y devolviendo códigos HTTP informativos ante los errores: [link a gitHub](#).

3.4.2.2. src/main/resources: archivos de configuración

Dentro de los archivos de configuración es necesario mencionar el archivo [application.properties](#): en él se define la conexión a la base de datos mySQL ([detalle líneas](#)), se pide que el ORM valide si las anotaciones de la clase Usuari coinciden con el DDL de la base de datos ([detalle líneas](#)) y se pide que se habilite otro perfil que va a sobrescribir las propiedades que ya estén escritas en *application.properties*: por ejemplo, nosotros hemos habilitado el perfil *application-prod.properties* con [esta linea](#). Por lo tanto, si ahora escribimos en el mismo árbol de directorio que *application.properties* otro archivo de configuración denominado *application-prod.properties* vamos a sobreescibir en *application.properties* aquellas propiedades que defina este nuevo perfil. Esto es útil, por ejemplo, para entornos de producción donde queremos cambiar la base de datos de una base de datos en un servidor y no en local.

3.4.2.3. pom.xml: dependencias de maven

Los proyectos java pueden correr en gradle o maven. En nuestro caso el proyecto se ha hecho con maven. En pom.xml (ver [aquí](#)) se describen todas las dependencias que tiene nuestro proyecto, conjuntamente con la versión de las mismas. Si queremos instalar una nueva dependencia en el proyecto basta con añadirla en el pom.xml y dentro de intelliJ hacemos click derecho *Maven → syncProject*. Entonces IntelliJ nos descargará las dependencias nuevas desde maven central (que es donde están todos los repositorios de Java).

Por ejemplo, para poder definir JWT o JSON Web Token hemos tenido que cargar esta dependencia específica en pom.xml (podeis ver más información en el apartado 3.4.3.3). Por ejemplo, para definir Lombok¹⁵ hay que definir este xml y, sobretodo, incluir la versión que utiliza la dependencia, tal que así:

¹⁵Es una dependencia muy usada en Spring Boot, que permite generar getters y setters para una clase, pero sin que estén escritos: así si se cambia el nombre de los atributos de una clase, los getters y setters cambian también de forma automática.

```

<!-- fes getters i setters automatics -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.36</version>
    <scope>provided</scope>
</dependency>

```

Para saber cuál es la versión más actualizada -o la más usada- de un plugin basta con entrar a mvnrepository.com y consultarla. Por ejemplo, para el caso de lombok tenemos este [link](#).

3.4.3. Autenticación y Autorización

3.4.3.1. método utilizado: JWT

Para autenticar y autorizar a los usuarios no utilizaremos sesiones. Las sesiones, tal y como vimos en la asignatura de desarrollo web entorno servidor, requieren guardar un estado en el servidor (si tenemos 100 usuarios conectados necesitamos rastrear 100 personas en el servidor) y un identificador de sesión en una cookie segura con HttpOnly puesto a True guardada en el navegador de cada uno de los usuarios conectados que lo identifica en relación al servidor.

Sin embargo, existe un método de acceso por token más escalable que no requiere guardar sesiones en el servidor (es decir, es un método “stateless” o sin estado) con el que nos basta tener solamente la Cookie Segura para guardarlo y ya está. Es un token que está autocontenido: es decir, puede contener ya el ID de usuario, nombre de usuario, roles que luego permitirán dar permisos o no en el servidor para acceder a determinadas APIs o recursos, etc. En definitiva, con JWT tenemos una autenticación más eficiente y un control del acceso preciso (autorización) sin necesidad de almacenar sesiones en el servidor.

A este sistema lo llamamos JSON Web Token (JWT) y toda la información que contiene está **firmada digitalmente** con SHA256 mediante una clave privada (el “secret”) que solo tenemos nosotros en el servidor¹⁶. Esta clave es igual para todos los tokens que generemos: la firma digital que emana de esta clave estará embedida, por así decirlo, en cada uno de esos tokens y *será inválida* si un atacante ha modificado el token y nos lo devuelve al servidor tratando de suplantar la identidad de algún usuario; con ello, el servidor rechazará la integridad del token y evitara que pueda acceder a recursos del usuario al que trata de suplantar.

JWT no es perfecto, por supuesto. Una desventaja del JWT es que una vez puesta

¹⁶El token está firmado, pero no cifrado: todo el mundo puede ver su contenido.

una fecha de expiración el desarrollador ya no la puede cambiar. En las sesiones del servidor se pueden extender las sesiones si se detecta actividad del usuario, acortarlas si pasa justo lo contrario o incluso cerrar la sesión de un usuario en remoto; pero con JWT no es posible: una vez creado el Token de acceso la fecha de actividad del mismo no se puede modificar (¡porque no puedes invalidar un token ya existente!), lo cual permite que simplemente un atacante robe el token de acceso sin modificarlo y lo use hasta su fecha de expiración.

Se proponen dos soluciones posibles a este problema, ninguna de las cuales ha sido implementada en este proyecto y queda definitivamente como uno de los puntos de mejora:

- 1. Tener dos tokens almacenados en el cliente: el “access token” que es el que permite autenticar y autorizar, del que hemos hablado hasta ahora; y otro token denominado “refresh token”, que se utiliza para obtener un nuevo token de acceso cuando el token de acceso actual expire, o para obtener tokens de acceso adicionales con una duración igual o más corta [7]
- 2. Crear una black-list de tokens de acceso donde se añadirán los usuarios que hayan hecho “log out” **antes** de la expiración programada de su token de acceso: así si un token de acceso todavía no expirado sabemos que su usuario se ha deslogueado, y en caso que sea robado el servidor podrá rechazar la petición no permitiendo acceso a recursos [8]) porque tendrá información de la voluntad de desloguearse por parte del usuario real.

En este trabajo solo utilizaremos “access tokens” y ya está. Cuando un usuario se desloguee, lo que haremos simplemente será borrar el access token del *local storage*. Tampoco guardaremos los tokens en una cookie segura porque complica el desarrollo (de nuevo, otro punto a mejorar a futuro en este proyecto).

En resumen, **las ventajas** que tiene JWT vs uso de sesiones (si asumiéramos que tanto el JWT como el SESSID se guardasen en una cookie segura, respectivamente) serían las siguientes:

- **No depende del almacenamiento en el servidor**
- **Firmado digitalmente**
- **Mayor control sobre el acceso**
- **Mayor descentralización**
- **Menos carga para el servidor**

Y las **desventajas** más evidentes que tiene JWT, en nuestra opinión, son ¹⁷:

- **Caducidad de tokens irrevocable**
- **Necesidad de manejar tokens de refresco + tokens de acceso**

¹⁷Se puede ver una tabla de diferencias más en profundidad, especialmente en materia de seguridad en el anexo 5.2)

3.4.3.2. ¿Qué compone un JWT?

El JWT se compone de tres partes separadas por *puntos*: **el header**, **el payload** y **la signatura**. En la página <https://jwt.io/>, como veremos después, se puede ver si los tokens son válidos, observar el contenido de su payload, etc. [9]. A saber:

- **Los headers:** Aportan información sobre el algoritmo que lo encriptó.
- **El payload:** Es donde está la información que nos interesa del token: el sujeto que lo generó (“sub”), el momento en que se generó el token (“iat”, o “issued at”) y la fecha de expiración (“exp” o “expiration time”). También podemos tener ahí otros pares clave valor que podremos querer definir, por ejemplo, que contengan el id del usuario y sus roles o permisos que son los que nos permitirán dejar que un determinado usuario pueda consultar o no ciertos recursos.
- **La signatura:** Es la parte que garantiza la integridad del token y evita que sea alterado por terceros. Se genera aplicando un algoritmo de hash (en nuestro caso el SHA256) a la combinación del header y el payload, junto con la clave secreta que solo conoce el servidor (es lo que permitirá al servidor rechazar el token si no es válido -i.e. ha sido manipulado).

Podéis observar estas tres partes en colores en la figura 3.7 que veremos después.

3.4.3.3. Implementación de JWT en java SpringBoot

Para poder implementarlo añadimos la dependencia **jjwt** en **pom.xml**:

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.12.6</version>
</dependency>
```

Usando en el proyecto esta dependencia se han creado tres clases dentro de sus respectivos archivos en la ruta **src/main/java/ miApp.app/seguretat/jwt** denominadas:

- **JwtUtil**
- **AccessToken**
- **RefreshToken**

En la clase **JwtUtil** hemos creado un método que obtiene las *claims* (pares clave valor que contienen la carga útil de un JWT) y en el constructor hemos creado la

definición de una clave privada con la que derivar todas las instancias que hagamos de esa clase -es decir, todos los tokens que se cifren con esa contraseña-. De esta clase hemos heretado las otras dos: la subclase que nos genera el *token de acceso*, **AccessToken**; y la que nos genera el *token de refresco*, **RefreshToken**¹⁸. A continuación podéis, de estas tres, ver la más importante:

```
@Component
public class AccessToken extends JwtUtil {

    private static int tExpM; //minuts per a exprimir el token

    public AccessToken() {this.tExpM = 10;}

    //FINALITAT: Generar un JWT d'accés.
    public String genera(String correu, int idUsuari, byte permisos) {
        Map<String, Object> dadesExtraApayload = new HashMap<>();
        dadesExtraApayload.put("permisos", permisos);
        dadesExtraApayload.put("idUsuari", idUsuari);

        return Jwts.builder()
            .setClaims(dadesExtraApayload) //dades customitzades
            .setSubject(correu)           //guardo nom subjecte (clau "sub")
            .setIssuedAt(new Date())      //data creacio (clau "iat" payload)
            .setExpiration(new Date(System.currentTimeMillis() + (tExpM*60*1000)))
            .signWith(SignatureAlgorithm.HS256, clauSecreta.getBytes())
            .compact();
    }
}
```

Con la función **genera()** de la clase **AccessToken** arriba mostrada, y con los parámetros necesarios para autorizar (**idUsuari**) y autenticar (**permisos**), podemos generar un token de acceso en “accesJWT” que es el usado en la aplicación para permitir acceder a los endpoints o no:

```
AccessToken accessToken = new AccessToken();
String accesJWT = accessToken.genera(
    "santo@gmail.com", //campoSub
    2, //idUsuari
    1 //permisos
);
```

Si vemos la figura 3.7 que tenemos a continuación, veremos en la mitad izquierda un token de acceso generado por la función anterior. Fijémonos que internamente ese token está estructurado en las tres partes mostradas en la mitad derecha de la imagen, siendo la parte “payload” la más importante, porque contiene la carga útil

¹⁸La clase RefreshToken la hemos programado para usarla a futuro pero no se ha utilizado para la implementación del sistema de autenticación y autorización en este proyecto.

que usaremos para autenticar y autorizar en mercApp:

Figura 3.7: Decodificación mediante jwt.io de un token de acceso usado en nuestra aplicación generado con la función “genera()” de la clase AccessToken. La Payload con las claims en flecha verde.

Encoded	Decoded
eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MSwiaWRVc3VhcmkiOjIsInN1YiI6InNhbnRvQGdtYWlsLmNvbSIzImlhCI6MTc0MzE1MjM1MCwiZXhwIjoxNzQzMTUyOTUwfQ.ixC6NKkuG99zrxLcrgxfjRRKi8NgYbUrirFgPMEcUC0	HEADER: ALGORITHM & TOKEN TYPE <pre>{ "alg": "HS256" }</pre> PAYOUT: DATA <pre>{ "permisos": 1, "idUsuario": 2, "sub": "santo@gmail.com", "iat": 1743152350, "exp": 1743152950 }</pre> VERIFY SIGNATURE <pre>HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-a8f7d9gbb6c3!) □ secret base64 encoded</pre>

```
{
  "permisos": 1,
  "idUsuario": 2,
  "sub": "santo@gmail.com",
  "iat": 1743152350,
  "exp": 1743152950
}
```

Al generar las tres clases hemos utilizado herencia porque la clave privada es la misma para ambos tipos de token (tanto el de acceso como el de refresco), mientras que los métodos para generar cada uno de los dos tipos de token cambian. En StackOverflow existe un debate para ver si hay que tener una clave privada distinta para cada tipo de token, por si el lector está interesado [10]. Después de ver la entrada en StackOverflow se ha optado por compartir claves para ambos tipos.

En la clase **JwtUtil** tenemos una función denominada `getClaims()` que es la que utilizaremos en el Service de nuestra aplicación para poder autenticar y autorizar usuarios. Las tres clases pueden ser consultadas en el anexo 5.3, en la carpeta del GitHub del proyecto (ver [link](#) o hipervínculos al principio del subapartado.)¹⁹

¹⁹Se recomienda encarecidamente al lector optar por esta última opción, dado que al poner las tres clases en anexo se omitieron las funciones main donde se testeaban las funciones de creación de tokens con control de excepciones, comentarios e imports por falta de espacio en el DIN A4.

3.4.3.4. Enviar por primera vez el Access Token hacia el front-end (registro)

Cuando el usuario consigue poner la contraseña correcta en la página de registro del front-end (`pas3C_crearContrasenya.html`), esta contraseña se manda conjuntamente con el correo electrónico²⁰ mediante una petición POST al controlador de endpoint `api/registraUsuari` de nuestro back-end de Spring Boot. Este controlador responde entonces mandando de vuelta al cliente el **token de acceso**, que se **guardará** en el localStorage del navegador del usuario.

Así las cosas, podemos testear que esto funciona como es debido haciendo una solicitud POST con la aplicación Postman^[11] al endpoint `api/registraUsuari` con un correo que no esté guardado en la tabla `usuaris`: por ejemplo, “`nuevoUsuario@gmail.com`”; y con una contraseña válida para ser guardada de acuerdo con nuestros requisitos de seguridad²¹: si todo va bien deberemos obtener la figura 3.8:

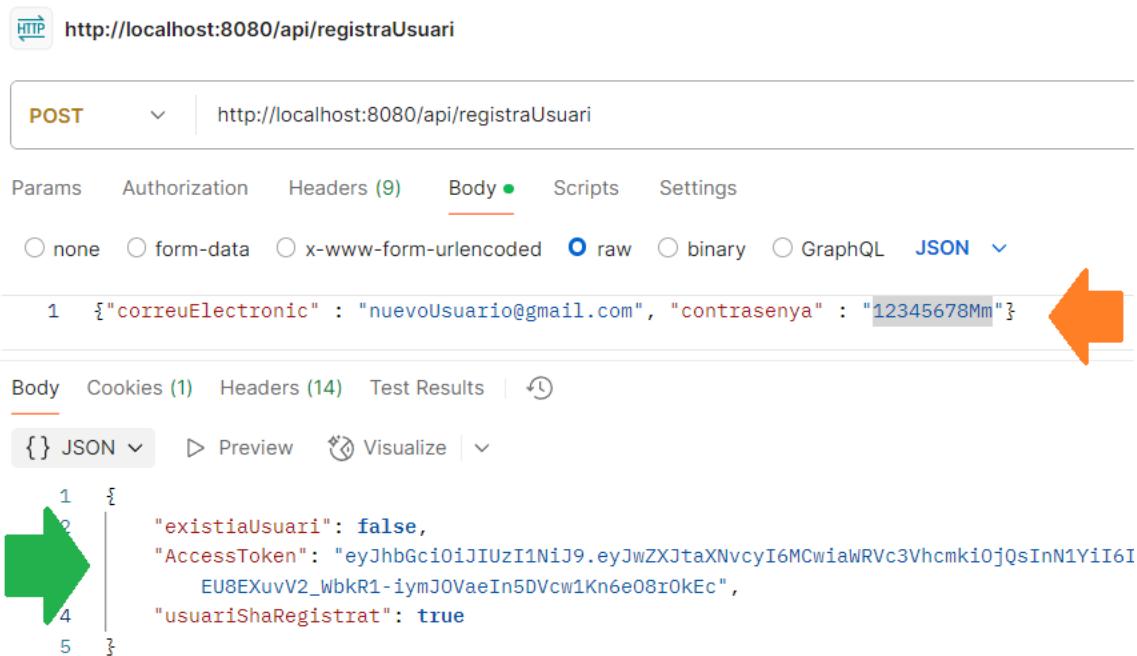


Figura 3.8: Creación de un nuevo usuario llamando con una solicitud POST al endpoint “`api/registraUsuari`” cuando las validaciones del objeto `RegistreDTO` del back-end lo permiten. En naranja se muestra el body de la petición (lo que el cliente envía al servidor) y en verde el body de la respuesta (lo que el servidor devuelve al cliente). NOTA: El código de estado no se muestra pero: es 200 OK.

Tenemos otro endpoint que también expide tokens de acceso, mucho más habitualmente que el anterior: es el endpoint que se consume cuando iniciamos sesión,

²⁰el correo electrónico lo teníamos guardado en el localStorage de la página de registro inicial.

²¹Mínimo 8 caracteres, una minúscula y una mayúscula y sin caracteres peligrosos.

en `pas2C_login.html`: el endpoint ubicado en la URI²² `/api/login`. Si intentamos iniciar sesión con un usuario ya existente en la tabla de usuarios obtendremos algo como esto:

```

POST http://localhost:8080/api/login
Params Authorization Headers (9) Body Scripts Settings
none form-data x-www-form-urlencoded raw binary GraphQL JSON
1 {"correuElectronic": "acces@gmail.com", "contrasenya": "12345678Mm_"}
Body Cookies (1) Headers (14) Test Results
{} JSON Preview Visualize
1 {
2   "usuari": {
3     "aliases": "blackcub3s",
4     "permisos": 1,
5     "idUsuari": 2
6   },
7   "existeixUsuari": true,
8   "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MSwiוארVc3VhcmkiOjIsInN1YiI6ImFjY2VzQGdtYwlsLmNvMqcDhr3cjhnaAetzBuKnn_DpbYsLkdtZtYF8VUcxZbM",
9   "teAccesArecursos": true,
10  "contrasenyaCorrecta": true
11 }

```

Figura 3.9: Iniciando sesión con un usuario ya existente mediante “`api/login`” mandando los datos de “`correuElectronic`” y “`contrasenya`” que leerá y validará el LoginDTO del back-end. En naranja es el body de la petición y en verde el body de la respuesta.

Por ejemplo podéis ver el endpoint -función de controlador- en GitHub pinchando en el link: [api/registra-usuari](#). Asimismo podéis hacer lo mismo para el endpoint [api/login](#). Todo ello son detalles de líneas de código pertenecientes a la clase Spring Boot [ControladorUsuaris.java](#)

NOTA: La parte de recepción del token en el front-end y de su manejo podéis encontrarla en el apartado 3.6.5

3.4.3.5. Recibir en el back-end el JWT enviado desde el front-end y con él securizar un endpoint de la API mediante interpretación de claims y verificación de firma.

Después de crear las tres clases en Java de las que hemos hablado en el apartado 3.4.3.3 anterior y haber mandado el token de acceso al front mediante el body de las *responses* a los endpoints `api/registraUsuari` o `api/login`, podemos empezar a implementar la protección o securización de endpoints con JWT.

²²Uniform Resource Identifier

Asumamos que nos llega al back-end un token de acceso en una solicitud HTTP de un usuario que ya ha recibido su token y quiere ahora borrar los datos de usuario de **otro** usuario, es decir, de alguien con un idUsuari distinto. Esto lo puede intentar variando la URI del endpoint DELETE (el token llega al servidor través de la header “Authorization” y dentro contiene el idUsuari real del usuario, pero el usuario pone en el URI de la solicitud **otro** idUsuari que no es suyo, tal que así: `/api/usuaris/idQueNoEsSuyo`). El token en este caso seguiría siendo válido, y dado que este usuario malicioso tendría dentro de las claims del token el mismo valor en la variable permisos que el usuario real propietario de los datos al que quiere borrárselos, podría hacer llamadas al endpoint y borrar sus datos: en Spring Boot esto hay que evitarlo mediante configuración manual, sino los endpoints de otros usuarios no estarán protegidos! En las próximas líneas veremos cómo se soluciona.

En JavaScript puro, desde el cliente, esta solicitud HTTP de método DELETE la podríamos conseguir con la función `fetch()`, poniendo el header “Authorization” a “Bearer” (por convenio con espacio después de “Bearer”) seguido del token²³:

```
fetch("http://localhost:8080/api/usuaris/{id}" , {
    method: "DELETE",
    headers: {
        "Content-Type": "application/json",
        "Authorization": "Bearer "+tokenJWT;
    },
    ...
})
```

Queremos conseguir que ese endpoint permita en cada solicitud **autenticarlo**, es decir, determinar que dice ser quien es mediante el hecho de encontrar en el token verificado su **idUsuari** correspondiente (y acceder a la información de sus tickets); y a su vez **autorizarlo**, es decir, dar acceso a ese usuario a los recursos a los que se le permita acceso mediante la variable **permisos** correspondiente.

Estos dos pasos irán en función del valor que haya emanado para cada variable de interés de la tabla usuaris de mySQL y que haya llenado las claims del token: el token incluirá **idUsuari** para el caso de la **autenticación** -ver [línea github](#)-, y también **permisos** para el caso de la **autorización**, siendo estas variables obtenidas de la BBDD a través del model de la @Entity class Usuari - ver [línea github](#)-. Para ello hay **tres** pasos que debemos implementar dentro del back-end de SpringBoot:

- **PASO 1:** Extraer la información del usuario autenticado desde *el payload* del token JWT entrante. Para ello crearemos un **Filtro de Autenticación** dentro de **FiltreAutenticacio.java**

²³Con postman pasamos el token desde la pestaña Authorization y en Auth Type le seleccionamos, en el desplegable, Bearer Token.

- **PASO 2:** Configurar el contexto de seguridad para que Spring Security reconozca los permisos, dentro de `ConfiguracioSeguretat.java`.
- **PASO 3:** Aplicar restricciones con `@PreAuthorize` en cada *endpoint* que queramos proteger en el controlador, dentro `UsuariControlador.java`

PASO 1: Extracción del payload (*FiltreAutenticació.java*)

Esta parte del código está llena de boilerplate. La clase `FiltreAutenticacióJwt.java` extiende de `OncePerRequestFilter` [12], que como dice el propio nombre de la clase implementa un filtro que se desarrollará una y solo una vez por cada petición al servidor.

Lo que hay que hacer aquí es implementar el método `doFilterInternal()` donde colocamos la lógica específica del filtro. Se puede consultar este archivo [FiltreAutenticacióJwt.java](#) en el GitHub del proyecto.

Lo primero que debemos tener en cuenta al diseñar la clase **FiltreAutenticacióJwt.java** es que tenemos que hacer una inyección de dependencias para que funcione: es decir, debemos incluir la clase `AccessToken` que hemos diseñado para implementar el token de acceso: Lo haremos simplemente incluyéndola en el constructor como un parámetro ([detalle](#)).

Lo segundo que hay que considerar es la extracción del payload del token (donde tenemos la información que nos permitirá autorizar y autenticar). Para encontrar el token se hace de la **cabecera** “Authorization” de la solicitud HTTP entrante del front-end. La clave es “Authorization” y el valor asociado es, por convenio, un String “Bearer ” concatenado al token de interés; algo así:

“Authorization” : “Bearer OJALWQ03P1WNOEGBO...”

La programación necesaria para conseguir lo mencionado en el párrafo anterior queda recogida en este rango de líneas de GitHub ([ver rango](#)). Luego una vez tenemos el token dentro de Spring Boot tratamos de sacar las Claims del Payload, es decir, la carga útil del token ([ver rango](#)). Y con ello ya podemos asignar tres roles a partir de la variable `permisos` del payload: 0, 1 o 2. Hay que dejar claro que:

- **0** → se asigna en la variable `permisos` cuando un usuario ya se ha registrado dando correo y contraseña, pero no ha dado acceso a tickets digitales todavía.
- **1** → se asigna en `permisos` cuando usuario ya se ha registrado y conseguido que el servidor haya guardado tickets en formato estructurado en mongoDB (eso le habilitará a poder ver el dashboard de la aplicación).
- **2** → se asigna en `permisos` cuando el usuario tiene acceso a consultar todos los recursos de la aplicación, tickets de los demás usuarios, etc. ([ver rango](#)):

Todo ello es posible gracias a las líneas de código que permiten esta asignación:

```
Claims claims = accessToken.getClaims(token);

Integer permisos = (Integer) claims.get("permisos");
Integer idUsuari = (Integer) claims.get("idUsuari");

// Creo autoritat basada en permisos
String role;
if (permisos == 2) {
    role = "ROLE_ADMIN";
} else if (permisos == 1) {
    role = "ROLE_USER";
} else {
    role = null;
}
```

Importante es mencionar que en el fragmento de código anterior, al llamar al método `getClaims(token)` se lanzará una excepción de tipo `ExpiredJwtException` en caso que el token haya expirado, que recogeremos en el primer bloque Catch; y si el token está manipulado y no es válido, entonces se lanzará otra excepción que se recogerá en el segundo bloque Catch. Todo ello se informará como una response al cliente ([ver rango de líneas de código](#)).

Y finalmente hay que crear un objeto de tipo `UsernamePasswordAuthenticationToken` ya definido dentro de Spring Boot. Su constructor permitirá tres parámetros:

- **el principal**, el primero, al que le pasaremos el `idUsuari`
- **credentials**, el segundo, que lo dejamos a null porque en JWT no se debe manejar credenciales ya que están contenidas dentro del token.
- **una collection con el role**, el tercero, que contendrá los roles que definimos antes.

Este constructor nos permitirá restringir permisos para las APIs según el `idUsuari` al que esté vinculado su login (autenticación) y también según el valor de `permisos` que tenga (autorización)²⁴.

```
UsernamePasswordAuthenticationToken authentication =
new UsernamePasswordAuthenticationToken(
    idUsuari,
```

²⁴Cuidado! Lo cierto es que deben considerarse ambos parámetros a la vez en el controlador, como veremos en el paso 3. No es suficiente añadir roles a un determinado id. Solamente con los roles, Spring Boot no nos dejará, por ejemplo, que en una API que toma el `idUsuari` como parámetro en la URL (como la de este ejemplo) se pueda restringir a ese usuario específico para que no consulte los recursos de los demás usuarios con `idUsuari` distintos.

```

    null,
    Collections.singletonList(new SimpleGrantedAuthority(role))
);

```

Este objeto *authentication* que acabamos de crear entonces tenemos que guardarlo **dentro** del *SecurityContextHolder*:

```
SecurityContextHolder.getContext().setAuthentication(authentication)
```

PASO 2: Configuración contexto de seguridad (*ConfiguracioSeguretat.java*)

Sin esta clase, al añadir la dependencia de seguridad “spring-boot-starter-security” en *pom.xml* cualquier llamada a cualquier API va a devolver un código de error 401. Esto pasa porque al añadir la dependencia de seguridad mencionada nos encontramos con que se precisan ciertas configuraciones. Las tres cosas que hay que hacer en la clase *ConfiguracioSeguretat.java* para llevar a término las mencionadas configuraciones son:

- 1. **Inyectar jwtAuthenticationFilter**, una instancia de *FiltreAutenticacionJwt.java*, a través del constructor para que actúe como dependencia.
- 2. **Especificar que jwtAuthenticationFilter va ANTES del filtro** estándar de Spring para autenticación por usuario/contraseña ([ver Línea de código](#))²⁵.
- 3. **Definir endpoints a restringir con requestMatchers**: por ejemplo, hay un endpoint que permite cambiar la contraseña de un usuario (una solicitud PATCH). Ese endpoint queda marcado [en esta línea de ConfiguracioSeguretat.java](#) y nos define que solo usuario de rol “ADMIN” (*permisos = 2*) y usuario de rol “USER” (*permisos = 1*) pueden hacer llamadas a él y conseguir cambiar su contraseña:

```
.requestMatchers("/api/*/contrasenya").hasAnyRole("USER", "ADMIN")
```

NOTA: La clase *ConfiguracioSeguretat.java*, en el momento de escribir estas líneas, podéis verla también en el anexo [5.4.1](#). Se recomienda ver el [link](#) actualizado de GitHub de este archivo.

PASO 3: Restricciones en el controlador

En el endpoint que acabamos de mencionar, todo el trabajo de autenticación y autorización que nos proponíamos *no está terminado*. Ahora mismo cualquier usuario con roles “USER” (*permisos = 1*) *todavía* puede cambiar la contraseña

²⁵JWT no funciona directamente con Spring Security: su implementación con Spring Security no es directa porque hay que añadir otra dependencia en *pom.xml*, la dependencia “io.jsonwebtoken”. De ahí que debamos decirle a Spring Boot que la utilice no de la forma estandar que tiene spring security.

de cualquier otro usuario. Si este usuario (al que llamaremos X) desea cambiar la contraseña del usuario con idUsuari = 31, por ejemplo, solo deberá mandar una solicitud PATCH dirigida a “`/api/31/contrasenya`” incluyendo el token de acceso de X (sí, aunque su idUsuari sea distinto) con Postman.

¿Esto sería inadmisible, verdad? ¡Si uno fuese el usuario de idUsuari 31 no le haría mucha gracia que otro usuario pudiera cambiar su contraseña! ¡No parece un buen diseño de seguridad!

Para evitarlo no nos queda otra que afinar a nivel de controlador con una anotación denominada `@PreAuthorize` mediante la cual indicamos a ese endpoint de controlador en específico de quién permitir solicitudes. Con esta anotación, pasándole los parámetros adecuados, conseguiremos restringir que *solo*²⁶ un `idUsuari` pueda acceder a él (este usuari será el del `principal`²⁷, el id propio). El uso de la anotación `@PreAuthorize` solamente se habilita por parte del framework si en la clase anterior del paso dos añadimos la anotación `@EnableMethodSecurity` encima del encabezado de la clase `ConfiguracioSeguretat.java` ([link a línea](#)). Una vez añadida la anotación `@EnableMethodSecurity` ya podemos ir a `UsuariControlador.java` y añadir la anotación `@PreAuthorize` encima de la función correspondiente del endpoint en cuestión ([ver en contexto](#)), tal que así:

```
@PreAuthorize("hasRole('ADMIN') or #id == principal")
```

NOTA: Podéis ver la función completa del controlador en el anexo [5.4.2](#)

3.4.4. Validación de datos (endpoints back-end)

NOTA: Los datos validados en el back-end siguen las mismas expresiones regulares y restricciones que las validaciones hechas en el front-end (ver sección [3.6.6](#)).

Los endpoints del back-end a los que apuntamos con llamadas fetch desde los campos de formulario de correo electrónico y contraseña desde el HTML deben protegerse también en el back-end, no solamente en el front.

El motivo de ello es porque no podemos permitir que entren unos datos no validados (nulos, con caracteres peligrosos, etc.) a través de **llamadas directas a la API**. Hay que tener mucho cuidado con esto!

La parte donde definimos la validación de datos de los endpoints de entrada de los datos de usuario está en `utils`²⁸ pero se produce parte de validación a través de

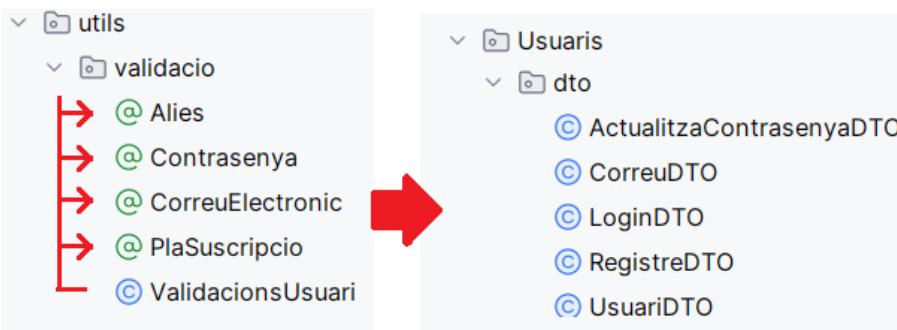
²⁶A menos que se sea ADMIN, en cuyo caso no importará el idUsuari del ADMIN (`permisos = 2`) y libremente podrá cambiar cualquier contraseña.

²⁷El principal era el idUsuari que pasamos al crear el objeto `UsernamePasswordAuthenticacionToken` dentro del paso1 en `FiltreAutenticacio.java`.

²⁸podéis ver la imagen completa de la estructura del proyecto SpringBoot si queréis, de nuevo, en [3.2](#)

Usuaris → *dto*. Una primera aproximación o esquema a entender cómo se relacionan entre ellos estos dos tipos de clases está en la figura 3.10.

Figura 3.10: **Izquierda:** Archivos de clases de anotación (@), la clase donde se definen mensajes de error que aprovecharán las clases de anotación | **Derecha:** los DTOs o *Data Transfer Objects* que es donde se aplican las validaciones definidas en las clases de anotación.



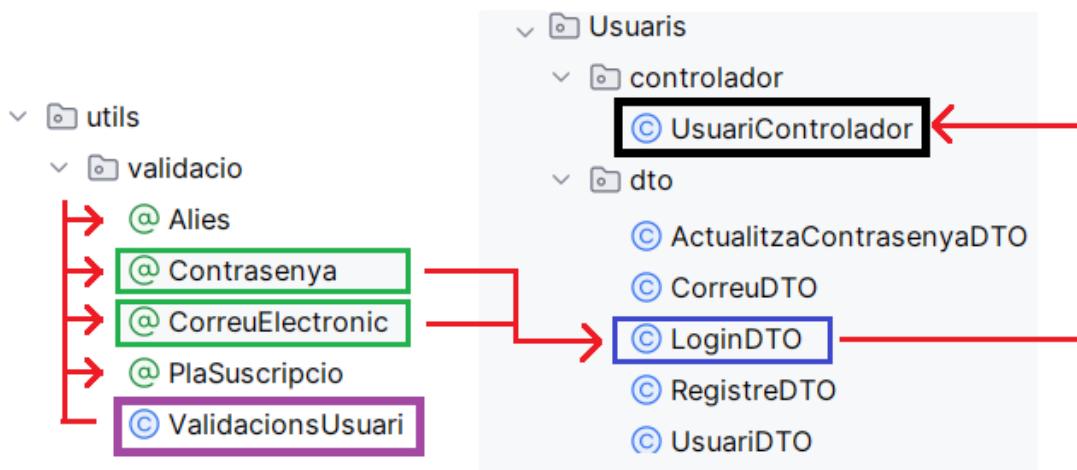
Vamos a hablar en términos concretos. Pongamos por caso que queremos validar los datos de entrada que llegan a través del endpoint que recibe la llamada POST del front-end para el inicio de sesión, es decir, el endpoint “[api/login](#)” del archivo [ControladorUsuari.java](#) de Spring Boot.

Debemos entender que en este caso, cuando se haga una llamada exitosa a ese controlador, tendremos un *body* de la petición del estilo siguiente:

```
{"correuElectronic" : "acces@gmail.com", "contrasenya" : "12345678Mm\_"}
```

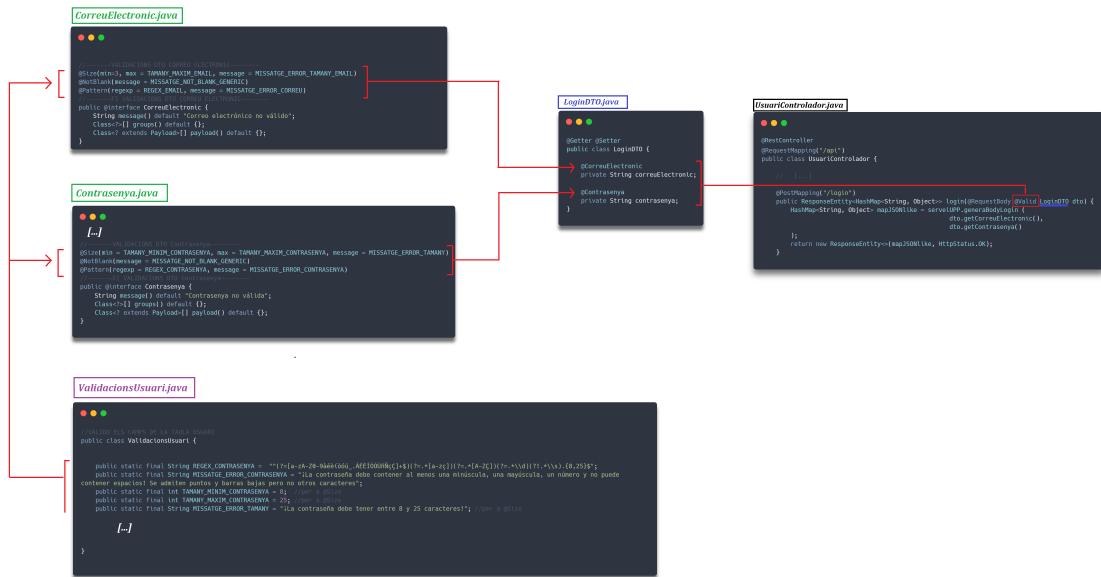
Según los datos entrantes esperados, las clases utilizadas y activadas para validar estos datos de entrada serán las de la figura 3.11:

Figura 3.11: Activación de clases de validación para responder a una solicitud POST de inicio de sesión hecha hacia el endpoint [/api/login](#) del controlador [UsuariControlador.java](#).



Asimismo, en la figura 3.12 podréis ver exactamente qué es lo que está pasando para la llamada a este endpoint. Podréis ver en esencia como existe una clase *LoginDTO*²⁹ que permite hacer la validación de datos cuando un usuario inicia sesión, a través de anotaciones personalizadas (*@CorreuElectronic* y *@contrasenya*) que a su vez llaman a otras anotaciones propias del framework springboot (*@pattern*, para aplicar expresiones regulares; *@NotBlank* para evitar que el campo esté vacío, etc.):

Figura 3.12: Veremos que en *UsuariControlador.java* (imagen de la derecha) el tipo de datos que espera el endpoint *api/login* se guarda en el parámetro de entrada *dto*. Esto pasa gracias a la anotación **@RequestBody** que mapea la entrada en formato JSON con el parámetro de entrada que esté a su derecha: en este caso *dto*. *dto* tiene que tener un formato compatible con los datos JSON entrantes por el body de la solicitud POST, obviamente. El *dto* está tipado con *LoginDTO*, que es una clase que hemos definido nosotros y permite recoger los datos entrantes haciendo un match entre los nombres de las claves del JSON y los nombres de los atributos de la clase. Esta clase también permite definir las validaciones mediante anotaciones personalizadas, que están justamente encima de cada atributo y permiten validar los datos para ese atributo. Estas anotaciones en *LoginDTO* se activan si y solo si *UsuariControlador.java* tiene definida la anotación **@Valid** al lado del parámetro *dto*. Las dos anotaciones personalizadas de *LoginDTO* son **@Contraseña** y **@CorreuElectronic** que llaman asimismo a otras clases de anotación ya definidas del lenguaje: *@pattern*, para aplicar expresiones regulares o *@NotBlank* para evitar que el campo esté vacío



²⁹Los data transfer objects o DTOs son simplemente clases Java que mediante sus atributos definen los nombres exactos que tienen que tener las claves entrantes del body de las solicitudes POST y, además, aplican restricciones a los valores que éstas llevan asociadas según unas anotaciones definidas encima, que asimismo llaman a otras anotaciones de las clases de anotación, si y solo si, añadimos la anotación **@Valid** antes del parámetro *dto* del *UsuariControlador*.

Podría parecer complejo hacer lo que hemos hecho en la figura 3.12, pero una vez guardarmos campos en la base de datos a través de múltiples endpoints que requieren modificaciones parciales de esos campos hacerlo así tiene utilidad y es una buena práctica en Java.

Todos los endpoints de la clase `UsuariControlador.java` están protegidos con validaciones para evitar que entren caracteres problemáticos, como podéis ver en las expresiones regulares definidas en [ValidacionsUsuari.java](#)

3.4.5. Hasheado de contraseñas

Las contraseñas de los usuarios no se han guardado en texto plano. Por seguridad, se han guardado en forma de hash, es decir, con encriptación unidireccional. A tal efecto se ha utilizado la librería bcrypt de Spring security^[13] y se ha hecho una clase [EncriptaContrasenyes.java](#) que ha permitido envolver con nombres más pedagógicos las funciones de bcrypt que han sido usadas (ver figura 3.13).

Figura 3.13: Clase `EncriptaContrasenyes.java`, utilizada para encriptar contraseñas y comparar hash encriptados.

```
public class EncriptaContrasenyes {
    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();

    //PRE: una contraseña.
    //POST: el hash de la contraseña.
    public String hashejaContrasenya(String contrasenya) {
        return passwordEncoder.encode(contrasenya);
    }

    //PRE: una contraseña plana i una contraseña hashejada
    //POST: retorna true si la contraseña plana, un cop hashejada coincideix amb la contraseñaHash.
    public boolean verificaContrasenya(String contrasenyaPlana, String contrasenyaHash) {
        return passwordEncoder.matches(contrasenyaPlana, contrasenyaHash);
    }
}
```

Cuando un usuario se registre o inicie sesión, de la clase [EncriptaContrasenyes.java](#) se invocarán los métodos `hashejaContrasenya()` o `verificaContrasenya()`, respectivamente ((figura 3.13)). Estas funciones convenientemente guardan, con un nombre más agradable, las funciones de librería Bcrypt `encode()`³⁰ y `matches()`³¹

Esto se hará en la clase de servicio `UsuariServei.java`. Por ejemplo, para el inicio de sesión el uso de `matches()` a través de `verificaContrasenya()` lo encontraremos en esta línea de código de dicha clase: [link](#).

³⁰Para obtener el hash de una contraseña creada.

³¹Para verificar si una contraseña plana es coincidente con el hash que se guardó en base de datos en el momento del registro.

¡Es interesante hacer notar que una misma contraseña encriptada varias veces por la función *encode()* (envuelta en *hashejaContrasenya()*) produce hash distintos! Es decir, el hashing no solo imposibilita hacer un vistazo a la base de datos y ver las contraseñas de los usuarios, sino que también impide ver si dos usuarios tienen la misma contraseña. Esto también hace que, para un mismo usuario, no podamos comparar con una función simple de comparación de strings el hash guardado de una contraseña en el momento del registro con el hash generado por un logueo. Por eso, a tal efecto, nos vemos obligados a usar el método *matches()*.

3.5. Desarrollo back-end (FastAPI, Python)

3.5.1. Contenerización

Este microservicio lo contenerizaremos gracias a este [Dockerfile](#). Para hacerlo, usaremos el Dockerfile para crear una imagen desde la imagen `python:3-11 alpine`³² descargada del registry de docker (ver nota sobre instalación de docker³³).

Para crear la imagen con este Dockerfile, para hacer pruebas y ver como se despliega el contenedor, el lector puede probar de crear una imagen denominada “back-end-fastapi”. Para hacerlo puede moverse a donde está el Dockerfile y ejecutar el siguiente subcomando de Docker (*build*), tal que así:

```
docker build -t back-end-fastapi .
```

Ahora la imagen “back-end-fastapi” ya está creada y con docker images veremos que así es (figura 3.14):

Figura 3.14: comando docker images para ver las imágenes creadas

PS C:\Users\dilap\Mi unidad__SAMPLES\mercApp\APP WEB__FastAPI__> docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
back-end-fastapi	latest	8cfffe7fa9ac6	7 minutes ago	130MB

Acto seguido, una vez ya creada la imagen que contendrá todo el sistema de archivos de la aplicación con Python y sus dependencias (*pip install ...*³⁴), debemos usar esta imagen para derivar de ella un contenedor (es decir, crear una instancia de

³²Las imágenes alpine son ligeras: pesan 100 o 200MB a diferencia de las que provienen de la imagen entera de Python, que pesan más de 1GB.

³³Instalar docker en linux no se puede hacer con un solo comando. Si el lector lo quiere instalar y usa linux, le facilito el script que programé hace un tiempo, para poder instalarlo sin preocuparse de nada: [link](#). Si el lector usa Windows solamente debe preocuparse de instalar docker desktop y asegurarse que se añade la docker CLI para poder usar comandos desde las terminales disponibles en el sistema.

³⁴Pip es a Python lo que Maven es a Java, para entendernos: una especie de gestor de paquetes o dependencias.

esa imagen). Crearemos el contenedor de nombre *contFastApi* a partir de la imagen anterior y le pediremos que exponga el puerto 8000³⁵ -puerto con el que se comunica FastAPI en el interior del contenedor- y haremos que exponga ese puerto también al exterior del contenedor³⁶. Podemos hacerlo con run o con create. En este caso con create porque no queremos todavía arrancar el contenedor:

```
docker create -p 8000:8000 --name contFastApi back-end-fastapi
```

Una vez creado el contenedor ya podemos arrancarlo. Aquí ya no hay que definir puertos nuevamente porque ya se definieron al crear la imagen. Es también opcional usar las flags -a³⁷ e -i³⁸:

```
docker start -ai contFastApi
```

Ahora el contenedor se podría acceder a través de otro contenedor. Nótese que en esta línea del dockerfile hemos definido el host de la aplicación FastAPI desde dentro del contenedor para que sea la IP 0.0.0.0 . No hemos usado la IP loopback por defecto (localhost, loopback o 127.0.0.1), sino la IP “wildcard” o dirección de red (0.0.0.0). Si usáramos la localhost dentro del contenedor la aplicación estaría usando la dirección de dentro de la red del contenedor, pero no saldría fuera de este y sería inútil. La dirección 0.0.0.0 nos permite justamente que podamos acceder a los endpoints de FastAPI desde el localhost del sistema host no solo desde el localhost del propio contenedor. Es decir, nos permite que podamos acceder desde el navegador de nuestro ordenador -desde fuera del contenedor- o desde otros contenedores que estén corriendo en nuestra máquina. Véase en la figura 3.15 siguiente lo que puede hacer nuestro contenedor con esta configuración y como se construye, de forma más pormenorizada:

³⁵Hace referencia al 8000 que va a la **derecha** de los dos puntos.

³⁶Es el 8000 emplazado a la **izquierda** de los dos puntos.

³⁷Simplemente veremos lo que imprima la terminal

³⁸Nos permitiría interaccionar con el programa a través de la terminal si fuese necesario

Figura 3.15: definición de un endpoint de FastApi -¡no es un endpoint usado en nuestro proyecto, cuidado! Perteneces a la fase de pruebas!- que correrá dentro del contenedor Docker contFastApi (*subcaptura A*). | Configuración del comando para correr FastAPI dentro de los contenedores que se instancien a partir de la imagen del Dockerfile, con la dirección de red 0.0.0.0 que permite exponer endpoints de contenedor hacia fuera del contenedor (*subcaptura B, rectángulo rojo*). También podemos ver la configuración de que FastAPI correrá en el puerto 8000 **dentro** del contenedor instanciado (*subcaptura B, rectángulo verde*). | Finalmente, podemos ver también el comando que crea un contenedor desde la imagen, pidiéndole que escuche en el puerto 8000 de la red interna del contenedor (*subcaptura C, rectángulo verde*) y lo de ahí lo saque al puerto 8000 (*subcaptura C, rectángulo naranja*), siendo el resultado de la exposición del endpoint observable en nuestro equipo anfitrión del contenedor en el localhost en ese mismo puerto 8000 (*subcaptura D, rectángulo naranja*).

```

A) # ENDPOINTS PROVA RESTCLIENT -----
@app.get("/api/usuari/{id}")
def mostraUsuari(id):          # id es enter
    return {"dadesUsuari": "les dades de

B) Dockerfile > ...
22   CMD ["uvicorn", "controlador:app", "--host", "0.0.0.0", "--port", "8000"]

C) $ crealmatge_i_arancaContenidor.sh
11  #creo contenedor contFastApi des de la imatge back-end-fastapi creada.
12  #redirccione port 8000 (dreta dels dospoints) al port xxxx (esquerra).
13  docker create -p 8000:8000 --name contFastApi back-end-fastapi
14  docker start contFastApi

D) localhost:8000/api/usuari/1
Impresión con sangría □
{"dadesUsuari": "les dades de l'usuari 1 ASD."}

```

A) Código Python de FastAPI que define un endpoint para obtener datos de un usuario.

B) Fragmento de Dockerfile que ejecuta el comando `uvicorn` con el argumento `--host` establecido en `0.0.0.0` y el argumento `--port` establecido en `8000`.

C) Script `crealmatge_i_arancaContenidor.sh` que crea un contenedor Docker llamado `contFastApi` basado en la imagen `back-end-fastapi`, y lo hace escuchar en el puerto `8000` del host.

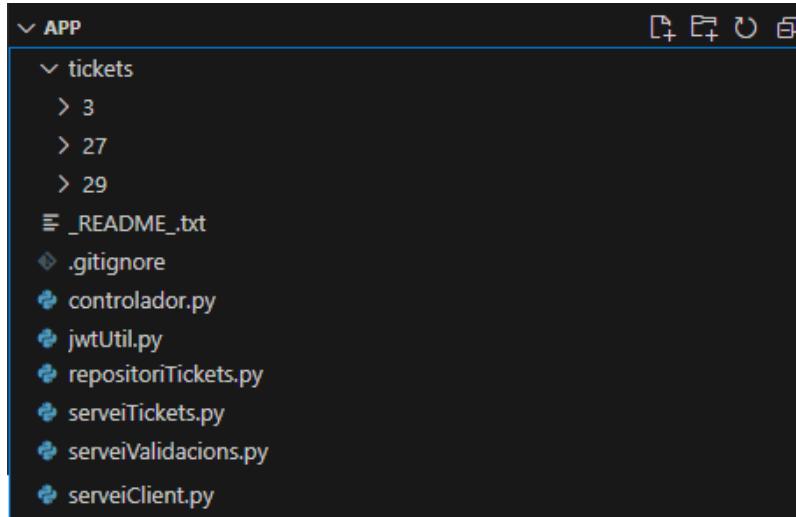
D) Captura de pantalla de un navegador web mostrando la respuesta del endpoint `/api/usuari/1` de la aplicación FastAPI.

Permite que la app esté accesible desde fuera el contenedor
(en este caso desde la loopback address del host donde se ejecuta)

Para automatizar el proceso de crear imagen con FastAPI y arrancar contenedor desde esta imagen y, finalmente, exponer los endpoints de FastAPI fuera del contenedor, puede usarse el script en bash `crealmatge_i_arancaContenidor.sh`, que incluye todos los comandos necesarios (inclusive aquellos para parar y destruir contenedores antiguos).

3.5.2. estructura de la aplicación

Figura 3.16: La estructura escogida simula un proyecto de Spring Boot: no es la arquitectura que se utiliza por defecto en FastAPI; pero dado que estamos acostumbrados a Java así se ha hecho. Todos los archivos están en una misma carpeta -que es evidentemente mejorable a nivel de patrón de diseño-.



En la figura 3.16 podemos ver los archivos Python (extensión .py). El [controlador.py](#) incluye los endpoints (la API de los tickets); [serveiTickets.py](#), [serveiValidacions.py](#) y [serveiClient.py](#) implementan la lógica de negocio usada para parsear tickets en PDF, validar sus tamaños y nombres de archivo, y para hacer llamadas HTTP al otro back-end -Spring Boot-, **respectivamente**. Finalmente la clase de servicio que extre los tickets ([serveiTickets.py](#)) hace uso de [repositoriTickets.py](#), que es donde definiremos las operaciones de persistencia para guardar y leer tickets de mongoDB. [jwtUtil.py](#) se ha utilizado para leer los tokens de acceso. La carpeta *tickets* incluye subcarpetas de creación automática que creamos programáticamente cuando los usuarios suben los tickets en PDF físicos al sistema de archivos del servidor (como veremos es un paso que precede a la extracción con parseo a formato estructurado y posterior guardado en MongoDB). Nótese que aquí no usamos una clase dentro de “model” ni tomamos uso de ningún sistema ORM, porque Python no es un lenguaje tan orientado a objetos como Java y haberlo incluido hubiese ido en contra de la simplicidad que ya nos brinda el lenguaje.

NOTA: Para más información de la estructura “controller” (controlador), “service” (servicio), “repository” (repositorio) y “model” (modelo) podéis ver el apartado de Spring Boot donde se comenta más en detalle este tipo de arquitectura 3.4.2.1.

3.5.3. Solicitud de subida y parseo de datos

NOTA: las peticiones POST con JavaScript desde el front-end hacia los endpoints de subida de PDFs y de parseo de PDFs, que mostraremos en las próximas líneas del lado del servidor, ya fueran mencionadas desde el lado del front-end en la PARTE 2 y 3 del apartado 3.6.9.2, dedicado exclusivamente a front-end.

A continuación mostramos los 4 pasos o partes, que deberán ejecutarse **secuencialmente** y **con éxito** para que un usuario durante el proceso de registro pueda adjuntar tickets y luego se parseen y guarden en la base de datos. Las partes 1, 2, 3 y 4 que mostraremos a continuación tienen correspondencia con las letras *f*, *g*, *h*, y finalmente, *j* de la figura esquemática anterior 2.2, que suponía el esquema de sistemas de la aplicación en la fase de registro del usuario en el apartado preliminar de diseño:

- **PARTE 1:** FastAPI recibe la POST **request** en el endpoint [/api/subir-tickets-pdf](#) con los tickets adjuntos desde el cliente y con el *token de acceso* con permisos a 0 [3.5.3.1]³⁹.
- **PARTE 2:** FastAPI recibe una solicitud POST en el endpoint [/api/parsea-y-guarda-pdfs-en-bbdd](#) que consigue que el servidor parsee con un algoritmo de extracción todos los tickets subidos en PARTE 1 [3.5.3.2].
- **PARTE 3:** FastAPI manda a MongoDB los datos de los tickets parseados [3.5.3.3] (mientras dura solicitud HTTP de PARTE 2)
- **PARTE 4:** FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot y, después de poner permisos a 1, le manda de vuelta un nuevo token (con permisos a 1). FastAPI transmite token y datos de tickets al navegador [3.5.3.4].

NOTA: A modo de recordatorio, hay que decir que después de la PARTE 4, al recibir en el front-end el nuevo token de acceso con permisos a 1, la página [pas4_concedirAccesGmail.html](#) redirigirá automáticamente al dashboard sin que tengamos que hacer nada más (vuélvase a ver lógica de redirecciones en la figura 3.26).

3.5.3.1. PARTE 1: FastAPI recibe la POST request en el endpoint [/api/subir-tickets-pdf](#) con los tickets adjuntos desde el cliente y con el token de acceso con permisos a 0

Con la solicitud HTTP mencionada hacia el endpoint [/api/subir-tickets-pdf](#)⁴⁰ lo que consigue un usuario es adjuntar los PDFs del ordenador del usuario al sistema

³⁹0 es el caso de uso más frecuente, pero también podría ser 2 si fuese admin. Ese token lo expidió Spring Boot al acceder a pas4, pero ahora lo usa fastAPI para permitir acceso a su API.

⁴⁰Recordamos que se produce al clicar al ícono del clip en el pas4.

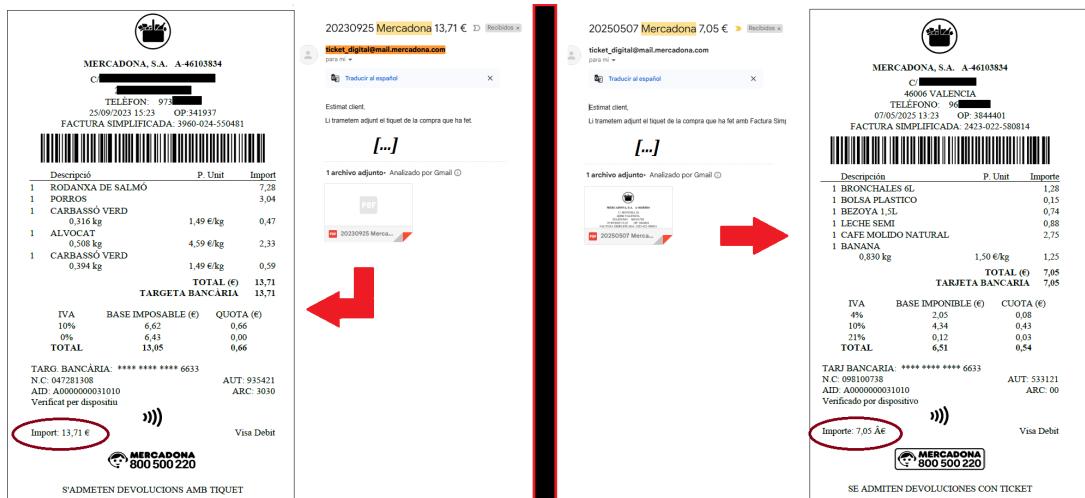
de archivos del servidor (la validación de datos de este endpoint se especifica en sección 3.5.5).

3.5.3.2. PARTE 2: FastAPI parsea todos los tickets con un algoritmo de extracción

Para iniciar el parseo de ficheros se empieza con una llamada POST al endpoint [/api/parsea-y-guarda-pdfs-en-bbdd⁴¹](#). El algoritmo de extracción se encuentra en el archivo `serveiTickets.py` al que llamamos desde el endpoint mencionado y su programación ha sido posible gracias a la existencia de la librería PyPDF2 [14].

El correo electrónico más antiguo y más nuevo contienen tickets que prácticamente no difieren. Por ello el algoritmo de extracción servirá para todos los tickets de Mercadona: como podemos ver en la imagen 3.17.

Figura 3.17: A la **izquierda**: la primera compra en Mercadona que hice en 2023 con ticket digital, en un supermercado en Catalunya; a la **derecha** la última compra que se hizo: en la Comunitat Valenciana. Nótese que en la extracción hay que tener en cuenta el distinto idioma que se puede dar en distintas comunidades autónomas (Euskadi, Galicia no han sido testeadas). La dirección de correo desde la que se mandan los tickets digitales no ha cambiado en dos años y el formato general del ticket es exactamente igual en términos de parseo: hay que prestar, empero, atención a posibles problemas con UNICODE (círculo marrón) en tickets antiguos.



Por ejemplo, el parseo del ticket de la derecha de la figura anterior 3.17 se convertirá en formato JSON después del parseo. Para conseguirlo, se introduce a un diccionario⁴² de Python que luego se podrá guardar directamente en una colección

⁴¹Recordamos que esta llamada se hace con el click al engranaje del pas4.

⁴²Un diccionario en Python es como un HashMap de Java o un Javascript Object: con una estructura de pares clave:valor.

de MongoDB con un formato estructurado compacto (figura 3.18) o ampliado si queremos más legibilidad (figura 3.19)⁴³.

Hagamos lo que hagamos vamos a recorrer todas las líneas con la función *fesScrapTicketMercadona()* del fichero [serveiTickets.py](#) y extraeremos los datos más importantes del ticket. Para hacerlo debemos tener en cuenta que el layout del ticket es distinto si tratamos de parsear un producto a granel o bien si es un producto que se vende a un precio unitario: los productos a granel, a diferencia de los que no lo son, ocupan dos líneas en el ticket porque añaden una línea más donde se añaden el precio por kg y el precio total. Además, otra diferencia, es que en la misma línea del nombre del producto no sale el precio o importe (porque sale en la segunda).

Esto es una fuente de variabilidad que encontraremos en cada ticket y se tendrá que tener en cuenta al tratar el string del ticket. Dentro de la función *fesScrapTicketMercadona()* vamos a tener en cuenta estos aspectos y los vamos a comentar en el código.

Una parte amable del ticket es que ya incluye, en principio, una clave candidata a ser clave primaria. Si concatenamos la factura simplificada con OP (número de operación) deberíamos tener algo que nunca -salvo rarísimas excepciones- se repite.

⁴³MongoDB no guarda JSON exactamente. Lo que hace es guardar objetos BSON (binary JSON) que son una representación binaria del JSON, pero que en esencia está optimizada para rendimiento en operaciones de consulta y escritura y, también, para que ocupe menos espacio que un JSON no binario (texto plano).

Figura 3.18: Primera aproximación (naif) a un formato compacto en el que podríamos haber parseado cada ticket digital de Mercadona para luego persistir en MongoDB y habilitar consultas eficientes. |Dentro de cada producto hay una lista que indica respectivamente si el producto es granel (0 si no lo es, 1 si sí lo es). Entonces, el segundo elemento de la lista indica el precio unitario o el precio por kg (en función de si es granel o no lo es, respectivamente); el tercer elemento, indica el número de unidades compradas o el número de kg comprados (en función del primer parámetro); el cuarto elemento es la categoría. En la próxima figura mostraremos la versión final que usamos en el proyecto, en donde todo es más explícito, con diccionarios para la información de cada producto en lugar de listas sin explicación alguna y con un booleano esGranel en lugar de 0 y 1.

A screenshot of a terminal window with a black background and white text. The window has three colored window control buttons (red, yellow, green) at the top left. The terminal content is a JSON object representing a shopping cart. It includes fields for the ticket ID, user ID, a products object mapping item names to arrays of four values, a total price, a store address, and a date. The JSON is formatted with commas and colons, and some values like the date are enclosed in quotes.

```
{
  "IdTicket": "2424-022-580814_OP3844401",
  "idUser": 10,
  "productesAdquirits": {
    "BRONCHALES 6L": [0, 1.28, 1, 13],
    "BOLSA PLASTICO": [0, 0.15, 1, 13],
    "BEZOYA 1,5L": [0, 0.74, 1, 13],
    "LECHE SEMI": [0, 0.88, 1, 13],
    "CAFE MOLIDO NATURAL": [0, 2.75, 1, 13],
    "BANANA": [1, 1.50, 0.83, 13]
  },
  "totalTicket" : 7.05,
  "direccioSuper": "C/ VALENCIA, 46006 VALENCIA",
  "fecha": "2025-05-07" //format aaaa-mm-dd
}
```

Figura 3.19: Formato en el que parsearemos cada ticket digital de Mercadona para poder guardarlo en MongoDB y habilitar consultas eficientes formato con claves descriptivas por cada producto (items sin clasificar). “*“esGranel”*”: booleano autoexplicativo | “*“preuUnitari”*”: €/unidad. (si no granel) o €/kg (si granel) | “*“quantitat”*”: número unidades (si no granel) o número de kg (si granel) | “*“categoria”*”: de 0 a 13, véase diccionario clasificatorio | “*“importe”*”: €, *“quantitat”***“preuUnitari”* (parseado).

```
{
    "_id": "2423-022-580814_0P3844401",
    "idUsuari": 3,
    "productesAdquirits": {
        "BRONCHALES 6L": {
            "esGranel": false,
            "preuUnitari": 1.28,
            "quantitat": 1,
            "categoria": 13,
            "import": 1.28
        },
        "BOLSA PLASTICO": {
            "esGranel": false,
            "preuUnitari": 0.15,
            "quantitat": 1,
            "categoria": 13,
            "import": 0.15
        },
        "BEZOYA 1,5L": {
            "esGranel": false,
            "preuUnitari": 0.74,
            "quantitat": 1,
            "categoria": 13,
            "import": 0.74
        },
        "LECHE SEMI": {
            "esGranel": false,
            "preuUnitari": 0.88,
            "quantitat": 1,
            "categoria": 13,
            "import": 0.88
        },
        "CAFE MOLIDO NATURAL": {
            "esGranel": false,
            "preuUnitari": 2.75,
            "quantitat": 1,
            "categoria": 13,
            "import": 2.75
        },
        "BANANA": {
            "esGranel": true,
            "preuUnitari": 1.5,
            "quantitat": 0.83,
            "categoria": 13,
            "import": 1.25
        }
    },
    "totalTicket": 7.05,
    "direccioSuper": "C/ SENYERA 24 46006 VALENCIA",
    "data": "2025-05-07",
    "hora": "13:23"
}
```

3.5.3.3. PARTE 3: FastAPI manda a MongoDB los datos de los tickets parseados

Mientras dura la llamada POST al endpoint [/api/parsea-y-guarda-pdf-en-bbdd](#), que iniciamos en la PARTE 2, va a acontecer el guardado de los datos extraídos a formato estructurado a una base de datos NOSql: MongoDB, como vimos en la figura 3.18. Esto nos habilitará a hacer búsquedas eficientes.

Desde el algoritmo de extracción que ya hemos mencionado ([serveiTickets.py](#)) podemos ver que hacemos llamada a funciones que residen en la capa de persistencia de nuestro back-end de FastAPI [repositoriTickets.py](#), que es la responsable de guardar y extraer los datos hacia o desde MongoDB.

NOTA: *Este paso se puede alargar en el tiempo dependiendo del número de tickets que debamos procesar. Debemos estar seguros que el tiempo que pase entre la request al endpoint llamado en PARTE 2 y su response en la PARTE 4 sea inferior a los límites que el navegador imponga para recibir la mencionada response a la solicitud. Si ese tiempo superase estos límites, entonces el la elección de protocolo HTTP no nos serviría para este diseño. De darse el caso entonces deberíamos explorar otros protocolos de comunicación (Websockets, gRPC), que sí permiten conexiones abiertas más tiempo. En el momento de la entrega del proyecto puede que hayamos tomado alguno de esos protocolos alternativos -pero a ser posible se seguirá con protocolo HTTP por simplicidad-.*

3.5.3.4. PARTE 4: FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot y, después de persistir el cambio en los permisos, Spring Boot le manda de vuelta un nuevo token (con permisos a 1). FastAPI transmite al cliente los datos de los tickets

NOTA: *Lo que se describe en este apartado corresponde con las letras h, i, j k, l del diagrama de sistemas para el caso de registro (figura 2.2).*

Desde FastAPI, mientras todavía dura la solicitud HTTP a [/api/parsea-y-guarda-pdf-en-bbdd](#) que iniciamos en la PARTE 2, haremos ahora una llamada (como si fuese un cliente) hacia Spring Boot desde esta clase de servicio [serveiClient.py](#) usando el módulo `httpx`. Si el token saliente de FastAPI se dictamina válido en Spring Boot, entonces este último framework actualizará la variable de permisos en la tabla Usuaris de la bbdd mySql cambiándolo de 0 a 1. Acto seguido, Spring Boot nos mandará de vuelta un nuevo token de acceso “fresco” con los permisos actualizados a 1. Al recibirla en FastAPI se mandará la response al cliente (navegador) cerrando ya esa solicitud HTTP iniciada en la PARTE 2. El archivo del cliente que lo recibirá será la página `pas4_concedirAccesGmail.html`, cuyo JavaScript dirigirá al dash-

board en menos de un segundo y mostrará el dashboard de visualización al guardar el nuevo token.

3.5.4. Gestión de solicitudes: token de acceso

Con FastAPI no emitimos tokens de acceso: esto lo gestionamos con Spring Boot. Con FastAPI los recibimos del cliente y los procesamos⁴⁴: **si al hacerlo son válidos y no caducados, permitimos solicitudes entrantes.**

Por ejemplo, en dos endpoints de FastAPI (“/api/subir-tickets-pdf” y “/api/conta-pdfs-servidor”) llamados desde `pas4_concedirAccesGmail.html` en `controlador.py` podemos ver esta línea: “`payload_token: dict = Depends(verificar_token)`”

Esta línea lo que está haciendo es llamar a la función `verificar_token` en `jwtU-tils.py`. Este archivo Python, al que convenientemente le hemos puesto casi el mismo nombre que el que tenía la clase en Spring Boot (`JwtUtil.java`) donde guardábamos el secret con el que generábamos los tokens desde el otro back-end, no es casual: justamente aquí también guardamos una copia del mismo secret!

Ese “secret” es indispensable para validar si el token entrante en cualquiera de los endpoints que tenemos es válido o no (i.e. no se ha manipulado y podemos certificar su origen como token expedido desde Spring Boot). Es por ello que se ha puesto por duplicado tanto en el back-end de FastAPI (Python) como en el back-end de Spring Boot (Java).

Es importante mencionar que la gestión de tokens en Python (FastAPI) se hace notablemente más sencilla que con Java (Spring Boot). Por ejemplo, con FastAPI podemos conseguir que los endpoints no permitan solicitudes entrantes si el token ha caducado sin tener que gestionar la lógica de programación manualmente como sí teníamos que hacer con Spring Boot: en Spring Boot necesitábamos programar un total de 4 clases (dos clases para definir creación del token de acceso `AccessToken.java` y lectura `JwtUtil.java`; otra clase para manejar las excepciones de un token expirado y forzar que solo usuarios con un cierto id puedan acceder a contenidos: `FiltreAutenticacioJwt.java`; y finalmente una clase que defina un Bean que permita acceso a determinados endpoints a determinados usuarios según permisos, con la función `requestMatchers`: `ConfiguracioSeguretat.java`). Por el contrario, en FastAPI, con la librería de Python *JOSE* [15]⁴⁵ se hace todo muy rápido y solamente con las dos primeras funciones del archivo `jwtUtil.py` lo hemos podido solventar.

⁴⁴A excepción del caso especial de la PARTE 4!

⁴⁵Un nombre aparentemente muy español pero cuyo acrónimo significa *JavaScript Object Signing and Encryption (JOSE)*

3.5.5. Validación de datos (“/api/subir-tickets-pdf”)

Figura 3.20: Llamada al endpoint “/api/subir-tickets-pdf” | **Izq:** Solicitud POST con Postman: en verde, archivos que se suben correctamente; en rojo, archivos que servidor rechaza con las validaciones en FastAPI. | **Der:** Solicitud POST con el navegador | **Deabajo:** PDFs subidos al sistema de archivos del servidor con FastAPI.

The diagram illustrates the validation process for uploaded tickets using Postman and a browser, and their storage in a server's file system.

Left Side (Postman Request):

- Postman Headers:** POST localhost:8000/api/subir-tickets-pdf
- Postman Body (form-data):**
 - Key: arxius, Value: 20230925 Mercadona 1...
 - Key: arxius, Value: 20231004 Mercadona 2...
 - Key: arxius, Value: pdfMassaGran.pdf (highlighted with a green border)
 - Key: arxius, Value: pdfMassaPetit.pdf
 - Key: arxius, Value: un audio.mp3
 - Key: arxius, Value: 20250515 Carrefo...
- Postman Response:**

```

1 {
2   "subidos": 2,
3   "rechazados": 4,
4   "existentes": 2,
5   "estadosArchivos": [
6     {
7       "archivo": "20230925 Mercadona 13,71 €.pdf",
8       "estado": "Guardado correctamente",
9       "tamany": 35.6
10      },
11      {
12        "archivo": "20231004 Mercadona 27,40 €.pdf",
13        "estado": "Guardado correctamente",
14        "tamany": 35.7
15      },
16      {
17        "archivo": "pdfMassaGran.pdf",
18        "estado": "Archivo rechazado y no guardado (tamaño incorrecto!)",
19        "tamany": 120.8
20      },
21      {
22        "archivo": "pdfMassaPetit.pdf",
23        "estado": "Archivo rechazado y no guardado (tamaño incorrecto!)",
24        "tamany": 13.5
25      },
26      {
27        "archivo": "un audio.mp3",
28        "estado": "No guardado! No es un PDF"
29      },
30      {
31        "archivo": "20250515 Carrefour 26,16 €.pdf",
32        "estado": "Archivo rechazado y no guardado (NOMBRE incorrecto!)",
33        "tamany": 32.6
34      }
35   ]
}

```

Right Side (Browser Screenshot):

- File Explorer:** Shows a list of files:

20250507 Mercadona 7,05 €	33 KB	14/05/2025 23
20250513 Mercadona 0,74 €	33 KB	14/05/2025 23
20250515 Carrefour 26,16 €	33 KB	15/05/2025 17
pdfMassaGran	121 KB	21/03/2025 19
pdfMassaPetit	14 KB	18/05/2025 13
un audio	889 KB	06/04/2024 15
- Download Confirmation:** A modal window shows: “20250507 Mercadona 7,05 €” “20250513 Mercadona 0,74 €” “20250515 Carrefour 26,16 €” Cargar desde un dispositivo móvil Abrir Cancelar
- Icon:** An orange icon of a ticket with a paperclip.
- Text:**

Paso 2: Mándanos tus tickets

Selecciona todos los tickets digitales que se descargaron en la carpeta descargas y adjúntalos (no cambies su nombre!).

2 subidos | 4 rechazados total: 2 tickets facilitados

Bottom (File System):

- A terminal window shows the directory structure: APP > _pycache_ > tickets\3
- Files listed: 20230925 Mercadona 13,71 €.pdf and 20231004 Mercadona 27,40 €.pdf

Al subir PDFs al sistema de archivos del servidor hemos tenido mucho cuidado de no dejar pasar cualquier PDF o archivo, como vemos en la figura 3.20. La validación de los datos se ha hecho antes de guardar los PDFs en el sistema de archivos del servidor, desde el lado del servidor (no desde el cliente). Para hacer esta validación el orden es este: la solicitud al endpoint correspondiente (“/api/subir-tickets-pdf”)

ubicado en [controlador.py](#) llama a una función de [serveiTickets.py](#) (*guardaTicketsAsistemaDarxius()*) que a su vez invoca una función de [serveiValidacions.py](#) (*ticketValidat()*) que dirá si permite su guardado en el sistema de archivos del servidor o no.

Sabemos que será el usuario quien se descargue los tickets en pdf de su Gmail; pero luego, no podemos confiar que este sea responsable y suba archivos sin alterar al sistema de archivos de FastAPI. Por ende, se han verificado que se suban archivos en PDF (MIME type “application/pdf”) como tipo de datos entrante. Luego, se ha mirado que los tamaños de los archivos estén comprendidos entre unos límites razonables, dado que cada Ticket digital de Mercadona ocupaba 36KB en 2023, reduciéndose a 33KB en 2025, podemos establecer unos intervalos razonables a partir de esta información -un poco más amplios, pero no mucho más- que nos permitan acomodar variaciones futuras en el tamaño de los tickets que puedan hacer los desarrolladores de Mercadona (véase [este detalle](#) de líneas en [serveiTickets.py](#)) que a su vez nos protejan de uso malicioso de la API. En la función *guardaTicketsAsistemaDarxius()* de [serveiTickets.py](#) tenemos en cuenta que no se cargue el archivo en memoria del servidor (no solamente protegemos su almacenaje en disco duro), por si este es demasiado grande: así evitamos usos fraudulentos de la API. Si y solo si el archivo está entre los intervalos definidos, dejaremos que se lea en su totalidad y que luego se guarde de forma persistente en el sistema de archivos de FastAPI.

NOTA: Estos tickets a futuro probablemente se van a borrar del sistema de archivos una vez sus datos sean parseados y guardados en MongoDB. La razón de tenerlos físicamente guardados ya no obedecerá, entonces, a criterios operativos (consumen espacio, que al final es un recurso en un servidor por el cual pagaríamos: especialmente en arquitecturas serverless modernas -AWS, Azure, Google Cloud-, generalmente caras).

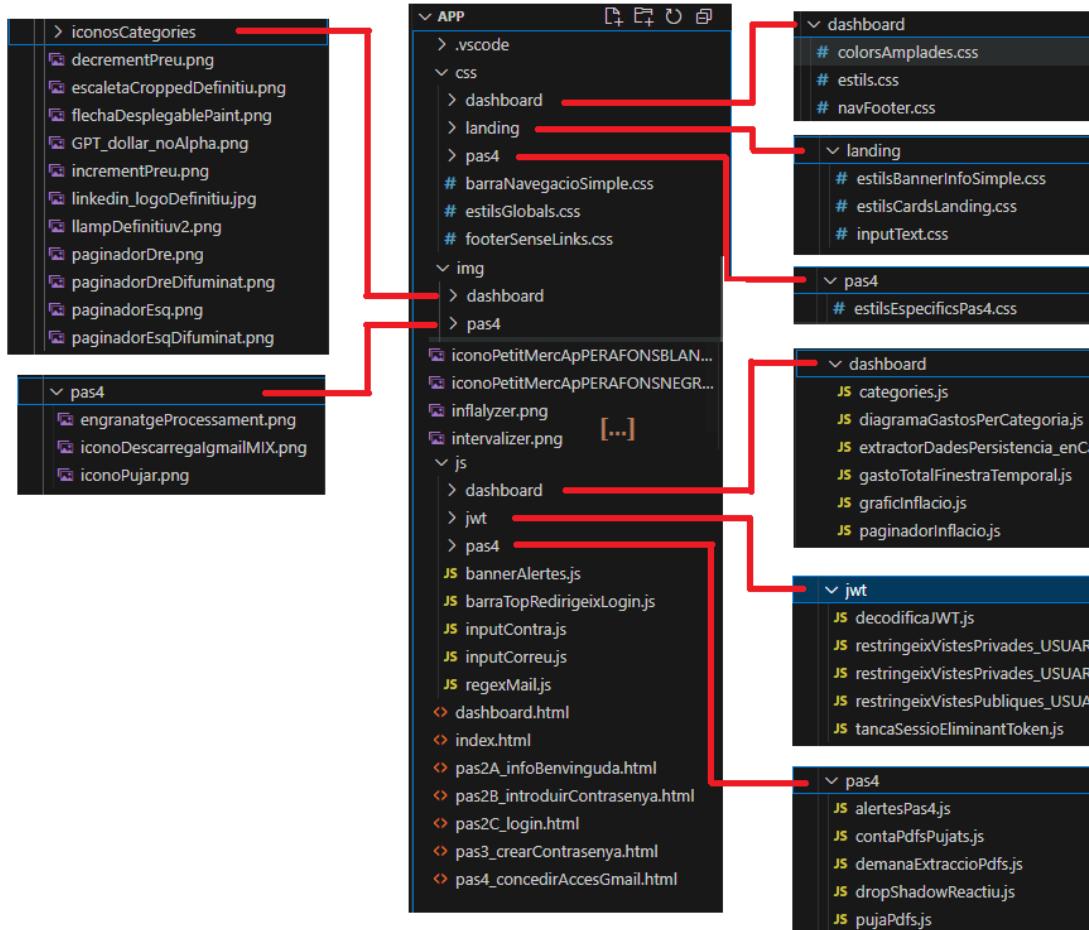
3.6. Desarrollo del front-end

3.6.1. Contenerización

Se ha utilizado Nginx, un servidor de alto rendimiento para servir los archivos estáticos (HTML, CSS y JavaScript). Podéis ver su dockerfile [aquí](#), y el script que crea la imagen e instancia de contenedor [aquí](#). Para más información del uso de Docker podéis ver el apartado [3.5.1](#) de la contenerización de Python.

3.6.2. Estructura de la aplicación

Figura 3.21: Estructura de carpetas y archivos de la aplicación front-end: punto de entrada a la aplicación es index.html. Hay que abrir el proyecto con live server desde la carpeta app o si corremos el contenedor Docker de Nginx moviéndonos al directorio donde está Dockerfile (moviéndonos a un nivel por encima de *app*) y derivar de él la imagen y el contenedor.



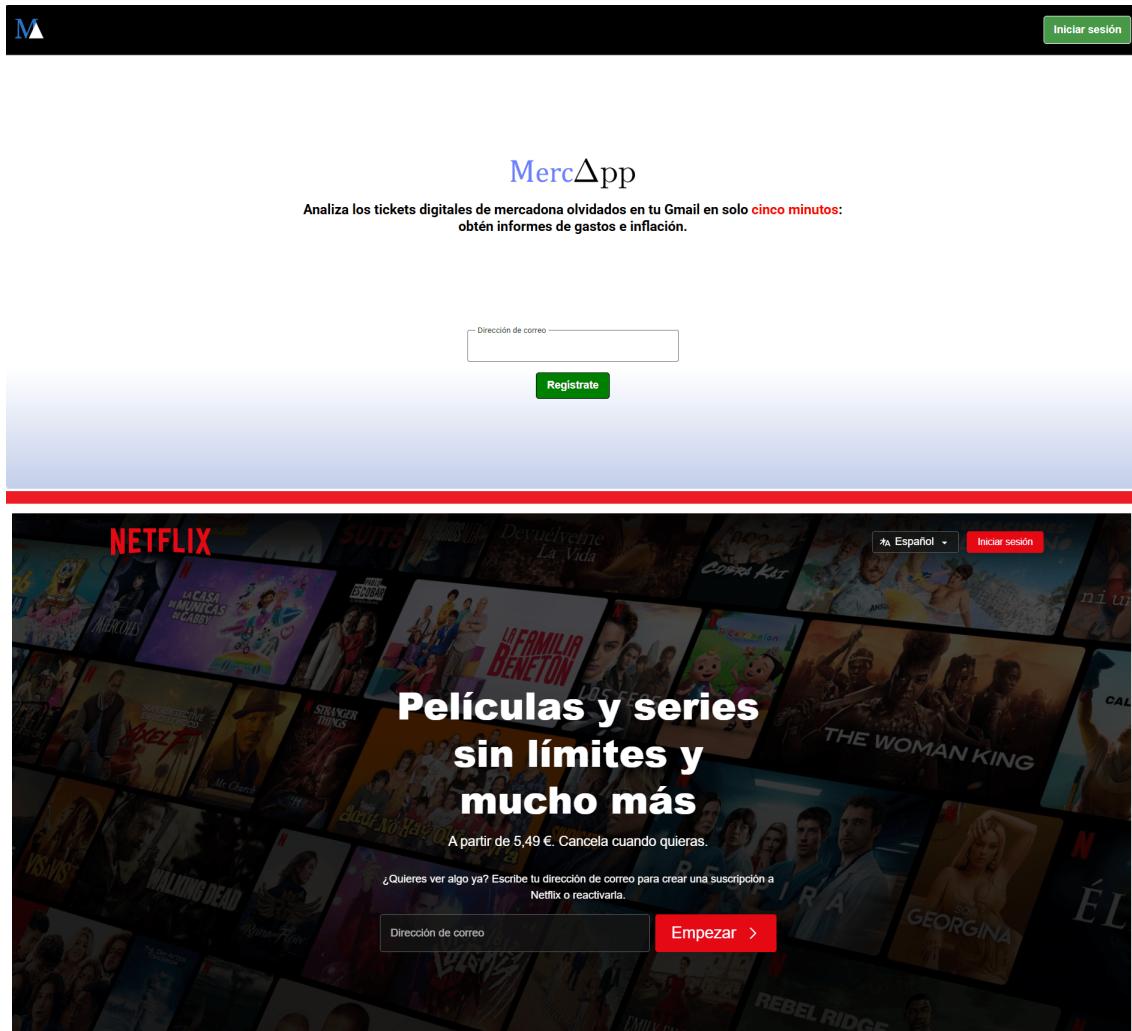
3.6.3. Enrutamiento de vistas

Cuando un usuario introduzca su correo en el formulario de registro de la página principal de la web (`pas1_LandingSignUp.html`, que renombraremos a `index.html`) va a ser redirigido con JavaScript a partir de las llamadas al back-end de Spring Boot: éste ultimo nos permitirá acceder al valor de la variable “permisos” de la tabla “usuaris” de mySql, siendo así redirigido a unas páginas u otras (el asunto de como se evita que ciertas páginas sean vistas por usuarios ya autorizados se cubre en otras secciones: apartado 3.6.4.2 y 3.6.4.3).

Estas redirecciones no son fruto del azar. Se ha hecho un proceso de desarrollo

inverso del proceso de registro de la plataforma NetFlix: replicándolo, desde cero, y adaptándolo a nuestro caso particular. Si Netflix utiliza ese esquema es porque tiene un impacto en la facilidad de captación de clientes y qué mejor que tratar de replicar los sistemas de los grandes *players* (podéis ver anexo 5.12 para más información).

Figura 3.22: **Imagen superior:** Detalle de la landing page `pas1_LandingSignUp.html` (`index.html`) donde el usuario introducirá inicialmente su correo para registrarse -aunque ese mismo formulario en realidad nos servirá para todo gracias al enrutamiento de vistas- || **Imagen inferior:** la página de Netflix en la que nos hemos inspirado para el diseño minimalista.



El esquema simplificado del proceso de enrutamiento durante el registro de un usuario en NetFlix queda recogido en el diagrama del anexo (ver apartado 5.5) y puede consultarse también en uno de los repositorios de mi github ([link](#)).

Asimismo, el proceso de registro que utilizamos en mercApp es convenientemente una derivación de este mismo: si bien en NetFlix primeramente se redirige al usuario a unas cartas de pago, nosotros aquí le llevamos a una página para que nos dé acceso al Gmail en el que Mercadona envía los tickets digitales al usuario (la página

`pas4_ConcedirAccesGmail.html`); de nuevo, análogamente a la solución de NetFlix, donde al usuario que ya ha pagado se le concede inmediatamente el acceso a las películas y series, en nuestro caso al usuario se le dará acceso al usuario al tablón de visualización de análisis de datos de los tickets digitales (`dashboard.html`), donde se visualizan el resultado de la minería y extracción de datos de esos tickets.

El proceso de enrutamiento de los usuarios desde que introducen el correo en el formulario de `pas1_LandingSignUp.html` (`index.html`) hasta que acceden al `dashboard` se encuentra recogido en el diagrama de la figura 3.23. Para entenderla ver la pregunta siguiente:

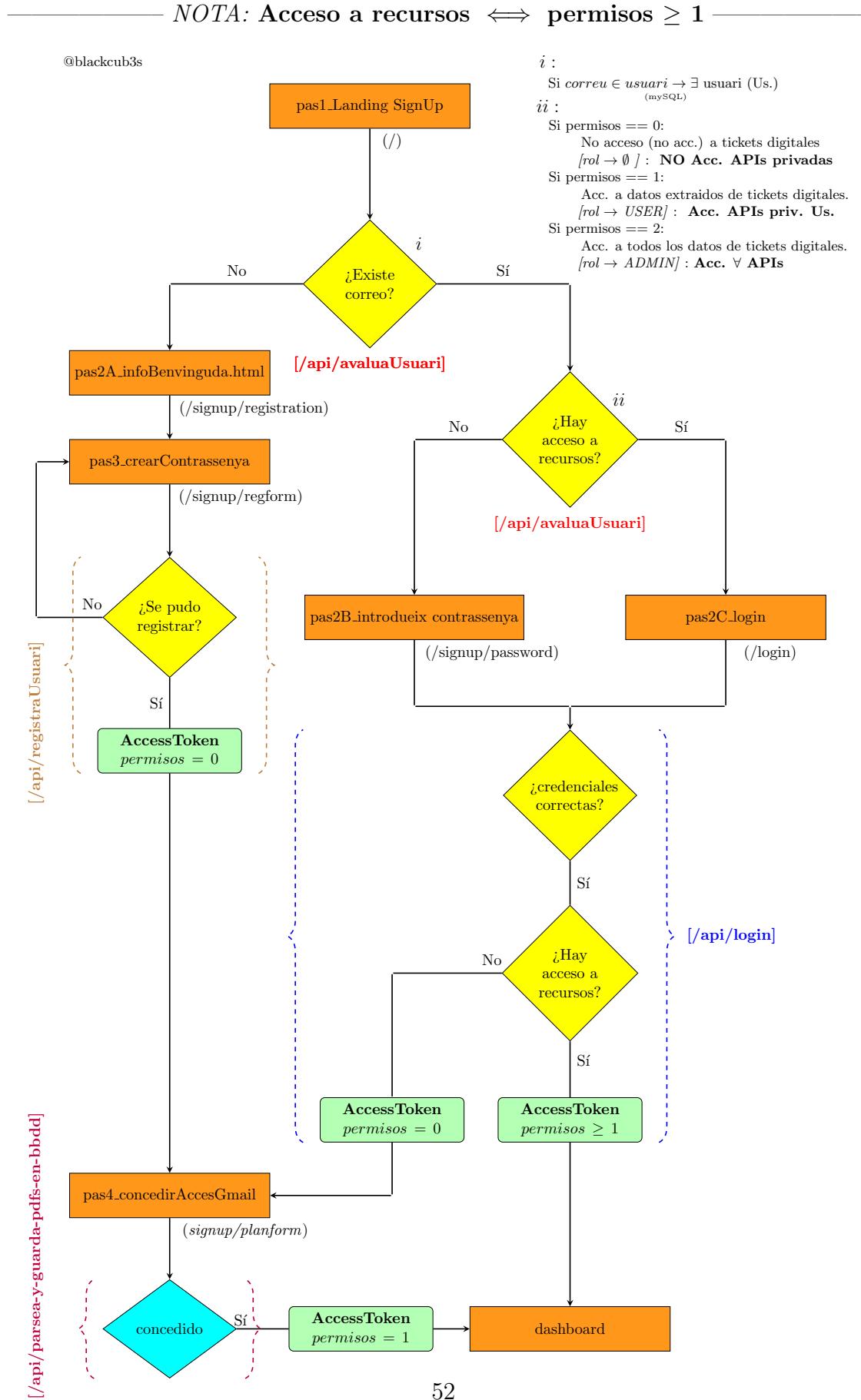
¿Qué significan los distintos elementos visuales del diagrama de 3.23?

- **rectángulos naranja**. → Representan las vistas -archivos html- a las que redirige JavaScript mediante la llamada a `window.location.href` a partir de los resultados de las llamadas a endpoint de APIs.
- **rombos amarillos** → Representación de decisiones del back-end de **Spring Boot** hechas como respuesta a peticiones de front-end a sus endpoints.
- **rombo azul cian** → Una decisión del back-end de **FastAPI** en respuesta a petición front-end a un endpoint de su API.
- **rectángulos verdes** → Representan expedición de tokens de acceso (JWT) desde el endpoint del que emanan y su enviado a la vista a la que apuntan ⁴⁶.
- Los endpoints de API que consume el front-end para hacer el enrutamiento se muestran entre corchetes (`[]`) y con colores. Concretamente tenemos:
 - **[/api/avaluaUsuari]**: *end-point* de Spring Boot que evalúa si el correo electrónico introducido pertenece a un usuario registrado y sus permisos.
 - **[/api/registraUsuari]**: *end-point* de Spring Boot que registra un nuevo usuario en el sistema y expide su AccessToken con permisos a 0 en las *claims* de su *payload*⁴⁷.
 - **[/api/login]**: *end-point* de Spring Boot que gestiona el proceso de autenticación y generación del *JWT Access Token* con tres niveles de permisos posibles (0, 1 y 2).
 - **[/api/parsea-y-guarda-pdf-s-en-bbdd]**: *end-point* de FastAPI que traslada al front-end AccessToken expedido por Spring B. con permisos a 1.
- Finalmente, debajo de los rectángulos naranja (vistas), hay un paréntesis del estilo `(/dir1/dir2)`, sin color y pequeño. Estos paréntesis hacen referencia a las URLs de las páginas o vistas de Netflix cuyo comportamiento replicamos en cierta medida mediante las vistas de nuestro proyecto (rectángulos naranja). Muy importante: véase anexo 5.12.

⁴⁶Consultad la estructura de los mismos en la figura 3.7.

⁴⁷El lector puede consultar la explicación sobre lo que son las claims y el payload de un JWT en el apartado 3.4.3.2.

Figura 3.23: Diagrama de flujo del enrutamiento completo del sistema *front-end* durante el registro de un usuario, desde que este introduce su correo en `pas1_LandingSignUp.html` (`index.html`) hasta obtener acceso al `dashboard`.



3.6.4. Manejar vistas en función de Autenticación y autorización

Como hemos visto antes Podemos considerar que cada archivo HTML y su CSS asociado es una “vista” de nuestra aplicación. Habrá vistas que **no nos interesará enseñar a ciertos usuarios**, porque o bien no serán relevantes para ellos o bien harán llamadas a APIs cuya información no podrá ser obtenida para ellos.

Si bien Spring Boot permite servir los archivos estáticos⁴⁸ de dentro del mismo back-end y utilizar un sistema de plantillas (Thymeleaf) que permite impedir visualizaciones de vistas a usuarios no autorizados, esto realmente no es, para nada, lo ideal. Lo ideal es definir un front-end y un back-end separados partiendo de principios de *separación de responsabilidades* o *SoC*⁴⁹, y así lo hemos hecho en este proyecto⁵⁰. Las ventajas son grandes y tienen implicaciones en términos de mantenimiento, escalabilidad y reutilización tanto del front-end como del back-end (ver ventajas justificadas en anexo 5.7).

Sin embargo, no todo es ideal. Siempre existen concesiones (o como diríamos en inglés “trade-offs”). Al tener el front-end y el back-end desacoplados esto también aumenta considerablemente la complejidad inicial en el desarrollo: la protección de las vistas a usuarios que no deben visualizarlas se hace más difícil porque no las sirve el back-end y no las puede proteger directamente este⁵¹.

Por ejemplo, del mismo modo que los endpoints de nuestra API del back-end en Spring Boot o del back-end de FastAPI están protegidos por token y no devuelven datos cuando el JWT de acceso que tengamos en el front-end haya caducado, sea inexistente, o sea inválido (porque haya sido manipulado o no tenga en el *payload* el “idUsuari” que permita el acceso a un cierto recurso), también pasará que ciertas páginas del front-end no podrán obtener la información deseada si llaman a un endpoint para el que no tienen autorización: en este caso ello tendrá implicaciones para las vistas, y deberemos modificar su DOM para la ocasión mostrando un mensaje de error, instando al usuario a iniciar sesión y/o bien redirigir al usuario a la página correcta, por ejemplo. Nosotros hemos optado por este último enfoque.

⁴⁸HTML, CSS y JS son archivos estáticos.

⁴⁹Separation of concerns

⁵⁰Si tenemos ambas partes desacopladas podremos hacer modificaciones independientes en ambas. Por ejemplo, podremos cargar los archivos front-end en una CDN o un Proxy o tenerlos cacheados en un servidor que los sirva mucho más rápido, como Nginx. Es más, lo óptimo sería generar los archivos del front-end mediante un sistema de desarrollo por componentes (como Angular, React o Vue) para facilitar el desarrollo cuando la aplicación crezca y utilizar una paradigma *SPA* (*Single Page Application*). Sin embargo, en este caso, por el tiempo disponible y el tamaño de la aplicación se ha optado por hacerlo con HTML, CSS y JS puros.

⁵¹A diferencia de lo que sí haría una aplicación back-end hecha en php tradicional como las que hemos visto en desarrollo web entorno servidor, donde servimos el HTML desde dentro del mismo PHP).

Con tal de conseguirlo, deberemos manejar la lógica en cada caso particular desde el front-end usando JavaScript. Tengo entendido que en frameworks como Angular esto se puede hacer de forma muy sencilla, solo definiéndolo en una ocasión. Aquí cada página particular requerirá una programación específica con JavaScript para redirigir automáticamente a los usuarios, con 3 casos de uso identificados: de [3.6.4.1](#) hasta [3.6.4.3](#):

3.6.4.1. Protegiendo vistas “privadas” ante usuarios no “logueados”: cuando el token ha expirado

Existen dos páginas de nuestra web que, cuando expire el token de acceso que se requiere para acceder a los recursos back-end que hay detrás de ellas (o este no exista), no deberán ser visualizables:

- `pas4_concedirAccesGmail.html`
- `dashboard.html`

Como se desprende del diagrama de flujo del enrutamiento del proceso de registro que vimos en la figura [3.23](#), esas dos páginas son aquellas páginas de nuestro proyecto a las que redirigimos los usuarios justo después de generar tokens de acceso; por ende, su visionado requiere cierto grado de autenticación y autorización. De ahí que consideremos no permitir visualizarlas si el grado de permisos requerido no se llegase a satisfacer.

La primera página (`dashboard.html`), requiere tener un token con permisos a 0; la segunda (`pas4_concedirAccesGmail.html`) un token con permisos superior o igual a 1. En otras palabras: ambas requieren tener algún tipo de autenticación de usuario que se materialice en un token de acceso con una variable de permisos (i.e a esto nos referimos con usuarios “logueados” en el título), como veremos en el siguiente apartado; pero está claro que para visualizar cualquiera de estas dos páginas o vistas, la condición *sine qua non* comuna es que dentro de `localStorage.getItem("AccessToken")` se albergue un token de acceso que no esté expirado y que sea descifrable⁵².

Si está expirado, cuando hagamos la diferencia entre el valor “exp” del payload⁵³ y la función `Date.now()/1000`⁵⁴ saldrá un número negativo. En caso contrario, positivo.

Si el token está expirado -o es inexistente- inmediatamente redirigiremos al usuario a la landing page llamando a `redirigeixALandingPage()`, impidiendo así el visio-

⁵²Ojo, describirable no significa validable. Validable es lo que hacemos en el back-end con el secret, que no se pasa jamás al cliente.

⁵³Segundos en que el token expira o expiró, desde la epoch.

⁵⁴Segundos actuales del navegador, desde la epoch.

nado de cualquiera de las dos páginas (se cargará el script que contiene esta función antes de que se cargue el DOM⁵⁵). En cambio, si el token no está expirado seguiremos revaluando la expiración del token -y su existencia- con una frecuencia de un segundo: esto se conseguirá mediante la función asíncrona `setInterval()` que hemos visto en desarrollo web entorno cliente de segundo curso.

Para ello, en el *head* de cada una de las dos páginas en cuestión veréis que **antes** siquiera **de cargar el DOM** se cargan sendos archivos:

```
<script src="/js/jwt/decodificaJWT.js"></script>
<script src="/js/jwt/restringeixVistesPrivades_USUARI_NO_LOGUEJAT.js">
</script>
```

En el primero tenemos la función para extraer el payload de un token. Y en el segundo está la lógica que acabamos de explicar (ver figura 3.24):

Figura 3.24: Script `restringeixVistesPrivades_USUARI_NO_LOGUEJAT.js`, utilizando para regresar automáticamente a la landing page cuando el token de acceso de un usuario logueado expira -o es borrado- o cuando un usuario no logueado intenta acceder a las páginas que requieren permisos de acceso: `pas4_concedirAccesGmail.html` y `dashboard.html`.

```
function zeroPadding(segons) {
    if (segons <= 9) {
        return `0${segons}`;
    }
    return `${segons}`;
}

function redirigeix_a_landing() {
    localStorage.removeItem("AccessToken"); //borrem el token
    window.location.href = "pas1_landingSignUp.html";
}

function mostra_temps_restant(secFinsExpiracio) {
    console.clear();
    console.log(`Queden ${Math.floor(secFinsExpiracio/60)}:${zeroPadding(Math.round(secFinsExpiracio%60))}s\nfins a l'expiració`);
}

//REQUEREIX CARREGAR PRIMER script decodificaJWT.js!!
//exemple de payload --> { permisos: 0, idUsuari: 3, sub: 'noacces@gmail.com', iat: 1744214393, exp: 1744215293}
function redirigeixALandingPage() {
    try {
        const payload = decodificaPayloadJWT(localStorage.getItem("AccessToken"));
        let secFinsExpiracio = payload.exp - (Date.now()/1000); //comparam els dos temps fins la epoch
        let tokenHaExpirat = secFinsExpiracio < 0;

        if (tokenHaExpirat) {
            redirigeix_a_landing();
        } else {
            mostra_temps_restant(secFinsExpiracio);
        }
    } catch (error) {
        redirigeix_a_landing(); //si salta excepció és que no hi ha token o s'ha manipulat.
    }
}

redirigeixALandingPage(); //així la primera crida a la funció no espera 1 segon
setInterval(redirigeixALandingPage, 1000); //repetim crides subsequentes cada segon
```

⁵⁵El lector puede hacer la prueba siguiente: si el token ha caducado o el usuario no se ha logueado, accediendo a `pas4_concedirAccesGmail.htm` o a `dashboard.html` verá como automáticamente se produce una redirección a `pas1_landingSignUp.html` (`index.html`); o si se ha logueado, abrir la consola y verá una cuenta atrás del tiempo que le queda al token de acceso para su expiración y para la redirección a la landing page.

3.6.4.2. Protegiendo vistas “públicas” para usuarios “logueados”: cuando el token no ha expirado

Para empezar, es necesario mencionar que se establecen tres niveles de permisos en la aplicación: estos tienen un impacto sobre qué puede visualizar el usuario “logueado” y qué no; y cómo se permite que el usuario navegue a medida que va moviéndose en el proceso de registro cuando no está “logueado”.

Antes ya hemos visto que estos permisos nos han permitido construir el enruteamiento del front-end de la aplicación (su impacto podemos verlo en la parte superior derecha de la figura del enruteamiento 3.23 y más resumidamente en el cuadro 3.1), pero también ahora debemos utilizarlos también para **impedir** visualizar ciertas páginas en usuarios ya logueados, es decir, aquellas páginas a las que nos referimos con el término “vistas públicas” empleado en el título de esta sección.

Cuadro 3.1: Significado de la variable `permisos` en la tabla `usuariis` de mySQL.

permisos	Significado
0	No hay acceso a tickets digitales (pero ya tenemos email y contraseña)
1	Hay acceso a tickets [guardados en MongoDB] como usuario (USER)
2	Hay acceso a tickets [guardados en MongoDB] como admin (ADMIN)

Programáticamente, debemos conseguir que estas “vistas públicas” **no sean visualizables** jamás si, en el *local storage*, existe un token de acceso que **no haya expirado**⁵⁶. Estas páginas vetadas a los usuarios “logueados” son las siguientes:

- A) pas1_landingSignUp.html (index.html)
- B) pas2A_infoBenvinguda.html
- C) pas2B_introduirContrasenya.html
- D) pas2C_login.html
- E) pas3_crearContrasenya.html

La forma que optamos para impedir su visualización es poner **dos scripts** en cada una de las cinco vistas públicas arriba mencionadas, que permitirán redirigir al usuario a la página “privada” que le corresponda según el valor que toma la variable “permisos”⁵⁷ dentro del *payload* del token de acceso guardado en el *local storage*⁵⁸, según se muestra en la tabla siguiente:

En cada una de las páginas accedidas A), B), C), D) y E), antes que cargue el DOM, los dos scripts mencionados a cargar son:

⁵⁶Si existe ese token no expirado significa que el usuario ya no debe acceder a ellas, porque ya se ha registrado y/o iniciado sesión y solo debe ver páginas de usuario registrado!

⁵⁷No hace falta mencionar que se saca del campo permisos de la tabla usuariis de la base de datos que tenemos en mySQL.

⁵⁸Si queréis ver a qué me refiero con el payload, revisad de nuevo la figura 3.7.

página accedida	Permisos token	Redirigimos automáticamente a
A), B, C), D) o E)	0 1 2	pas4_concedirAccesGmail.html dashboard.html

```
<script src="/js/jwt/decodificaJWT.js"></script>
<script src="/js/jwt/restringeixVistesPubliques_USUARI_LOGUEJAT.js"></script>
```

El segundo script, `restringeixVistesPubliques_USUARI_loguejat.js` es una modificación para la ocasión del script mostrado en la figura 3.24 previa, y lo mostramos a continuación en la figura 3.25:

Figura 3.25: Script `restringeixVistesPubliques_USUARI_LOGUEJAT.js`, utilizado para redirigir a un usuario logueado fuera de las páginas públicas de forma automática y directo a las 2 páginas posibles que requieren credencial de acceso (“privadas”), según proceda de acuerdo con la variable permisos de su token de acceso -si el token existe y no ha expirado: `pas4_concedirAccesGmail.html` y `dashboard.html`.

```
3 function redirigeixApaginaProtegida() {
4     try {
5         const payload = decodificaPayloadJWT(localStorage.getItem("AccessToken"));
6         let secFinsExpiracio = payload.exp - (Date.now() / 1000); //comparam els dos temps fins la epoch
7         let tokenHaExpirat = secFinsExpiracio < 0;
8
9         if (!tokenHaExpirat) {
10             if (payload.permisos >=1)
11                 window.location.href = "dashboard.html";
12             else if (payload.permisos == 0)
13                 window.location.href = "pas4_concedirAccesGmail.html";
14         }
15     } catch (error) {
16         console.log("NO HI HA TOKEN :). Per tant, no cal fer redirecció i deixem l'usuari en la pàgina pública :)");
17     }
18 }
19
20 redirigeixApaginaProtegida(); //així la primera crida a la funció no espera 1 segon
21 setInterval(redirigeixApaginaProtegida, 1000); //repetim crides subsegüents cada segon (si queda oberta una pàgina en una altra pestanya ens redirigeix)
```

3.6.4.3. Protegiendo vistas “privadas” ante usuarios “logueados”: cuando el token no ha expirado.

Análogamente al apartado anterior, tenemos que tomar en consideración las páginas que requieren permisos de acceso. Estas páginas *ya están* protegidas del visión de usuarios no logueados (usuarios que no tengan token expedido): estos no las podrán ver nunca, porque hicimos el script de la figura 3.24 para redirigirlos a la landing page en caso que por error accedan a ellas.

Sin embargo, nos queda algo por hacer: hay que conseguir **evitar** que un usuario con permisos 1 o 2 en el payload de su token (es decir, que ya le hemos extraído y persistido los datos de los tickets) pueda pedir de nuevo volver a proporcionarnos tickets; algo completamente innecesario porque ya los ha facilitado a mercApp previamente en `pas4_concedirAccesGmail`; o, su opuesto: tenemos que evitar que un usuario con permisos a 0 (e.g., ha puesto correo y contraseña y se ha registrado, puede que incluso haya subido los tickets al sistema de archivos del servidor, pero **no ha pedido persistir** esos tickets a formato estructurado) pueda tratar de visualizar

el **dashboard** a la espera de obtener una información de unos tickets que todavía no está accesible en nuestro sistema⁵⁹

La solución a lo anterior es hacer que en función de los permisos existentes en el token inhabilitemos una vista de las “privadas” para el usuario, mediante la redirección automática del usuario a la página que sí debe visualizar *antes* de que cargue el DOM de la pagina vetada, tal que así:

p. privada accedida (vetada)	Permisos	Redirección automática a
dashboard.html	0	pas4_concedirAccesGmail.html
pas4_concedirAccesGmail.html	1 2	dashboard.html

Es decir, en la tabla anterior mostramos que si un usuario con permiso 0 (no se ha persistido todavía, para él, información de tickets en MongoDB) quiere acceder a la página **dashboard.html** para visualizar la explotación de datos de los tickets, no podrá verla porque le redirigiremos automáticamente a la página donde podrá proporcionar acceso a datos de tickets digitales: **pas4_concedirAccesGmail.html**. Y viceversa: Si entra en *pas4* teniendo permisos 1 o 2, será redirigido al *dashboard*.

Lo que acabamos de mencionar en la última tabla y en el párrafo anterior lo programamos en el siguiente script (cuyo prerequisito será decodificaJWT como en las anteriores ocasiones) y que podemos ver en la imagen 3.26:

```
<script src="/js/jwt/decodificaJWT.js"></script>
<script src="/js/jwt/restringeixVistesPrivades_USUARI_NO_LOGUEJAT.js">
</script>
```

⁵⁹Sí puede darse el caso que él haya subido los tickets -o parte de los tickets- al sistema de archivos del servidor, pero si todavía no ha dado su consentimiento para su extracción el sistema seguirá considerándolo un usuario “invitado” (con permisos a 0) y no un usuario con permisos a 1 (user). El token con permisos a 1 se expide solo cuando el usuario manda los tikets a mongoDB, persistidos; no cuando los manda al sistema de archivos del servidor.

Figura 3.26: Script `restringeixVistesPrivades_USUARI_LOGUEJAT.js`, utilizado para redirigir a un usuario logueado hacia `pas4_concedirAccesGmail.html` o `dashboard.html`, según proceda, en caso que el usuario entre en una de estas dos páginas privadas sin el permiso correspondiente.

```
function redirigeix_a_dashboard_o_pas4() {
    try {
        const payload = decodificaPayloadJWT(localStorage.getItem("AccessToken"));
        let secFinsExpiracio = payload.exp - (Date.now()/1000); //comparam els dos temps fins la epoch
        let tokenHaExpirat = secFinsExpiracio < 0;

        if (!tokenHaExpirat) {
            let paginaActual = window.location.pathname.split("/")[1];
            if (payload.permisos >= 1) {
                if (paginaActual == "pas4_concedirAccesGmail.html") {
                    window.location.href = "dashboard.html";
                }
            } else if (payload.permisos == 0) {
                if (paginaActual == "dashboard.html") {
                    window.location.href = "pas4_concedirAccesGmail.html";
                }
            }
        }
    } catch (error) {
        console.clear();
        console.log("NO HI HA TOKEN :)");
    }
}

redirigeix_a_dashboard_o_pas4();
```

Para entender como guardamos los datos de permisos e ids de usuario en el front-end, podemos ver el apartado [3.6.5](#) que viene a continuación, donde especificamos como se recibe el token del back-end y se guarda en el front-end.

3.6.4.4. Salir voluntariamente de las páginas privadas: botón “cerrar sesión”

El botón de “cerrar sesión” dentro de las dos páginas privadas `dashboard.html` y `pas4_concedirAccesGmail.html` en realidad no cierra ninguna sesión. Recorremos que usamos token de acceso, que sustituye las sesiones. En el botón, sin embargo, hemos decidido mantener el nombre, porque el cierre de sesión es un concepto arraigado en los usuarios de sitios web, incluso más que el concepto de “salir”⁶⁰.

Lo que hace es, simplemente, eliminar el token de acceso del `localStorage` cuando el usuario clica el botón de id “`botoEliminarToken`”.

En cada una de estas páginas privadas recordemos que tenemos un script que cada segundo está reevaluando si hay token de acceso y si este es válido (ver script

⁶⁰A nivel de usabilidad se me hace difícil justificar que un usuario diese click a un botón denominado “eliminar token” solamente porque el desarrollador quería ser preciso; dejemos esto como una prueba de como en ocasiones la usabilidad viene de la sencillez, no del honor a la verdad.

de figura 3.24). Por lo tanto, si lo borramos ese script va a redirigirnos a la página de inicio como máximo un segundo más tarde de la pulsación del botón de “cierre de sesión”.

Figura 3.27: Script `tancaSessioEliminantToken` que elimina el token de acceso cuando el usuario clica en el botón “cerrar sesión” en las páginas privadas `dashboard.html` y `pas4_concedirAccesGmail.html`

```
document.addEventListener("DOMContentLoaded", () => {
    const botoTancarSessio = document.getElementById("botoEliminarToken");
    botoTancarSessio.addEventListener("click", () => {
        localStorage.removeItem("AccessToken");
    });
});
```

3.6.5. Recibir el Access Token desde el back-end

Cuando un usuario del que ya tenemos su correo electrónico en BBDD intenta “loguearse” en mercApp⁶¹, el token de acceso lo recibe por primera vez en el cliente cuando este haga una llamada `fetch()` hacia el endpoint del back-end “`/api/-login`”. Esta llamada se hace desde `pas2C_login.html` o desde `pas2B_introduixContrassenya.html`, de idéntica forma.

Por ejemplo, explicaremos solamente el caso de `pas2Clogin.html`. En este archivo la recepción del token se hará en el JavaScript embedido cuando, por un lado, obtengamos el código 200 (OK) del servidor; pero también, cuando se cumpla que el usuario y contraseña introducidos por el usuario son correctos. Si y solo si se cumplen ambas condiciones, el cliente entonces recibirá el token de acceso en el body de la respuesta a su solicitud, que será una como la que sigue, de la que podremos extraer el “AccessToken” y guardarlo inmediatamente en el `localStorage` del navegador: ⁶² Para más información sobre el código JavaScript del front-end que lo permite véase figuras y 3.28 y 3.29):

```
{
    "usuari": {
        "aliases": "the protein kingdom",
        "permisos": 2,
        "idUsuari": 1
    },
    "existeixUsuari": true,
```

⁶¹Sea que ya estuvo registrado -permisos 0-, dio acceso a sus tickets digitales -permisos 1- o es superusuario -permisos 2-.

⁶²Esto lo hacemos para luego poder mandarlo de vuelta al servidor en la subsecuentes solicitudes que requieran autenticación y autorización.

```

        "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJ [...]" ,
        "teAccesArecursos": true ,
        "contrasenyaCorrecta": true
    }
}

```

Figura 3.28: Fragmento de código en `pas2C_login.html` dentro del código JavaScript para manejar códigos de error. Cuando el back-end de Spring Boot devuelva el código 200 significará que podremos extraer los datos del body de la respuesta. 400 se devolvería si hubiera problemas validación de campos en el back-end, algo que no debería producirse nunca con el front-end que se ha programado.

```

body: JSON.stringify({ email: email, contra: contra }), |
}) // ----- PAS 2: AVALUO SI LA REPUESTA ES EXITOSA (NO HA DONAT CODIS D'ERROR).
.then(response => {
    // verificar si la respuesta es exitosa.
    if (response.status == 200)
        return response.json(); // Convertir la respuesta a JSON
    else if (response.status == 400)
        return response.json().then(data => Promise.reject({ type: 'validacion', data }));
    else {
        throw new Error('Error! La respuesta de xarxa no ha sido exitosa!');
    }
}

}) // ----- PAS 3: MANEJO LA RESPUESTA JSON -----
.then(dadesExitosesJSON => {

```

Figura 3.29: Fragmento de código JavaScript en `pas2C_login.html` que guarda en localStorage el token de acceso (detalle en rojo), recién obtenido del servidor.

```

}) // ----- PAS 3: MANEJO LA RESPUESTA JSON -----
.then(dadesExitosesJSON => {

    console.log(dadesExitosesJSON); // Imprimir la respuesta en la consola (treure a producció: d
    if (dadesExitosesJSON.existeixUsuari) {
        if (dadesExitosesJSON.contrasenyaCorrecta) {
            let tokenAccesJWT = dadesExitosesJSON.AccessToken; //extrec el token
            localStorage.setItem("AccessToken", tokenAccesJWT); //guardem token al localStorage

            if (dadesExitosesJSON.teAccesArecursos) {
                bannerAlerta([], "bienvenidoAlaApp", "var(--verdAlerta)");
                setTimeout(() => {window.location.href = "/dashboard.html";}, tEspera); //ENVIO USUARI
            } else { //NO TE ACCES A RECURSOS (USUARI TE CONTA I CONTRASEÑA)
                bannerAlerta([email], "usuariExisteixPeroNoTieneRecursos", "var(--lilaAlerta)");
                setTimeout(() => { //ENVIO USUARI A OBTENER RECURSOS
                    window.location.href = "/pas4_concedirAccesGmail.html";
                }, tEspera*3); //PASO DE 1 SEGON A 3 SEGONS
            }
        }
    }
}

```

Ahora bien, si el usuario nunca se ha registrado en nuestra aplicación⁶³, cuando lo haga, lo hará por el proceso de registro y no por el proceso de inicio de sesión. Una vez introduzca su contraseña en `pas3_crearContrasenya.html` y se tome el correo

⁶³Es decir, no tenemos su correo electrónico en BBDD.

electrónico insertado por el usuario en páginas previas y guardado en el localStorage, se hará una llamada POST con `fetch()` al endpoint “`api/registraUsuari`” pasando esos datos por el body: si y solo si el usuario NO existía, se creará un nuevo registro en la tabla Usuaris y ahí se devolverá un JSON con el token de acceso para el nuevo usuario creado, ahora de permisos 0. Este JSON tendrá el siguiente aspecto (el token será mucho más largo):

```
{
    "existiaUsuari": false,
    "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJw [...]",
    "usuariShaRegistrat": true
}
```

Para entender de dónde viene el token desde el back-end redirigimos al lector a la sección [3.4.3.4](#), donde se trata ese aspecto. En la presente sección nos ocuparemos de JavaScript en el front. Por ahora el lector debe tener claro que, como hemos visto ya, existen tres páginas HTML con códigos Javascript embedidos que pueden hacer llamadas asíncronas y obtener un token de acceso del servidor y guardarla en el localStorage (un detalle de los formularios de estas páginas se encuentra en la figura [3.30](#)):

Figura 3.30: Detalle de los formularios que permiten generar llamadas a los dos endpoints generadores de tokens de acceso (`/api/login` y `/api/registraUsuari`). De izquierda a derecha las páginas que los contienen: `pas2C_login.html`, `pas3_crearContrasenia.html` y `pas2B_introducirContrasenia.html`

The figure displays three separate web forms side-by-side:

- Form 1 (Left): Iniciar sesión**
A login form with fields for 'Dirección de correo' containing 'superacces@gmail.com' and 'Contraseña' containing '*****'. It features a green 'Inicia Sesión' button and a blue '¿Has olvidado tu contraseña?' link.
- Form 2 (Middle): Crea tu contraseña!**
A password creation form with a note '¡Ya casi hemos terminado!' above a 'Contraseña' field. It has a green 'Siguiente' button.
- Form 3 (Right): Te damos de nuevo la bienvenida!**
A welcome back form with the text 'Escribe tu contraseña para acceder a tu usuario.' It includes a 'Correo electrónico:' field, an 'Escribe tu contraseña' field, and a green 'Siguiente' button.

NOTA: En este trabajo no implementaremos cookies ya que implica configuración extra tanto en el cliente como en el servidor. Vamos a guardar el token en el cliente en el localStorage (que es, de hecho, una práctica habitual en aplicaciones que no requieren un alto grado de seguridad). También hay que mencionar sobre que existe un debate para ver si en ese logIn el token de acceso recién generado en el servidor se debe mandar al cliente en el body de la respuesta de la solicitud POST o bien en la header “Authorization”. Sin embargo, es práctica común mandarlo en el body. Nótese, que para el paso inverso (cliente a servidor) sí debe mandarse en el Heather “Authorization” con el preámbulo “Bearer ” seguido del token.

3.6.6. Validación de datos (Formularios entrada)

NOTA: Los datos validados en el front-end siguen las mismas expresiones regulares y restricciones que las validaciones hechas en el back-end (ver sección 3.4.4) y establecen redundancia completa.

Vamos a poner el mismo caso concreto que pusimos en la sección homóloga de validaciones en el back-end de Spring Boot mencionado en el párrafo de la nota anterior. Asumamos también que un usuario quiere loguearse, pero no usando *Postman* y llamando directamente al back-end sin una interfaz, sino a través del formulario que hay en nuestro front-end para el inicio de sesión.

Para ello entra en la página `pas2C_login` e introduce sus datos en el formulario que esta contiene (sin más complicación podéis ver el aspecto de este formulario en la figura anterior 3.30: es el que está a la izquierda).

Con las validaciones aplicadas en el front se le impedirá a ese usuario mandar la solicitud POST (con `fetch`) al servidor si los datos que ha introducido no cumplen unos mínimos. Esto obedece a dos ventajas evidentes: la primera, es que el usuario obtiene feedback inmediato, porque es el cliente -navegador- el que gestiona los errores y no el servidor; la segunda, es que evitamos sobrecargar el servidor con procesos que se pueden gestionar perfectamente con el front.

En este caso estas validaciones deben ser **exactas** a las validaciones del back-end, y solamente son un complemento y no las sustituyen, como ya hemos dicho en el apartado homólogo dedicado al back-end: deben tener correspondencia completa con las validaciones del back-end porque si no están bien hechas en el front-end podría darse el caso que un usuario intentase poner unos datos que luego el back-end no admitiese o viceversa; además, no las sustituyen porque justamente un usuario puede modificar fácilmente el front-end, quitar las protecciones del javascript (bastaría quitar con el *inspector* del navegador las llamadas a funciones de validación en dos líneas del javascript ([éstas](#)) y entonces hacer llamadas directas a la API del back-end de Spring-Boot con la función `fetch` ya disponible para el usuario ([detalle líneas](#)). Entonces, el endpoint `/api/login` al que llamamos con la función `fetch` estaría desprotegido. A continuación mostramos como validamos el correo y la contraseña en el front:

3.6.6.1. Validación del correo

Cuando el usuario se loguea, es la función `correuApte(email)` del archivo [js/regexMail.js](#) la que valida la entrada de datos con una expresión regular. Podéis ver ese mismo archivo en la figura 3.31.

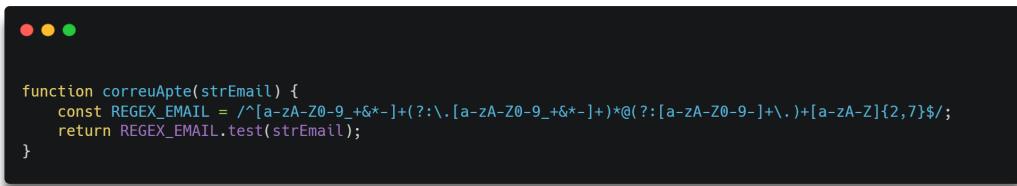
Para hablar de lo que hace esa expresión regular, vale la pena mencionar primero

las partes de las que se compone un correo electrónico:

```
parte local@dominio . subdominio
```

Con esa expresión regular se han permitido mayúsculas, minúsculas y ciertos caracteres como el +, el ampersand o el guión medio antes de la arroba (parte local). Después de la arroba de un correo introducido, se restringe de forma distinta el dominio y el subdominio: por ejemplo, el dominio no tiene restricción de longitud, pero el subdominio sí (debe tener más de 2 y menos de 7 caracteres). Nótese que esta misma expresión regular evita los caracteres peligrosos también (que igualmente protegería nuestro back-end): las “”, los < y >.

Figura 3.31: Mediante una sola expresión regular facilitada por un LLM hemos podido hacer una correspondencia completa con las validaciones del back-end.



```
function correuApte(strEmail) {
  const REGEX_EMAIL = /^[a-zA-Z0-9_+&*-]+(?:\.[a-zA-Z0-9_+&*-]+)*@(?:[a-zA-Z0-9-]+\.)+[a-zA-Z]{2,7}$/;
  return REGEX_EMAIL.test(strEmail);
}
```

Por ello, cuando un usuario ponga su correo en el formulario correspondiente de la vista de inicio de sesión en `pas2C_login.html`, al *darle a enviar* se va a activar en última instancia la función `correuApte()` y saltará una alerta que impedirá que pueda mandar el correo si la expresión regular no se cumple. La alerta la llamaremos dentro del script embedido en el html de la propia página, y va a llamar a las funciones que generan banners de alerta que usaremos solamente en las páginas públicas (ubicadas en `js/bannerAlertes.js`) tal como podemos ver en la figura 3.32.

Asimismo, no en la página de login, pero sí en la página del formulario de registro (`index.html` o `pas1`) usamos esta misma función `correuApte(email)` para generar información dinámica de los errores al usuario **antes** de que trate de enviar nada con el botón de registro. Esto lo hacemos mediante el uso de los eventos “focus”, “input” y “blur” ampliamente vistos en la asignatura desarrollo web entorno Cliente de segundo de DAW (véase función `esdevenimentsCorreu_SIGN_UP()` dentro del archivo de GitHub [/js/inputCorreu.js](#)) para poder ver cómo se han conseguido los efectos en el contorno del formulario de registro para el correo del `index.html`. Esto también es visible en la figura 3.33, en donde explicamos el impacto que estos 3 eventos de JavaScript tienen: principalmente, que ayudan a mostrar prevalidaciones al usuario de una forma intuitiva. Los colores del contorno y los mensajes van en línea con la forma en que Google y Netflix definen feedback a los usuarios en sus

formularios.

Figura 3.32: Activación del banner de alertas en el formulario de inicio de sesión de la vista pas2C_login.html ante un input de correo incorrecto después de clicar en “iniciar sesión”. ¡En este caso no se manda nada al servidor!

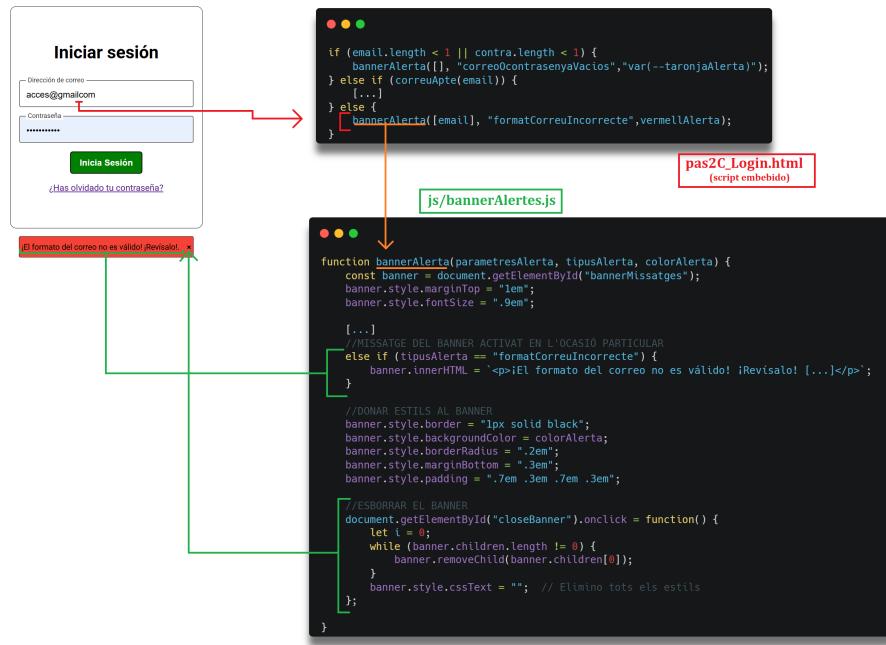
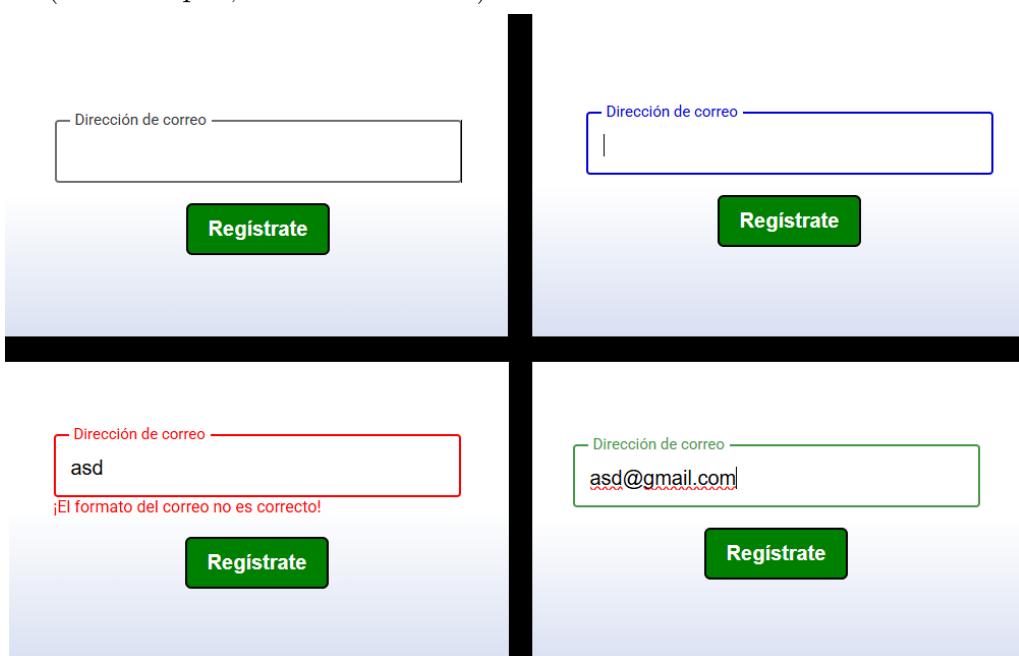


Figura 3.33: Prevalidaciones en tiempo real en pas1_Landing_SignUp o index.html.
 | **SupIzq**: estado natural (contorno gris)| **SupDer**: click encima del formulario sin datos (cambio ante evento “focus”, contorno azul)| **InflIzq**: datos introducidos incorrectos (cambio ante evento “blur”) | **InfDer**: contorno muestra éxito con un correo válido (evento input, contorno a verde).



3.6.6.2. Validación de la contraseña

La validación de la contraseña tanto en la página general de inicio de sesión `pas2C_login.html` como en la página pensada para aquellos usuarios que ya se registraron en nuestro sistema pero todavía no llegaron a obtener acceso al dashboard `pas2B_introducirContrasenya.html` siempre la hacemos **después** de que el usuario haga click en el botón de iniciar sesión: si la contraseña no cumple los mínimos no se mandará nada al servidor. Estas validaciones (o post-validaciones, a lo mejor, sería mejor término) las hacemos sin usar eventos que se activen antes del evento click (a excepción del uso de “focus” y “blur”, pero no para validar, sino para cambiar el contorno a azul cuando se hace click en el formulario: [ver detalle](#) de líneas de código).

Estos mínimos de los que hablamos permiten verificar que el usuario esté entrando una contraseña que luego las validaciones del back-end también admitan. Estas validaciones las hemos generado verificando expresiones regulares con la función `regex.test(contraseña)` vista también en desarrollo web entorno cliente, que nos devuelve un booleano que indica si la `regex` hace *match* con la contraseña pasada por parámetro. Así las cosas, tomando diversas funciones `.test()` aplicadas al binomio “contraseña introducida - expresión regular”, podemos hacer la intersección de todas ellas mediante la operación lógica AND, como podéis ver en la figura 3.34, testeando así si la contraseña introducida es válida o no. Podéis ver también en GitHub el archivo completo del que extraemos el fragmento de la figura: [/js/inputContra.js](#)

Figura 3.34: Subconjunto de código de `js/inputContra.js` responsable de las validaciones de contraseña. Hemos querido apostar por obligar al usuario a generar una contraseña segura con mínimo una mayúscula, una minúscula, un número y sin espacios: forzando que no se puedan usar caracteres distintos a los grupos anteriores exceptuando el punto y la barra baja. También permitimos acentos y cedilla (ç).



```
let regex_nomesLletres_i_nombres_barraBaixa_punts = /^[a-zA-Z0-9àéèòóú_ÀÈÒÓÙÑÇ]+$/;
let regex_min_obligat = /[a-z][ç]/;
let regex_maj_obligat = /[A-Z][ç]/;
let regex_nre_obligat = /[0-9]/;
let regex_noEspais_obligat = /^[^\s]+$/;

//PRE: la contraseña que ha introduit l'usuari (no serà mai buida).
//POST: un booleà que mostra si la contrasenya és apta segons
//      els estàndards que hem definit a les regex o no.
function contraseñaApta(contraseña) {
    console.log("contraseña: ", contraseña);
    return regex_nomesLletres_i_nombres_barraBaixa_punts.test(contraseña) &&
        regex_min_obligat.test(contraseña) &&
        regex_maj_obligat.test(contraseña) &&
        regex_nre_obligat.test(contraseña) &&
        regex_noEspais_obligat.test(contraseña);
}
```

Un ejemplo de puesta en escena de las expresiones regulares de la figura anterior la podemos ver en la figura 3.35 siguiente:

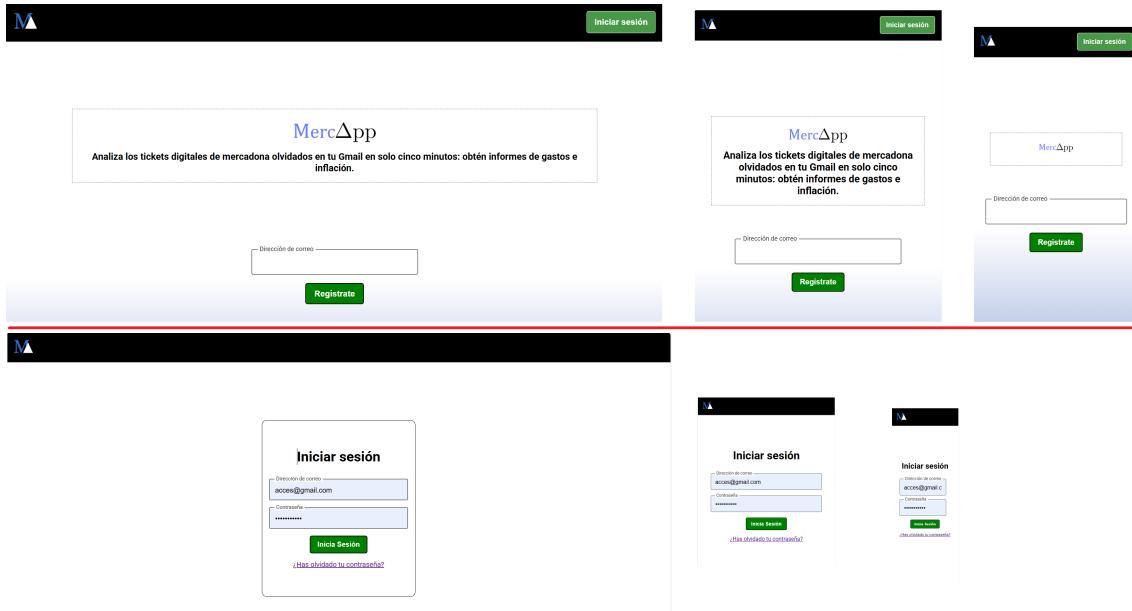
Figura 3.35: Hasta que la contraseña no reúne unos mínimos no se permite mandar nada desde el formulario de inicio de sesión hacia el servidor. Se muestra una relación de los archivos (scripts simplificados) y del recorrido de una contraseña erróneamente escogida e introducida en el formulario, que hacen posible mostrar el array de errores en pantalla vinculados con el formato incorrecto de la contraseña. Así el usuario tiene feedback y puede fácilmente ajustar la contraseña para que sea válida, dado que cada error emana de una expresión regular de las mostradas en la figura anterior. Nótese que seguimos sin hacer ninguna llamada al servidor.



3.6.7. Diseño (UX/UI): páginas públicas

El diseño de las páginas públicas (es decir, los archivos html que no son ni el dashboard ni el pas4_) también son responsive. Tienen por ahora un diseño minimalista para no distraer al usuario y facilitarle su registro. A continuación se puede ver cómo las distintas *media queries* afectan a dos de las páginas públicas:

Figura 3.36: Arriba vemos la parte superior de index.html. Debajo la página de inicio de sesión. De izquierda a derecha: vista de escritorio, tablet y móvil gracias a media queries en el CSS.



Las media queries de las páginas públicas se han separado en distintos archivos. Por ejemplo, la barra de navegación de `index.html`, `pas2A_infoBenvinguda`, `pas2B_introduirContrasenya` y `pas2C_login` tienen sus estilos y su media query aplicadas todos en el archivo siguiente: [css/barraNavegacioSimple.css](#). Dado que no hemos utilizado un framework de javascript que nos permita desarrollar en componentes hemos visto pertinente tratar de modularizar en la medida de lo posible aquello que hemos desarrollado. Con lo cual todas estas páginas con una navbar incluyen este archivo en el `head`.

Nótese que los formularios para introducir correo y contraseña tienen el mismo aspecto tanto en `index.html` como en `pas2C_login.html` y el botón de registro e inicio de sesión también son iguales. También hemos modularizado su CSS introduciéndolo en un mismo archivo, en la ruta: [css/landing/inputText.css](#)

Finalmente tenemos la footer del `index.html`: solo la hemos introducido por ahora en `pas4_concedirAccesGmail.html`(una de las páginas privadas), pero no en todas las demás páginas públicas: en algunas de estas últimas queremos solamente que los usuarios pongan contraseña o correo y no queremos distraer con detalles superfluos. De ahí que la barra de navegación de páginas públicas no tenga links en la navbar. Los links los encontramos una vez el usuario gana acceso al `dashboard`. Sin embargo, a futuro se podría expandir la página del `index.html` para incluir una barra de navegación ligera con información de quienes somos y una footer en todas las páginas públicas.

Figura 3.37: Creación de la footer en `index.html` y en `pas4_concedirAccesGmail.html` con diseño minimalista: link a linkedin y `display: flex` para centrar verticalmente y horizontalmente el mensaje en el espacio ocupado por la footer



3.6.8. Diseño (UX/UI): páginas privadas

3.6.8.1. Diseño del dashboard

Parte del diseño del dashboard es un reaprovechamiento del figma y del código HTML y CSS creado para el proyecto final de la asignatura de desarrollo de interfaces, concretamente, de una página donde se explicaban diferencias entre frameworks de front-end que el lector puede consultar [aquí](#) para ver la semilla del diseño original.

Esa página fue diseñada y programada por mí de forma íntegra, desde cero con CSS y HTML (a excepción de la Footer que la hizo mi compañero de grupo, dado que era un proyecto grupal). De la parte del footer no se ha aprovechado HTML ni CSS porque el código de la misma no fue de mi autoría, así que se ha optado por un diseño distinto.

El motivo del reaprovechamiento del código es que era imposible asegurar una firma visual distintiva y un aspecto profesional con un diseño desde cero en el poco tiempo para hacer el proyecto. Para hacer la página original, fueron dedicadas muchas, *muchísimas* horas por mi parte: una barbaridad de hecho, así que me sabía mal no reutilizarlo. Además, lo bueno de la reutilización es que se ha podido mejorar la página inicial con dos aspectos que en la asignatura de interfaces NO se pudieron cubrir por los requisitos de la misma:

- **Diseño responsive:** la página original `frontEnd.html` del proyecto de interfaces no podía ser responsive mientras que `dashboard` sí lo es⁶⁴.
- **Persistencia de datos:** las vistas de la página del dashboard permiten visualizar datos extraídos de una base de datos de mongoDB.

⁶⁴en la medida de lo posible, dado que no todas las librerías admiten responsividad: por ejemplo, `chart.js` no lo es.

Después de esta aclaración vamos a pasar a explicar las secciones del dashboard y de su diseño.

SECCIÓN 0 (S0): barra de navegación y botón de cierre de sesión

Esta sección incorpora el botón de cierre de sesión (que elimina el token de acceso, en realidad, como ya se ha visto en la sección 3.6.4.4) y la barra de navegación que conseguirá redirigir a ubicaciones distintas en función de si se es usuario (permisos=1) o superusuario⁶⁵ (permisos=2). También muestra links a páginas que nos permiten ver los tickets del usuario (si los tiene descargados), sus datos en una tabla y una página para contactar con nosotros (ver figuras 3.38, 3.39 y 3.40).

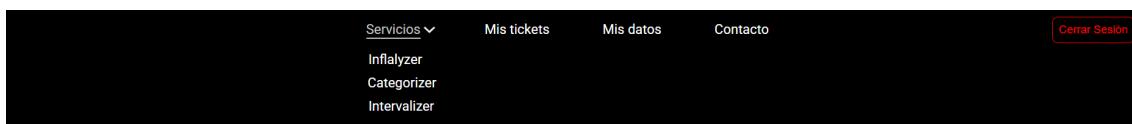
Además se ha utilizado un sistema (figura 3.41)para conseguir que al hacer hover en cada link cambie su color a gris y aparezca una línea por debajo con una transición suave (automatizando las propiedades *border-bottom* y *color*). Se ha tenido especial cuidado en que el añadir la línea debajo NO desplace el resto de la página hacia abajo. Para evitarlo todos los links tienen en realidad una línea por debajo del mismo color del fondo, que solo cambia de color al pasar por encima. Hay muchas líneas de código para conseguirlo y redirigimos al lector al GitHub para verlas (nótese el uso de selectores descendientes, por norma: [link a navFooter.css](#))

Figura 3.38: Barra de navegación del usuario de permisos=1 con sus 4 elementos



Dentro de la sección servicios hay un desplegable o “dropdown” que nos permite llegar a las distintas secciones del dashboard: el “inflalyzer”, el “categorizer” y el “intervalizer”.

Figura 3.39: Detalle barra de navegación con el drop-down desplegado.



⁶⁵También lo llamamos ADMIN.

Figura 3.40: Detalle del código que hizo posible el drop-down del menú. Cada elemento drop down se ha animado mediante animate.css definiendo distintas duraciones de la animación *fadeInDown* para conseguir un efecto persiana. El uso del selector descendiente y el no abuso de los IDs a menos que sea necesario es una metodología seguida a lo largo del diseño del dashboard.

```

/* SECCIÓN 0 (S0) ... NIVEL 0; Barra de navegación... */
header>
<button id = "botonEliminarToken">Cerrar Sesión</button>
<nav>

    <ul>
        <li>
            <span>Servicios</span> 
            <ul>
                <li><a href="#PRIV_evolucionInflacionPorProducto">Inflalyzer</a></li>
                <li><a href="#Categorizer">Categorizer</a></li>
                <li><a href="#">Intervalizer</a></li>
            </ul>
        </li>
        <li><span><a href="#">Mis tickets</a></span></li>
        <li><span><a href="#">Mis datos</a></span></li>
        <li><span><a href="#">Contacto</a></span></li>
    </ul>
</nav>

```

The diagram shows a red arrow pointing from the original navigation code on the left to a screenshot of the dashboard's header on the right. The screenshot shows a dropdown menu for 'Servicios' with three items: 'Inflalyzer', 'Categorizer', and 'Intervalizer'. Another red arrow points from the screenshot to the final CSS code on the right.

```

/*desplegable "Inflalyzer"*/
nav ul li ul li:first-child {
    position: relative;
    left: -4em; /*[A]*/
    animation: fadeInDown; /*[B]*/
    animation-duration: .8s; /*[C]*/
}

/*desplegable "Categorizer"*/
nav ul li ul li:nth-child(2) {
    position: relative;
    left: 1em; /*[A]*/
    animation: fadeInDown; /*[B]*/
    animation-duration: 1.6s; /*[C]*/
}

/*desplegable "Intervalizer"*/
nav ul li ul li:last-child {
    position: relative;
    left: .05em; /*[A]*/
    animation: fadeInDown; /*[B]*/
    animation-duration: 2.4s; /*[C]*/
}

/*Espaciado entre los elementos del desplegable*/
nav ul li ul li li {margin-top: .5em; }


```

Figura 3.41: Transiciones suaves de propiedades *border-bottom* y *color* al hacer hover en los spans que envuelven los links tanto en la barra de navegación como en la footer.



SECCIÓN 1 (S1): presentación Dashboard

Esta sección simplemente muestra un gradiente lineal que transiciona del negro de la barra de navegación hacia el azul claro que es colindante a la barra roja. Para hacerlo se ha hecho mediante un doble gradiente lineal. También se ha usado la clase fadeIn de animate.css para hacer que aparezca con un fundido de entrada al cargar o recargar la página. Podéis ver la figura que muestra el código en 3.42.

Figura 3.42: De izquierda a derecha: código HTML, código CSS y el resultado final de la vista de esa sección



SECCIÓN 2 (S2): características de la aplicación mercApp:

Esta sección contiene tres “cards” que resumen los servicios que ofrece el resto del dashboard (inflación por producto, gastos por categoría y gastos por ventana temporal): también incorporan los datos más relevantes de los tickets de cada usuario, que se extraen también de la BBDD (ver detalle figura 3.43). El lector puede consultar en el fichero [estilos.css](#) todos los selectores css descendientes que empiezan por el id con el que encabezamos el section de esa parte (id *PRIV_caracteristicasDashboard*).

De estas cards hay tres aspectos a destacar: En *primer lugar*, se ha vigilado en que sigan exactamente la proporción áurea. Al crearlas se ha definido la propiedad CSS *aspect-ratio* ([ver en GitHub](#) para que la relación de aspecto ancho-alto de cada una de ellas sea exactamente de 1 a 1.618 [16]. De este modo, creamos un rectángulo agradable a la vista para poder presentar los primeros datos del resumen de los tickets al usuario. En *segundo lugar*, en cada una de las cards se ha usado un gradiente lineal vertical mediante la propiedad CSS *background* ([ver línea](#)) que hemos utilizado también para cambiar dinámicamente ambos colores del gradiente al hacer hover ([ver línea](#)). En esos casos, además el hover altera también el *box-shadow* definido en esta [línea](#) y lo modifica a otro valor ([en esta](#)). El resultado de esta programación es el que se puede ver en la figura 3.44. También se han definido automatizaciones de la propiedad *transform* para las imágenes de dentro de las cards, para hacer que aumenten de tamaño al posarnos con el ratón por encima ([detalle líneas](#)).

Finalmente, merece la pena mencionar que mediante *wow.js* se ha conseguido crear las animaciones de *animate.css* (como las que se usaron en la sección 0, ver captura previa 3.40) de modo que en lugar de que aparezcan con un temporizador desde la carga de la página, que aparezcan con un temporizador desde que hacemos scroll a la sección que las incluye.

Así las cosas, las propiedades que nos define *animate.css* en este caso no las hemos escrito en el CSS sino que lo hemos hecho en el propio HTML de acuerdo con el requerimiento de la librería *wow.js*. De este modo al hacer scroll a la sección, la card de la izquierda aparecerá por la izquierda (*fadeInLeft*, [link](#)), la del medio vendrá

de de abajo (fadeInUp, [link](#)) y la de la derecha de esa misma dirección (fadeInRight, [link](#)). Una vez hecho esto se ajustó la duración y el *delay* de cada propiedad mediante atributos en esas mismas líneas del HTML.

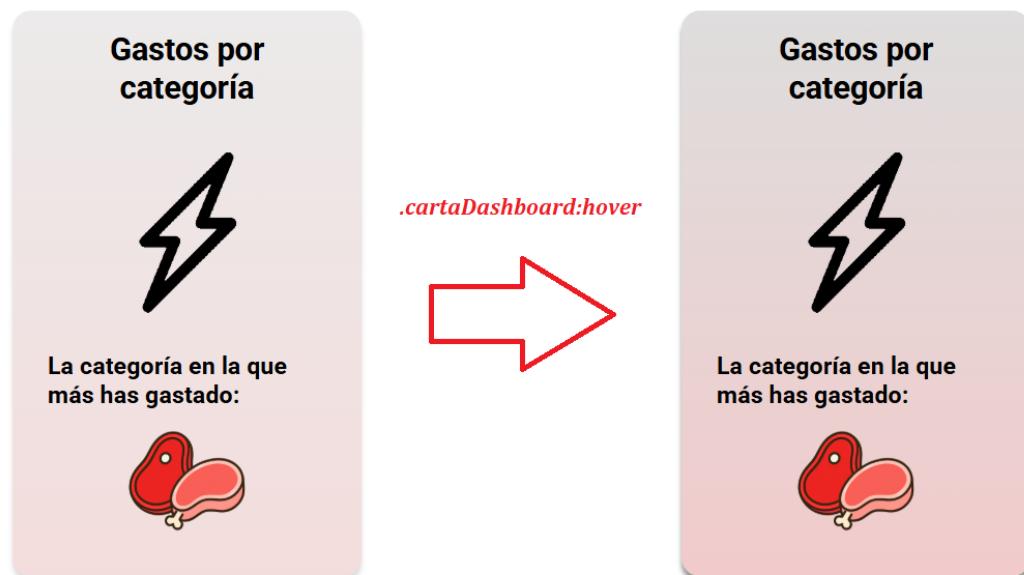
Figura 3.43: Detalle de la sección con las tres cards: características resumidas del dashboard. Los textos en color son placeholders para lo que se extraerá de la BBDD.

¡Hola Nombre de usuario!

MercApp ha encontrado **XXX** tickets digitales de mercadona en tu correo electrónico, desde **dd/mm/aa** hasta **DD/MM/AA**:
Resumen de todas estas compras a continuación:



Figura 3.44: Detalle del impacto que tiene la automatización de *box-shadow* y *background* al hacer hover encima de las cards.



SECCIÓN 3 (S3): “inflalyzer” (sin swiper: ¡esta vez hecho a mano!)

En la página original de la que hemos derivado parte del dashboard ([link](#)) utilizamos la librería *Swiper* para cambiar entre imágenes de una galería mediante clicks en unos paginadores (si no podéis ver la web, podéis ver detalle en figura 5.1 del anexo). Tratar de reutilizarlo para nuestro caso fue imposible: ahora no tenemos solo imágenes a ir variando con los paginadores, sino que tenemos una tabla entera a poner dentro de esos paginadores. Swiper esto no lo podía manejar.

Así las cosas hubo que crear un swiper manualmente para poder albergar una tabla dentro del paginador (ver figura 3.45). Este “swiper” artesanal, se creó tomando los mismos iconos de swiper, eliminando el canal alpha del fondo, cambiando su color de azul a rojo y definiendo eventos de click a las imágenes que conforman esas flechitas.

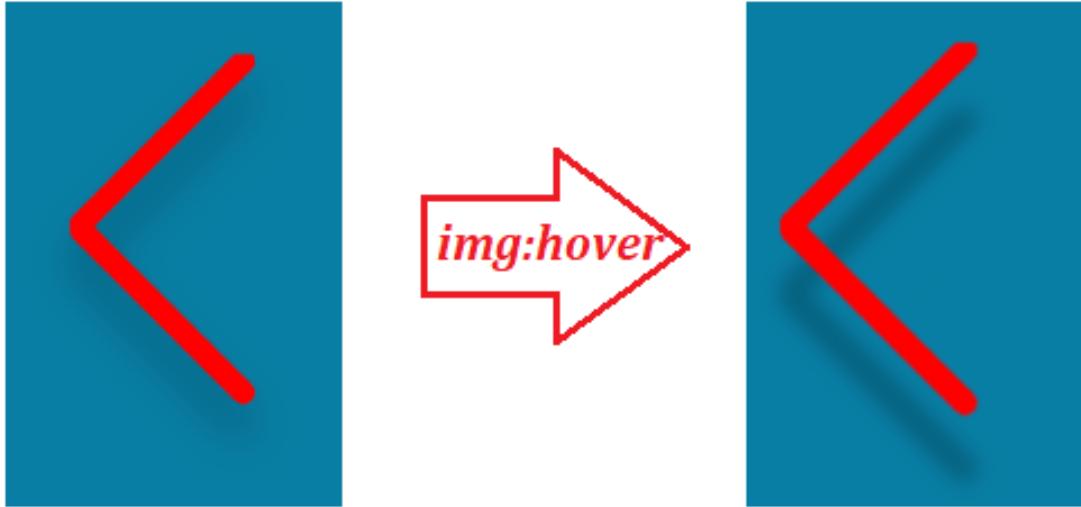
Figura 3.45: Detalle de la parte superior del intervalizer: obsérvese el paginador creado programáticamente.

Inflalyzer: evolución de los precios por producto (“inflación”)



A nivel de UX/UI el paginador de la figura 3.45 ha seguido los mismos principios. En este caso, hay un par de cosas destacables. El uso de imágenes customizadas para los botones de paginación nos ha habilitado para usar *hover* afectando la propiedad *drop-shadow* de la imagen del paginador, para así aprovechar los contornos del mismo hacer transform y dar la sensación de profundidad **combinando la sombra con el incremento de tamaño** (ver imagen 3.46).

Figura 3.46: Paginadores para moverse hacia productos con más frecuencia de compra: a la izquierda sin hover (drop-shadow difuso y poco visible) y con hover a la derecha de la captura (drop-shadow con menos difusión de imagen y ligeramente desplazado hacia abajo).



Asimismo, cuando llegamos al final del listado de productos del usuario, sea por la izquierda o la derecha, se cargan unas imágenes difuminadas que además con JavaScript no permitimos que traten de acceder a ningún dato ([ver detalle](#)).

El CSS de la imagen 3.45 está disponible en este rango de líneas de GitHub [link](#). Para observar el JavaScript de la paginación se puede visualizar el apartado 3.6.9.1.

SECCIÓN 4 (S4): “categoryizer” y SECCIÓN 5 (S5): “intervalizer”

Estas dos secciones todavía no tienen diseño y, por lo tanto, todavía no tienen CSS asociado. Cuando esté hecho el CSS aparecerá dentro del mismo archivo donde se han programado las tres secciones anteriores, con los debidos encabezados explicativos para distinguirlas de las demás: [/css/dashboard/estils.css](#)

3.6.8.2. Diseño del “pas4_concedirAccesGmail.html”

El diseño del *pas4_concedirAccesGmail.html* no entraña mucha más dificultad que el diseño del **dashboard**. Se ha seguido un procedimiento parecido para la generación de iconos: los iconos generados se hicieron con IA y se eliminó el fondo con Gimp, requiriendo alguna edición extra en caso de ser necesaria: podéis verlos en la carpeta de [edición de iconos](#).

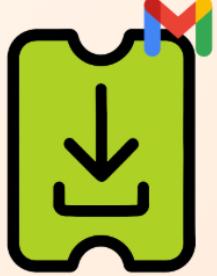
El diseño del botón CSS para cerrar sesión es el mismo que en el **dashboard**. En cambio, se han eliminado todos los links de la navBar. Luego se ha definido para la versión desktop un grid con dos columnas, alternando imágenes y texto, tal que así:

Figura 3.47: Vista del grid de dos columnas del pas4_concedirAccesGmail.html en la versión desktop. En los círculos a color saldrán mensajes informativos que mostrarán mensajes al usuario sobre el número de tickets subidos (Paso 2, rojo) y el éxito o no de la extracción de datos (Paso 3, verde)

M Cerrar Sesión

Paso 1: Descarga tus tickets digitales

Habilita ventanas emergentes y luego autentícate en tu gmail en una cuenta con más de dos tickets digitales dentro de días distintos. Descarga hasta los 500 tickets más recientes automáticamente.



Paso 2: Mándanos tus tickets digitales

Selecciona todos los tickets digitales que se han descargaron en la carpeta descargas de tu sistema en el paso anterior y adjúntalos (¡no cambies su nombre!).



Paso 3: Inicia la extracción de datos.

Una vez estés conforme con todos los tickets digitales subidos en el Paso anterior, clica en el ícono del engranaje: entonces nuestra aplicación (extraerá) el contenido de los tickets digitales en PDF y analizará sus datos. Al terminar serás redirigido a tu *dashboard* de visualización de datos.



Un aspecto diferencial con el **dashboard** es que el diseño responsive ha implicado generar una media query en la que le pedíamos en los selectores descendientes que invirtieran el orden en el que aparecían el contenedor de las imágenes en relación al contenedor del texto (h1 y h2): nótese que en el HTML el ícono descarga de tickets y el ícono del engranaje aparecen después del contenedor del texto porque queremos hacer que aparezcan a la derecha: así alternan los iconos y el texto en la vista de escritorio en un grid de dos columnas.

Sin embargo, este diseño de la versión de escritorio tiene un inconveniente: que al ponerlo en vertical en un grid de una sola columna para la versión mobile tanto el ícono de “descarga tickets” como el del “engranaje” aparezcan en orden inverso (debajo del texto y no encima). La solución a esta problemática está en hacer que la primera fila del grid en esos casos, que es el texto, ocupe la segunda posición en la columna vertical mediante la propiedad *grid-row: 2* como podemos ver en la imagen [3.48](#) (mismo código en contexto, marcado línea a línea [aquí](#)). En esta misma imagen también podemos ver la adición de una línea separadora en la versión mobile.

Figura 3.48: Media query para diseño responsive del grid de dos columnas de la versión desktop de **pas4**, convertido en un grid de una columna en la versión definitiva: uso de “grid-row” indispensable para compensar alternancia texto e imagen en el grid de dos columnas (se puede ver código en contexto en este snippet de GitHub). Se añaden dos líneas separadoras entre los tres pasos. *NOTA: Gradiente de fondo todavía no aplicado en hacer las capturas.*

```
/*MEDIA QUERIES (per a disseny responsive)*/
@media (max-width: 700px) {
  #tresPassos > section {
    grid-template-columns: 1fr; /*Pasamos a una sola columna de grid*/
  }
  /*Invertimos orden de grupo imagen y grupo h1/h2 en texto en paso1 y paso3*/
  #tresPassos > section:first-child section:first-child {grid-row: 2;}
  #tresPassos > section:last-child section:first-child {grid-row: 2;}

  /*Añadimos línea separadora para la página responsive*/
  #tresPassos > section:first-child, #tresPassos > section:nth-child(2) {border-bottom: 1px dashed black;}
}
```



Además, en pas4 utilizamos un *linear gradient* con un ángulo de 135 grados de blanco a naranja claro, muy sutil (ver la siguiente figura, 3.49). Con el fondo blanco ya queda perfecto, como hemos visto en la figura 3.48; pero para darle un toque más congruente con las páginas públicas y la página privada del dashboard hemos visto conveniente aplicarlo dado que usamos esta técnica en muchas de ellas:

Figura 3.49: Gradiente lineal a 135 grados para el diseño del pas4

The dashboard interface features a black header bar with a red 'Cerrar Sesión' button. Below the header, there's a light orange main area divided into three sections by thin vertical lines. Each section contains a title, a descriptive text, and an icon.

- Paso 1: Descarga tus tickets**
Habilita ventanas emergentes y luego auténticate en tu gmail en una cuenta con más de dos tickets digitales dentro de días distintos. Descarga hasta los 500 tickets más recientes automáticamente.
- Paso 2: Mándanos tus tickets**
Selecciona todos los tickets digitales que se descargaron en la carpeta descargas y adjúntalos.
- Paso 3: Relájate.**
Espera a que los guardemos en la base de datos el contenido de tus tickets. Una vez guardados te redirigiremos automáticamente a la página definitiva. Guarda tus tickets en PDF en una carpeta: serán de utilidad luego

Finalmente lo que sí es una diferencia en relación al dashboard es que se ha automatizado la propiedad dropshadow utilizando JavaScript en lugar de hacerlo mediante CSS. La idea era crear una sombra que fuese reactiva al ratón, automatizada cuya dirección sea opuesta a la desviación que hay entre el centro del ícono y el punto donde está el ratón (en caso que se emplace por encima de este ícono).

Esto se ha hecho posible gracias a [este script](#). No fue programado a mano, sino que se generó simplemente con el prompt que aparece en la izquierda de la figura 3.50. Después del código que proporcionó el prompt, se ajustaron los valores de *blur*

del *dropshadow*, la división que afecta al módulo del vector opuesto al vector que va del centro del ícono hasta el ratón para evitar generar una sombra demasiado grande y se añadió un event listener de tipo “DOMContentLoaded” para que cargue después de los elementos del DOM. El propio script se consigue con los eventos *mousemove* y *mouseleave*:

Figura 3.50: Desde el prompt inicial al código final

The diagram illustrates the transition from a user prompt to the final JavaScript code. On the left, a light gray box contains a user prompt in Spanish. An arrow points from this box to the right, where the final JavaScript code is shown in a dark gray box.

```

Quiero implementar un drop shadow que genere sombra dinámica.
Cuando pase el ratón por el "centro de masas" no va a pasar nada
(Solo un transform pero sin drop shadow). Si pongo el ratón por la
derecha de la imagen se generara un dropshadow a la izquierda. Si
pongo el ratón por la izquierda de la imagen se generara un
dropshadow hacia la derecha . En definitiva una desviación de (x,y)
unidades en relación al centro de la imagen (que sería el origen 0,0) va
a producir un dropshadow de (-x,-y) unidades.

Por ejemplo, yo tengo este CSS que ya consigue un drop shadow con
una transformación. Quiero, empero, que esta transformación sea
dinámica, más allá de un simple hover, de acuerdo a las
especificaciones del anterior párrafo:

/*Definim el tamany de les 3 imatges dels 3 passos*/
#tresPassos img {
  width: 15em;
  filter: drop-shadow(7px 6px 2px rgba(0, 0, 0.25));
  transition: filter var(-tiempoTransicionEstandar) ease-in-out,
              transform var(-tiempoTransicionEstandar) ease-in-out;
}
#tresPassos img:hover {
  filter: drop-shadow(10px 2px 2px rgba(0, 0, 0.25));
  transform: scale(var(-aumentoTamanyOEstandar));
}

document.addEventListener("DOMContentLoaded", () => {
  document.querySelectorAll('#tresPassos img').forEach(img => {
    // POSO DROP SHADOW EN POSAR-ME DAMUNT DE LA IMATGE (OPOSAT)
    img.addEventListener('mousemove', e => {
      const rect = img.getBoundingClientRect(); //Rectangle que ocupa l'ícono
      const x = e.clientX - rect.left; // pos ratoli respecte costat esquerre ícono
      const y = e.clientY - rect.top; // pos ratoli respecte costat superior ícono
      const centerX = rect.width / 2; // centre x del rectangle que ocupa l'ícono
      const centerY = rect.height / 2; // centre y " "
      // Diferència relativa al centre (x, y)
      const deltaX = x - centerX;
      const deltaY = y - centerY;
      // Invertim perquè la sombra vagi en la direcció oposada
      const offsetX = -deltaX / 5; // escala la sombra, ajusta el divisor
      const offsetY = -deltaY / 5;

      img.style.filter = `drop-shadow(${offsetX}px ${offsetY}px 2px rgba(0, 0, 0, 0.25))`;
    });
    // FAIG DESAPAREIXER EL DROP-SHADOW QUAN SURTO DE LA IMATGE
    img.addEventListener('mouseleave', () => {
      img.style.filter = 'drop-shadow(0px 0px 0px rgba(0,0,0,0))';
    });
  });
});

```

3.6.9. Arquitectura (JavaScript): páginas privadas

3.6.9.1. Arquitectura del dashboard

“Inflalyzer”

El **paginador customizado** que vimos en la figura 3.45 requiere JavaScript. El código que lo consigue orquestar se encuentra en la ruta [/js/dashboard/paginadorInflacio.js](#).

El **gráfico de inflación**, que también requiere JavaScript, cuando el proyecto esté terminado, tendrá su base de código dentro de [/js/dashboard/graficInflacio.js⁶⁶](#):

“Intervalizer” & “Categoryzer”

El sistema de **gastos por ventana temporal**, es decir, para ver los gastos *totales* de un usuario en distintas ventanas temporales (“intervalizer”) estará en: [/js/dashboard/gastoTotalFinestraTemporal.js](#)

El sistema para crear **diagrama de sectores** (“categorizer”), que también se podrá vincular a las ventanas temporales que se muevan junto con el intervalizer, estará emplazado en: [/js/dashboard/diagramaGastosPerCategoria.js](#)

⁶⁶Este archivo va a depender del anterior, porque al movernos por el paginador los gráficos de inflación cambian de forma acorde al producto visualizado en un momento dado.

NOTA: “categorizer” también mostrará ventanas temporales, junto con el límite inferior y superior del “intervalizer”; con lo cual “categorizer” e “intervalizer” dependen el uno del otro, de forma análoga a como el paginador y el gráfico de inflación del apartado inmediatamente anterior se interrelacionan.

Extracción de la base de datos: solicitudes GET

La extracción de datos de la base de datos se hace principalmente desde el archivo [extractorDadesPersistencia_enCarregarPagina.js](#). Si se necesitasen más archivos vinculados con persistencia en base de datos (con la extracción de la misma) van a tener todos, en el inicio de su nombre de archivo, la palabra “extractor”.

3.6.9.2. Arquitectura pas4.concedirAccesGmail.html

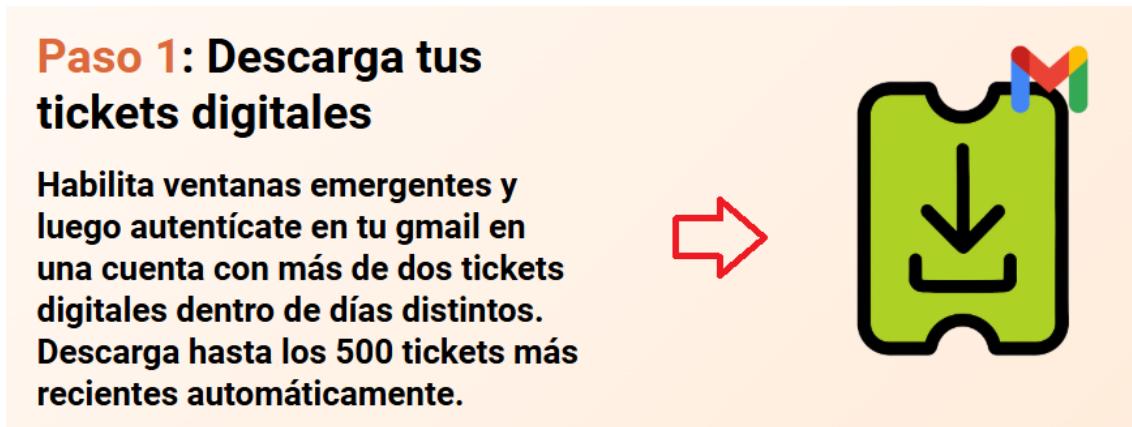
NOTA: las peticiones HTTP hacia FastAPI (*controlador.py*) vistas desde el punto de vista del back-end, se muestran en la sección [3.5.3](#) y son complementarias a este apartado -que sólo está dedicado al front-end-.

PARTE 1: llamada a google API client

Para hacer esta parte es importante mencionar que tenemos que configurar la consola de Google Cloud (<https://console.cloud.google.com/>). En última instancia tenemos que obtener el **Client ID** que identifique nuestra web en relación a Google Cloud. Todos estos pasos se indican en una sección a parte que recomendamos al lector que lea, porque ha supuesto varios días de trabajo: la sección [3.7](#).

Una vez obtenido este id, se permitirá que los usuarios de nuestro sitio web (en el [pas4.concedirAccesGmail.html](#)), siempre que tengan un Gmail con tickets digitales de Mercadona enviados por el supermercado y guardados en el inbox de su correo, puedan descargarse los tickets mencionados a la carpeta de descargas de su ordenador mediante **un solo click**: el click que ellos mismos harán al icono mostrado en la figura [3.51](#). Este click activará script de JavaScript que le mostrará al usuario una ventana de Google que le propondrá autenticarse con *Oauth2* y cedernos control de consulta a su Gmail, para así descargar sus tickets; sin más pasos intermedios, los tickets digitales empezarán a descargarse en su ordenador: dándole de antemano el control de todas sus compras.

Figura 3.51: Pulsando sobre el icono iniciamos la llamada a la API de Gmail: con ello autenticamos el usuario y éste concede acceso a su correo en cuyo inbox se encuentran los tickets digitales, que se descargan al sistema automáticamente.



Para hacer esta búsqueda se utilizará el remitente de la dirección de correo electrónico desde la que Mercadona nos manda los tickets digitales para poder filtrar los correos y descargar sus pdfs (ticket_digital@mail.mercadona.com) y las líneas de código donde conseguimos esta descarga las marcamos en este [snippet](#) de GitHub (pedimos hasta un máximo de 500 pdfs) del archivo javascript con el que conseguimos la descarga: [scriptExtraccionBoto.js](#)⁶⁷.

A medida que hemos ido desarrollando hemos llegado a la siguiente conclusión: Aunque descarguemos 500 pdfs de golpe, Google en principio no va a ralentizar las descargas. Se ha probado para mi caso particular: se descargaron 281 tickets digitales en 2-3 minutos desde mi cuenta de Gmail. Si bien existen unas cuotas máximas que podemos usar para descargar, estas son generosas y no se van a superar para nuestro caso de uso (podéis ver en anexo 5.11 el cálculo de unidades de cuota por ticket descargado, unidades de cuota por minuto por usuario de mercApp y unidades de cuota por minuto del sistema ante usuarios concurrentes).

La extracción de tickets no se puede hacer *en local* con JavaScript porque las políticas de seguridad de Google le dicen al navegador que impida las llamadas a su API (se activa algo llamado la “Content-Security-Policy”): esto pasa porque, entendemos, al no tener HTTPS -porque nuestro proyecto corre en local- el origen no es seguro. Debe hacerse, por lo tanto, desde un script en remoto, en un dominio autorizado en internet, con protocolo HTTPS y autorizado manualmente desde la configuración de la consola de Google (que ya hemos hecho). Para ello se ha creado este repositorio para hacer descargas de tickets [mercAppAuxPas4](#).

De este repo, el único archivo que usaremos para que podamos alimentar nuestro archivo `pas4_concedirAccesGmail` corriendo en localhost será este: [scriptExtraccionBoto.js](#). Para poder correr el script usaremos su ruta en GitHub Pages (no la ruta

⁶⁷Luego explicamos por qué este script pertenece a un repositorio auxiliar

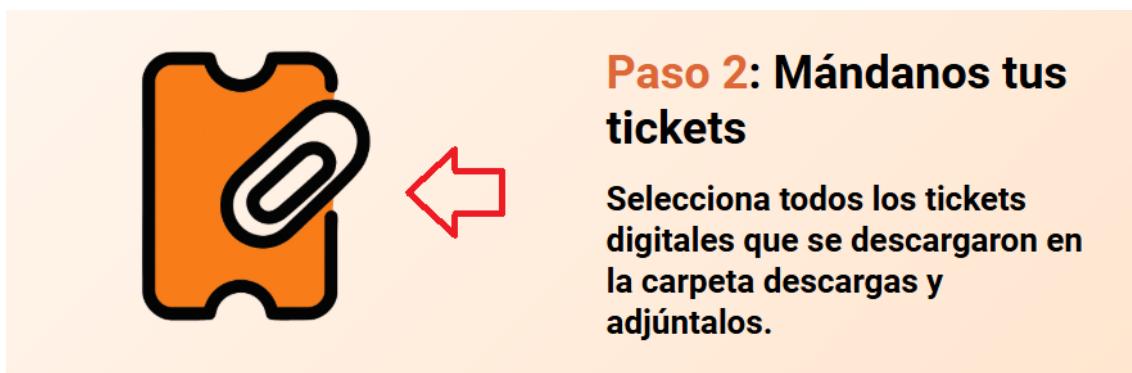
del repositorio GitHub, sino la ruta que tiene el archivo cuando se carga en la página de GitHub Pages que hace el *deploy* de este mismo repositorio). Esto lo haremos haciendo uso de la URL del repositorio, pero sin el grupo “blob/main” en ella. Luego, lo cargaremos en `pas4_concedirAccesGmail.html` como si fuera servido de una CDN externa⁶⁸:

```
<script src="https://blackcub3s.github.io/
mercAppAuxPas4/js/scriptExtraccionBoto.js"></script>
```

PARTE 2: Subir tickets digitales del ordenador del usuario al sistema de archivos del servidor de FastAPI

La subida de tickets digitales desde el front-end la hacemos como vemos en la figura 3.52:

Figura 3.52: llamada al endpoint de FastAPI `/api/subir-tickets-pdf` para subida de tickets mediante una solicitud POST con javascript (fetch), activada con el click en el icono marcado con *la flecha roja*.



El código JavaScript que consigue la subida de tickets digitales al sistema de archivos del servidor de FastAPI conforma íntegramente todo el contenido del archivo `pujaPdfs.js`. El código de ese archivo cobrará vida al clicar en el icono naranja de la figura anterior (evento *click*) que, a su vez, activará un evento *change* anidado que permitirá la apertura de un cuadro de diálogo para seleccionar y subir los PDFs que el usuario desee, dándole al botón abrir. Esto, automáticamente, hará que el archivo que acabamos de comentar haga una solicitud POST con la función *fetch* al endpoint `/api/subir-tickets-pdf` del controlador de FastAPI, subiendo así los tickets en PDF al sistema de archivos -aquí todavía no queremos guardar los datos de los PDFs en base de datos, porque primero tenemos que parsearlos⁶⁹.

Nótese que para subir PDFs necesitamos un formulario: normalmente usaríamos

⁶⁸Solo que la CDN la hemos creado expresamente para la ocasión.

⁶⁹Parsear se podría definir como “minar” o “extraer” los datos del PDF (productos, precios, unidades, código de ticket) a un formato estructurado transferible a objetos del lenguaje de programación.

el botón que se genera con la etiqueta de los formularios *input*. Pero no hay botón. ¿Por qué? Pues porque hemos conseguido que sea la propia imagen la que hace la función de *input*, mientras que el *input* lo hemos escondido (véase figura 3.53).

Figura 3.53: Anidamos la imagen en la etiqueta *label* con valor en atributo *for* compartido con etiqueta *input*. Así conseguimos que la imagen del Paso 2, al ser clicada, tenga comportamiento de botón *input*: abrir menú contextual y adjuntar tickets en PDF. Mediante CSS ocultamos el botón que genera el *input*: ya no es útil.

```

<section class = "contenedorIconos">
  <label for = "inputOcultEntradesPDFs" id = "labelIconoPujar">
    
  </label>
  <input type="file" id="inputOcultEntradesPDFs" multiple accept="application/pdf"/>
</section>

```

estilsEspecificsPas4.css

```

/*OCULTO L'INPUT REAL DEL FORMULARI; ja
#inputOcultEntradesPDFs {display: none;}

```

js/pas4/pujaPdfs.js

```

iconoAdjuntaTickets.addEventListener("click", () => {
  console.log("icono clicat!");
  /*PRIMER DE TOT. PUJO ELS PDFS */
  const input = document.querySelector('#inputOcultEntradesPDFs');
}

```

PARTE 3: Pedir al servidor el parseo de PDF's subidos en el paso 2

Figura 3.54: Llamada al endpoint */api/parsea-y-guarda-pdf-en-bbdd* de FastAPI mediante el click en el icono de engranaje (flecha): con ello conseguimos mandar al servidor una solicitud POST para que, para un usuario determinado -proporcionado token-, parsee los PDF's existentes en la carpeta *tickets/idUsuario* del sistema de archivos del servidor donde esté el back-end de FastAPI: si los PDFs tienen el contenido esperado en su interior ese parseo se persistirá en la base de datos MongoDB y automáticamente se redirigirá al usuario a *dashboard.html*



Como vemos en la figura 3.54 para mostrar el PASO 3 dedicado a procesar o parsear los tickets, si clicamos en el engranaje activaremos una llamada POST a un endpoint del back-end de fastAPI mediante el script [demandaExtraccionPdfs.js](#) que le dará la señal de empezar la extracción y guardar en MongoDB.

3.6.10. Diseño de iconos

3.6.10.1. Áreas de producto

Los iconos de las 12 áreas en las que hemos decidido clasificar los productos de Mercadona (véase apartado 2.2.1, requisito C) se han generado mediante inteligencia artificial generativa y se han editado posteriormente con Gimp para eliminar el fondo y generar transparencia en el mismo (ver [carpeta GitHub](#)).

En la figura 3.55, podemos ver la creación de uno de los iconos con chatGPT y la eliminación posterior del fondo con GIMP.

Hay que mencionar, sin embargo, que el procedimiento de GIMP ahí mostrado no permite conseguir una transparencia pura del fondo. Para conseguirlo en algunos iconos es indispensable usar otro procedimiento que involucra más pasos pero que nos permite asegurarnos que se elimina por completo el canal alpha del fondo (ver figura 3.56). La utilidad de ese último procedimiento en relación al anterior puede verse en la figura 3.57 donde comparamos la aplicación de ambos procedimientos al mismo ícono.

Figura 3.55: Se ha pedido a chatGPT la creación del ícono de verduras y hortalizas. ChatGPT no elimina el canal alpha del fondo automáticamente, así que hemos tenido que tomar la imagen saliente (A), pasarla por Gimp y exportarla sin el fondo (B). Para hacerlo dentro de Gimp seleccionamos el color blanco con la herramienta de selección (varita mágica) y seguimos la ruta *capa → transparencia → color a alpha*.

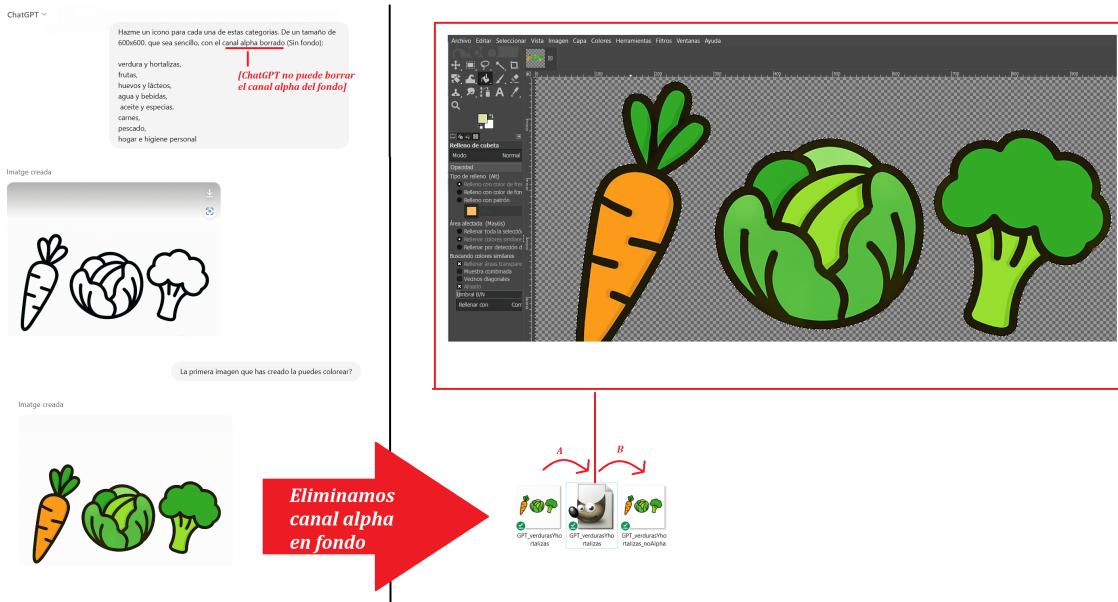


Figura 3.56: En determinados casos no es suficiente con la estrategia sencilla del paso anterior y debemos seguir este procedimiento, que incluye el uso de la varita mágica, y de los pasos *Seleccionar* → *invertir* (invertir selección) y *Ctrl + X* seguido de *Ctrl + V* como mostramos en la figura.

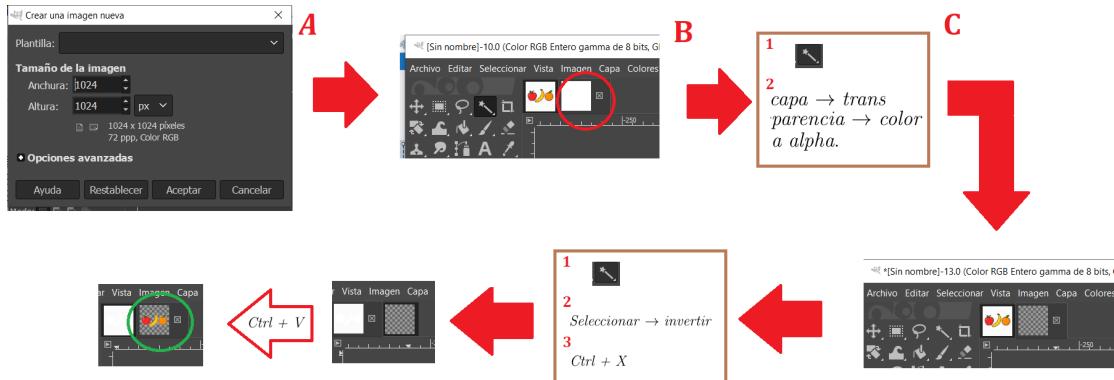


Figura 3.57: El procedimiento de Gimp mostrado en la figura 3.55 genera un ícono que en contexto se verá como la imagen de la izquierda: la card central del dashboard mostraría un ícono con bordes rectangulares no intencionados. El procedimiento de la figura 3.56, en cambio, nos da la imagen de la derecha que no tiene fondo alguno y permite que el ícono se asiente a la perfección en la card.

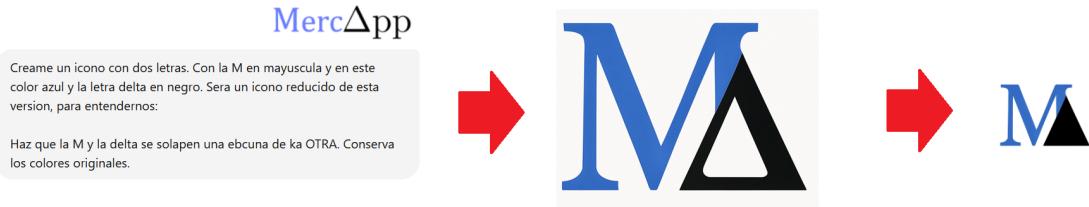


3.6.10.2. Ícono mercApp pequeño

Este ícono también se ha pedido a la IA y se ha editado posteriormente, mostrando el ícono grande para que lo usase. Se puede ver en el anexo 5.9 el proceso completo que también involucra edición en Gimp; Pero el proceso resumido viene a

ser el de la figura 3.58.

Figura 3.58: El prompt pasado a ChatGPT para obtener una versión reducida del icono de nuestra APP. Incluso con typos en el *prompt* la IA entiende perfectamente lo que necesitamos. Último paso completado con Gimp.



3.7. Desarrollo Cloud: configuración google API client

NOTA: Se puede regresar a sección 3.6.9.2 una vez comprendido este apartado, en caso que el lector no la haya leído todavía.

Para conseguir que la extracción de tickets funcione en el **pas4** (sección 3.6.9.2) es indispensable hacer múltiples configuraciones en la página de desarrolladores de google denominada <https://console.cloud.google.com/>.

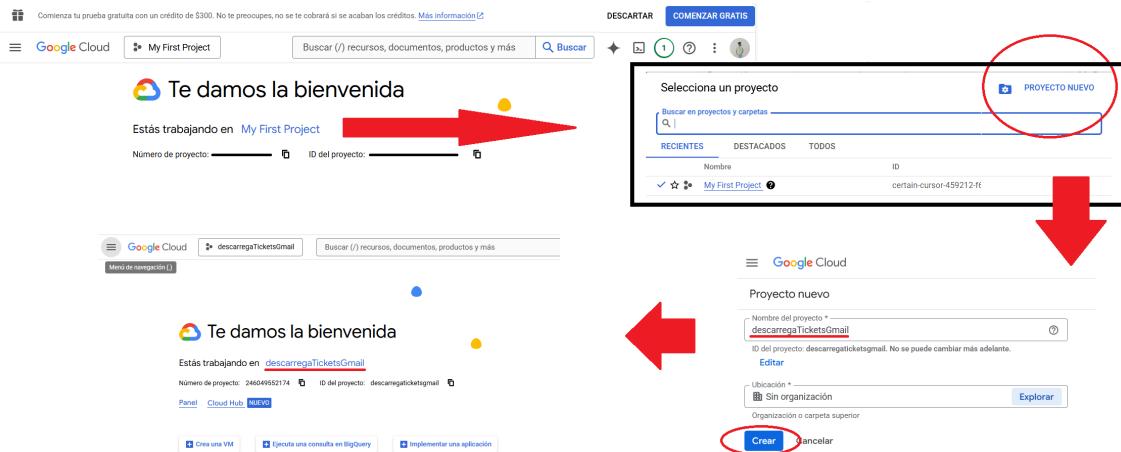
Estas configuraciones son necesarias por varios motivos: el *primero*, es que tenemos que obtener el (**Client ID**) para que se identifique nuestra web en relación a Google Cloud; el *segundo*, es configurar los “*scopes*” que le dicen a Google qué tratamos de conseguir del Gmail de los usuarios que se autoricen ante Google mediante su mecanismo de autenticación denominado Oauth2 (aquí el Scope es solo para lectura de Gmail -*gmail.readonly*-); el *tercero*, definir un listado de usuarios que permitirán usar nuestra aplicación (durante la fase de desarrollo); y el *último*, pero no por ello menos importante, la lista de direcciones IP desde las que Google deberá permitirnos hacer llamadas a su API.

A continuación explicamos todos los pasos de forma pormenorizada para conseguir aplicar de forma satisfactoria las configuraciones que nos permitirán que un usuario pueda autenticarse con OAuth2 ante Google y descargar tickets digitales de su correo Gmail personal:

PASO 1: Crear un nuevo proyecto en la consola

Para crear el nuevo proyecto una vez dentro de la vista general de la consola clicamos encima de *MyFirstProject* e introducimos su nombre: lo llamaremos “descarregaTicketsGmail” (ver figura 3.59):

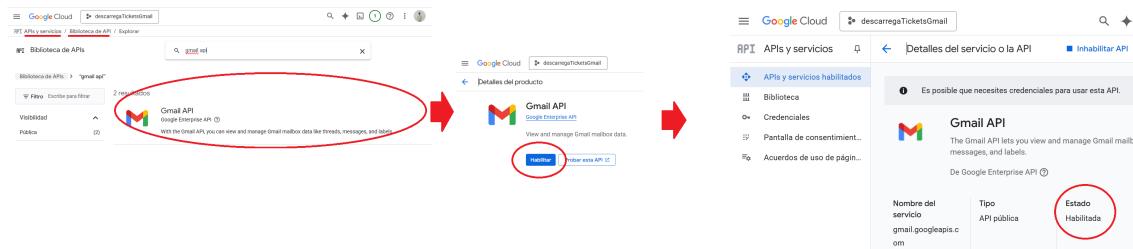
Figura 3.59: Pasos para crear un nuevo proyecto en la consola de google cloud.



PASO 2: Habilitar la Gmail API

Para habilitar la API de Gmail, que nos permitirá **ver y manejar datos del inbox de un correo electrónico** (que es la que usaremos para descargar los tickets adjuntos en PDF), debemos entrar en nuestro proyecto recién creado y seguir los menús “*APIs y servicios*” → *Biblioteca*. Ahí, buscar “*gmail API*” y habilitarla (figura 3.60).

Figura 3.60: Habilitación de la GmailAPI en el proyecto “descarregaTicketsGMAIL”



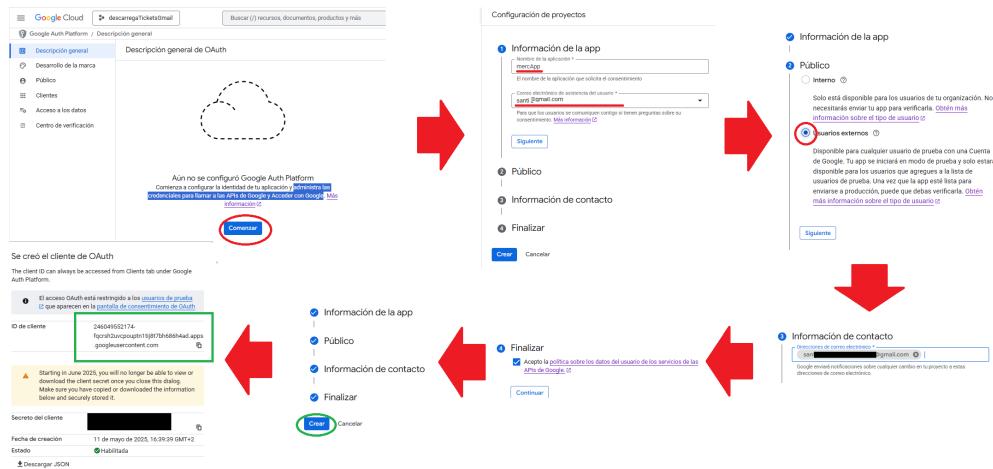
PASO 3: Configurar la pantalla de consentimiento OAuth

Oauth nos permitirá que los usuarios sean autorizados a servicios de Google como Gmail y, además, también les autenticará para dejar que nuestra aplicación en JavaScript del `pas4_concedirAccesGmail.html` pueda acceder a su correo Gmail⁷⁰.

Análogamente al paso anterior seguimos los menús de la izquierda “*APIs y servicios*” → *Pantalla de consentimiento OAuth* (ver figura 3.61).

⁷⁰¡Cuidado, no confundir la autenticación y autorización de nuestra aplicación y el manejo de nuestros AccessTokens con la autenticación con Oauth 2.0: esta última es OBLIGATORIA para acceder a los servicios de Google y la controla Google.

Figura 3.61: Habilitación de la pantalla de consentimiento de OAuth: Especificamos el nombre de la aplicación, el correo al que podrán comunicarse los usuarios de la aplicación mercApp del cloud si es necesario, marcamos el radioButton “usuarios externos”, que hará que puedan utilizar la app en el cloud usuarios sin una extensión de correo vinculada a una organización específica (si no lo marcásemos, estaríamos limitados a usuarios con una extensión específica de organización y no podríamos usar correos gmail!).



PASO 4: Crear credenciales OAuth 2.0 para Web

Al pinchar en el círculo verde de la imagen 3.61 obtenemos una ventana de configuración⁷¹ que nos permite modificar aquello para lo que sirve el *ClientID*⁷² como podemos ver en la imagen 3.62. A través de esta configuración tenemos que decirle a Google de qué dominios y de qué puertos puede recibir llamadas para autenticarse.

⁷¹Podemos llegar a esa ventana de configuración -es decir al contenido de la figura 3.62- del mismo modo si optamos por seguir una ruta por menús: *Apis y servicios* (buscador) → *credenciales* → *+ crear credenciales* → *ID de cliente Oauth*.

⁷²El clientID identifica la aplicación que corre en el cloud que estamos creando (descargaTicketsGmail). Sin él un usuario no podría llegar a hacer una llamada autenticándose con su cuenta de Google hacia nuestro cloud.

Figura 3.62: Configuración de Client ID. | *: http://localhost:5500 no habrá manera que funcione porque saltará la Content-Security-Policy que impide que hagamos llamadas desde local. Si no hubiera esta CSP, para desarrollo estaría bien, pero en producción habría que tener mucho cuidado, porque si alguien accediese al código con cualquier IP de localhost (cualquier ordenador del mundo) y un vscode con live server, podría hacer llamadas usando el cloud a nuestro nombre | **: En producción añadiremos otros orígenes que el de localhost: hay que poner aquí la IP fija de nuestro servidor o el dominio (no tenemos uno todavía), siempre habilitando protocolo HTTPS, sino, no funcionará. | ***: Para el despliegue en local tendremos que confiar en github pages para hacer la extracción.



Después de la configuración hecha en la imagen 3.62 ésta será la vista que tendremos si entramos en el buscador, buscamos **credenciales** y luego vemos los IDs de clientes OAuth2.0:

Figura 3.63: Vista de la aplicación cloud después de la creación del ClientID

Nombre	Fecha de creación	Tipo	ID de cliente	Acciones
mercAppWebP4	11 may 2025	Aplicación web	246849552174-fqcr...	

PASO 5: Añadir usuarios al listado de usuarios permitidos (desarrollo)

Google no deja que cualquier usuario pueda autenticarse con el **clientID**. Para ello o bien hacemos lo que veremos en el PASO 6 (que veremos que es solo para producción, porque requiere una aplicación ya hecha y justificar a Google su uso mediante una muestra que ellos -dicen- que van a evaluar) o bien guardamos una lista de usuarios autorizados a autorizarse -valga la redundancia- para que así puedan leer su Gmail y descargar sus tickets automáticamente. Esto lo haremos así: yo añadiré el correo electrónico Gmail donde Mercadona me manda a mi los tickets digitales (ver figura 3.64) y con eso la autenticación será posible, por ahora, solo para este correo.

Figura 3.64: Añadir usuario que podrá autenticarse.

Límite de usuarios de OAuth [?](#)
Mientras el estado de publicación sea "Prueba", solo los usuarios de prueba podrán acceder a la app. El límite de usuarios permitidos antes de que se verifique la app es de 100, y se calcula según el ciclo de vida completo de la app. [Más información](#)

1 usuario (1 de prueba, 0 de otro tipo)/límite de usuarios de 100

Público

Estado de publicación [?](#)
Prueba [Publicar aplicación](#)

Tipo de usuario
Usuarios externos [?](#)
[Marcar como interno](#)

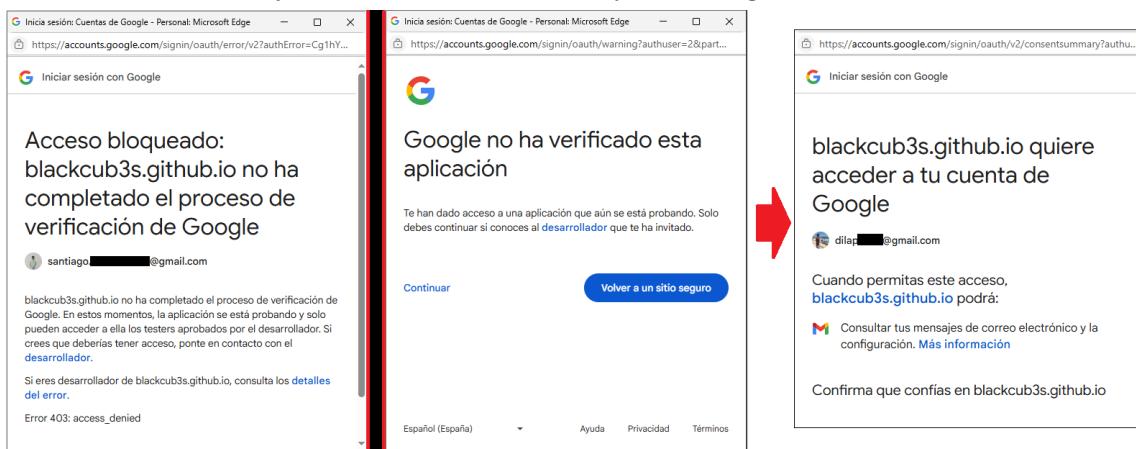
+ Add users

Filtro Ingresar el nombre o el valor de la propiedad [?](#)

Información del usuario
dilap[REDACTED]@gmail.com [Borrar](#)

A continuación podemos ver en la figura 3.65 qué pasa cuando tratamos de acceder a la autenticación a través de Google:

Figura 3.65: **IZQUIERDA**: tratamos de loguearnos con un usuario que no está entre los usuarios de prueba. **DERECHA**: intentamos acceder con el correo que sí está entre los usuarios de prueba y Google nos informa que estamos queriendo consultar los mensajes de correo electrónico y su configuración.



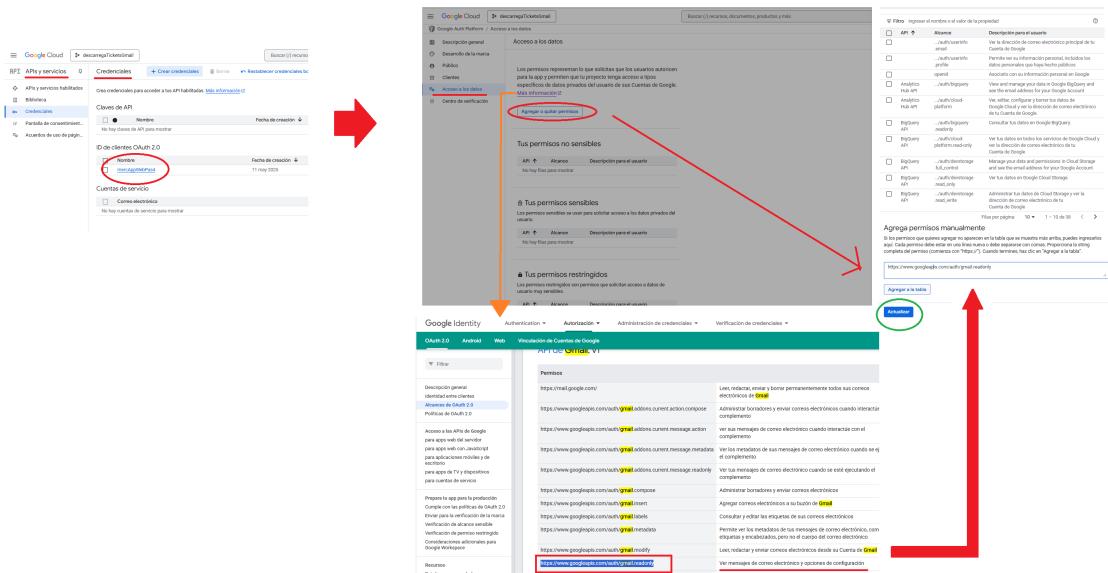
PASO 6: Validar la aplicación descargaTicketsGmail para que cualquier usuario externo pueda usarla.

Importante mencionar que para que cada usuario pueda acceder a su Gmail programáticamente tendremos que habilitar un *scope* de permisos específico que se denomina *gmail.readonly* y deberemos verificar aplicación en el cloud. Si no lo hiciéramos, tal y como está la configuración de "usuarios externos", por ahora, solo podríamos conseguir que un usuario usase nuestra aplicación cloud de descarga

de pdfs del correo mediante el **Client ID** (que estará en el pas4 del navegador) si también estuviese añadido manualmente a una *lista de verificación* de nuestro cloud.

Esto lo que haría sería que no nos permitiría escalar la aplicación web⁷³. Para solucionarlo tenemos que añadir el scope necesario como vemos en la figura 3.66:

Figura 3.66: Ahora vamos a “*Apis y servicios*” → “*Credenciales*” clicamos en la credencial oauth2 creada “mercAppWebPas4” y en acceso a los datos clicamos en agregar permisos. Añadimos el permiso pertinente que encontramos mediante la adición de un “scope”- Para ver los scopes podemos acceder al link “más información”, que nos lleva a [ésta página](#), y de ahí copiar el scope necesario (usamos **gmail.readonly**) y lo actualizamos con el botón azul marcado en el círculo verde de la imagen.



Después de darle a actualizar en la imagen 3.66 tenemos que guardar esta configuración. Para ello seguimos los pasos de la imagen 3.67; en caso contrario estaremos acotados al listado de usuarios -correos- que proporcionemos.

⁷³Si os fijáis en la figura 3.61 donde pone Usuarios externos, tenemos una letra pequeña debajo donde se deja claro que por defecto el servicio cloud que acabamos de configurar solo admite usuarios que estén en esa lista.

Figura 3.67: Último paso para poder permitir que los usuarios de nuestra aplicación puedan acceder a leer SUS correos electrónicos una vez autenticados, con oauth2. Esta vez vamos a “*Apis y servicios*” → “*Credenciales*” → “*MercAppWebPas4*” → “**“Acceso a los datos”** y le damos al botón guardar.

The screenshot shows the Google Auth Platform interface. On the left, there's a sidebar with options: Descripción general, Desarrollo de la marca, Público, Clientes, and Acceso a los datos (which is highlighted). Below that is Centro de verificación. The main area is titled 'Acceso a los datos' and features a lock icon and the heading 'Tus permisos restringidos'. It explains that restricted permissions are for sensitive user data. A table lists a single permission: 'Ver mensajes de correo electrónico y parámetros de configuración' (Scope: ./auth/gmail.readonly). At the bottom, there are 'Save' and 'Descartar cambios' buttons, with 'Save' being circled in green.

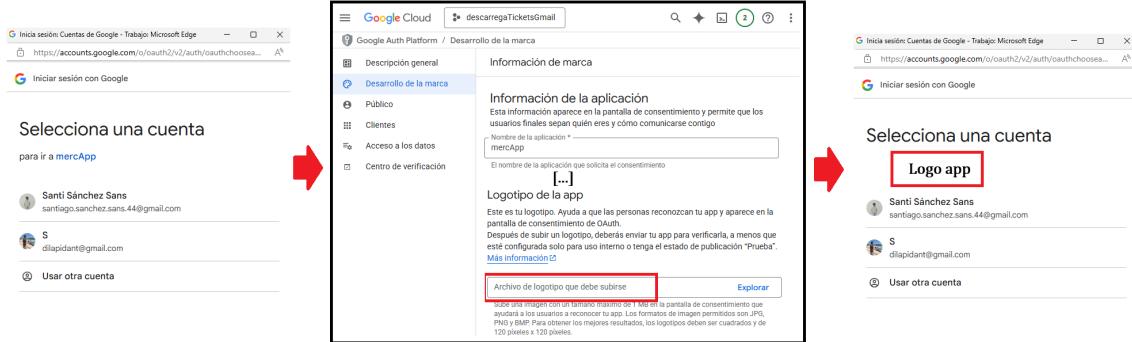
Y el último caso es *validar*. Si Google no nos da el visto bueno no podremos escalar la aplicación para que usuarios no informados en la lista de Gmails permitidos la usen a través de nuestro ClientID.

Sin embargo, damos por asumido que esto no va a pasar, al menos no en plazo para la entrega de este proyecto, con lo cual directamente solo podremos usar para descargar los tickets digitales los correos que estén informados en el listado de usuarios permitidos mostrado en el PASO 5: por ahora solo uno, el mío.

PASO 7 (Opcional): definir un logo para la aplicación

Para que al iniciar la autenticación con Google salga un ícono que lo vincule con nuestra aplicación, no solamente el nombre de mercApp, podemos subir un ícono como vemos en la figura 3.68. Como podemos ver en esta entrada de Stackoverflow [17], este logo va a aparecer solamente para aplicaciones verificadas por Google (en el momento de escribir estas líneas la nuestra no lo está y por ello seguirá apareciendo un link a la ubicación de la aplicación y nuestro mail de desarrollador).

Figura 3.68: Definición de icono saliente para mostrar a los usuarios que se autentiquen “Apis y servicios” → “Credenciales” → “MercAppWebPas4” → “Desarrollo de la marca”



3.8. Bases de datos

3.8.1. Base de datos mySQL

La base de datos MySQL es la que se conecta al framework Spring Boot. Es muy sencilla: tiene simplemente una tabla de usuarios y otra tabla de información ampliada de usuarios, que es hija, y mantiene una relación de dependencia con la primera. La tabla UsuariAmpliat no se ha utilizado para este proyecto, pero queda diseñada para futuras expansiones donde a determinados usuarios -no a todos- se les puedan pedir ampliaciones de datos como nombre y apellidos.

En este [link](#) podéis ver el archivo con el DDL de la BBDD., mientras que en la figura 3.69 que sigue, podéis ver el esquema de la base de datos:

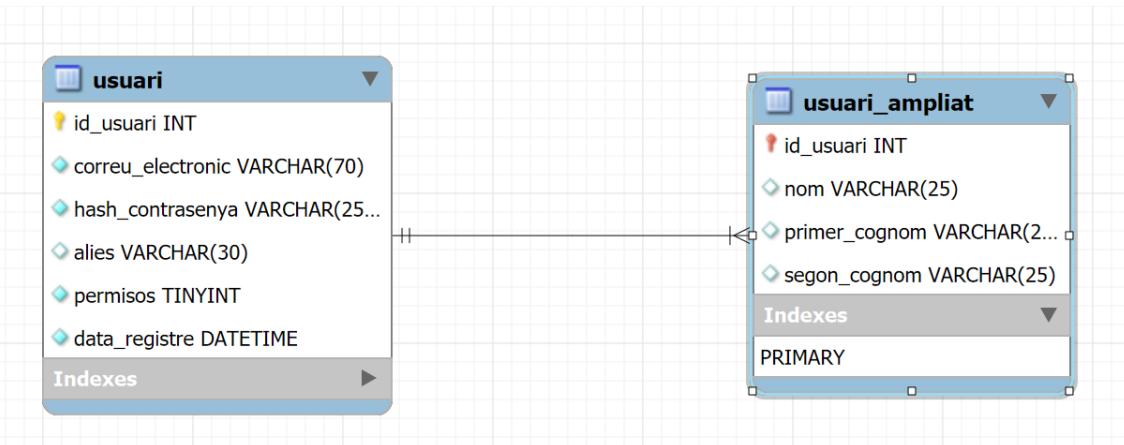


Figura 3.69: Esquema de las dos tablas de la BBDD a la que nos conectamos con el framework Spring Boot

3.8.2. Base de datos mongoDB

Los datos de cada ticket se guardarán como un documento independiente de una colección, a la que denominaremos tickets. En mongoDB una colección es lo que en mySQL sería una tabla; y un documento BSON sería lo que en una tabla mySQL sería una fila.

Con una sola colección nos basta para poder hacer las operaciones necesarias para resolver los problemas que entrañan los servicios del dashboard: filtrar por productos y por usuarios y calcular evolución de precios, hacer costos agregados por meses y por usuario, etc.

Los archivos necesarios para la base de datos no están en ninguna parte porque es una base de datos NoSQL sin esquema rígido, a diferencia de mySQL. Las operaciones de guardado y definiciones se encontrarán dentro de los archivos Python correspondientes (principalmente [repositoryTickets.py](#)).

Evaluación y Conclusiones Finales

El proyecto aquí presente me ha permitido llevar a término todas estas actividades y aprendizajes:

- Definir dos back-ends distintos desde cero: uno orientado a los usuarios y el otro orientado a los tickets digitales.
- Aprender un nuevo sistema de base de datos NoSQL, como es MongoDB.
- Programar el front-end de la aplicación con “vanilla” JavaScript gestionando programáticamente la visibilidad de sus vistas a partir de los tokens de acceso emitidos desde el back-end.
- Entender cómo funciona un JWT de acceso y de qué forma generarlo programando clases customizadas en Java, desde cero.
- Definir desde cero el CSS de todas las páginas o vistas y aprender a utilizar *media queries* sin confiar en el uso de frameworks de CSS como Tailwind o Bootstrap.
- Aprender a conectar componentes del back-end: aprender a hacer solicitudes de fastAPI hacia Spring Boot (con librería de Python httpx), y de Spring Boot hacia fastAPI (con librería de Java Rest Client).
- Definir una base de datos relacional sencilla en mySQL.
- Entender cómo funciona el mapeo de objetos Java a entidades de base de datos relacional mediante JPA (Java Persistence API).
- Utilizar las librerías Chart.js (gráficos), animate.css (animaciones prediseñadas en CSS) y wow.js (complemento para animate.css).
- Contenerizar los componentes del back-end y el front-end, automatizando la creación de imágenes y arranque de contenedores con scripts en bash.
- Hacer resúmenes de datos, paginadores, gráficos en JavaScript a partir de la información extraída de los Endpoints de FastAPI con solicitudes GET, desde el dashboard.
- Aprender a configurar la consola de google y hacer uso de un servicio de Google Cloud como es la Gmail API de Google Api Client mediante un script desplegado en GitHub *pages* para evitar problemas con la CSP (“Content Security Policy”)

- Aprender a configurar los servicios back-end para que permitan orígenes cruzados sin que den problemas de la *CORS policy*: es decir, que no se impidan llamadas desde el front-end hacia el back-end porque ambos corran en distintos puertos.
- Diferenciar entre autorización y autenticación y aprender a configurar en Java el correcto uso de ambos “mecanismos” para securizar correctamente los endpoints de un controlador.

Por todo ello he de decir que ha sido una experiencia enriquecedora, aunque intensa por el poco tiempo disponible juntamente en paralelo a la realización de las prácticas.

Como conclusión para futuros proyectos existen aspectos de mejora como los patrones de diseño seguidos en fast-API y el front-end que no obedecen realmente a buenas prácticas sino a intuición sobre qué funciona y qué no (se ha intentado, eso sí, en la medida de lo posible seguir el principio de separación de responsabilidades). El proyecto Spring Boot sí ha sido correctamente organizado gracias a que la empresa de prácticas me dió un curso de Spring Boot y tuve una mentoría mediante la cual un compañero me decía qué aspectos pulir de la API REST que iba haciendo.

Otro aspecto a mejorar sería el aprender a implementar Refresh Tokens y aprender sistema de desarrollo web front por componentes: Angular, Vue o React. Por ahora, sin embargo, el uso de vanilla javascript, html y CSS no ha sido sino una forma genial de reforzar los contenidos aprendidos en el grado superior durante estos dos últimos años que seguro serán una buena base para el futuro.

Bibliografía

- [1] European Parliament. Parliamentary question: Traces of endocrine disruptor bpa in tickets and receipts. https://www.europarl.europa.eu/doceo/document/P-8-2019-001136_EN.html, 2019. European Parliament.
- [2] José-Manuel Molina-Molina, Inmaculada Jiménez-Díaz, Montserrat Plata-Aquino, Juan-Carlos Martínez-Escoriza, Nicolás Olea, and Mariana F. Fernández. Determination of bisphenol a and bisphenol s concentrations and assessment of estrogen- and anti-androgen-like activities in thermal paper receipts from brazil, france, and spain. *Science of The Total Environment*, 647:1088–1096, 2019.
- [3] Canal UGR. Los tickets de la compra en los que se borra la tinta pueden provocar cáncer e infertilidad. Consultado en Canal UGR.
- [4] Chart.js. Chart.js — open source html5 charts for your website. <https://www.chartjs.org/>, 2025. Accessed: 2025-04-12.
- [5] Animate.css. Animate.css — a cross-browser library of css animations. <https://animate.style/>, 2025. Accessed: 2025-04-12.
- [6] Wow.js. Wow.js, 2025. Accessed: 2025-05-25.
- [7] Stack Overflow Community. What is the purpose of a “refresh token”? - stack overflow. <https://stackoverflow.com/questions/38986005/what-is-the-purpose-of-a-refresh-token>. Accedido el 6 abril de 2025.
- [8] Stack Overflow Community. Is it secure to send token in header of the request? <https://stackoverflow.com/questions/63225061/is-it-secure-to-send-token-in-header-of-the-request>, 2020. Accedido el 6 abril de 2025.
- [9] jwt.io. Json web tokens - jwt.io. <https://jwt.io/>. Accedido el 27 marzo de 2025.
- [10] Stack Overflow Community. Should refresh tokens in jwt authentication schemes be signed with a different secret than the access token? <https://stackoverflow.com/questions/63092165/should-refresh-tokens-in-jwt-authentication-schemes-be-signed-with-a-different>, 2020. Accedido el 28 marzo de 2025.
- [11] Postman. Postman api platform. <https://www.postman.com/>. Accedido el 9 abril de 2025.

- [12] Spring Framework. Onceperrequestfilter (spring framework api).
<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/filter/OncePerRequestFilter.html>. Accedido el 28 marzo de 2025.
- [13] Spring Security. Bcryptpasswordencoder (spring security api).
<https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html>. Accedido el 21 abril de 2025.
- [14] PyPDF2: A pure-python pdf library. <https://pypi.org/project/PyPDF2/>, 2023. <https://pypi.org/project/PyPDF2/>.
- [15] Michael Davis. python-jose: A jose implementation in python. <https://pypi.org/project/python-jose/>, 2025. Version 3.4.0, available on PyPI.
- [16] Wikipedia. Número áureo - wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/N%C3%BAmero_%C3%A1ureo. Accedido el 6 mayo de 2025.
- [17] Stack Overflow Community. Google oauth consent screen not showing app logo and name. <https://stackoverflow.com/questions/44138213/google-oauth-consent-screen-not-showing-app-logo-and-name>, 2017. Accedido el 14 mayo de 2025.
- [18] NGINX. Nginx — high performance load balancer, web server, & reverse proxy. <https://www.nginx.com/>. Accedido el 8 abril de 2025.
- [19] Google Developers. Cuotas y límites del api de gmail. <https://developers.google.com/workspace/gmail/api/reference/quota?hl=es-419>. Accedido el 14 mayo de 2025.

ANEXO

5.1. Flujo de trabajo habitual en git

```
# trabajamos con el proyecto y se introduce
# en el staging area
git add -A

# creamos rama para aglutinar los cambios
git branch backEnd

# cambiamos a la rama que acabamos de crear
git checkout backEnd

# guardamos los cambios como nodos dentro de
# la rama con la que desarrollamos.
git commit -m "commit 1"
git commit -m "commit 2"
# [...]
git commit -m "commit n"

#cambiamos a rama main local y luego integramos cambios
git checkout main
git merge backEnd

#Subimos los cambios al repo remoto
git push origin main
```

5.2. Diferencias de seguridad: JWT vs SESSID en cookies seguras

Característica	JWT en cookies seguras	Session ID en cookies seguras
Seguridad contra XSS	Más seguro si la cookie tiene <code>HttpOnly</code> y <code>Secure</code> , ya que no es accesible desde JavaScript.	Más seguro si la cookie tiene <code>HttpOnly</code> y <code>Secure</code> , ya que no es accesible desde JavaScript.
Seguridad contra CSRF	Puede ser vulnerable si la cookie no tiene <code>SameSite=Strict</code> .	Menos vulnerable si la cookie tiene <code>SameSite=Strict</code> .
Estado en el servidor	Stateless (no hay estado en el servidor, el JWT contiene toda la información).	Stateful (el servidor mantiene una sesión activa asociada con el Session ID).
Escalabilidad	Mejor escalabilidad porque no requiere almacenamiento de sesiones en el servidor.	Menos escalable, ya que el servidor debe manejar las sesiones activas.
Expiración y revocación	Difícil de revocar antes de que expire, a menos que se implemente una lista negra en el servidor.	Fácil de invalidar eliminando la sesión en el servidor.
Uso con JavaScript	No accesible desde JavaScript si la cookie tiene <code>HttpOnly</code> .	No accesible desde JavaScript si la cookie tiene <code>HttpOnly</code> .

Cuadro 5.1: Comparación de seguridad entre JWT y Session ID almacenados en cookies seguras con `HttpOnly=True`.

5.3. Clases para crear y verificar JWTs

5.3.1. Clase JWT

```
//NO INSTANCIEM AQUESTA CLASSE MAI. LA FEM ABSTRACTA
@Component
public abstract class JwtUtil {

    //es la clau privada de 256 bits com a minim per encriptar el token (tant el d'accés com el de refresh)
    //veure debat http://bit.ly/3RmBGK
    protected static String clauSecreta;

    public JwtUtil() {
        this.clauSecreta = "a8f7d9g0b6c3e5h2i4j7k1l0m9n8p6q3r5s2t1u4v0w9x8y7z";
    }

    //METODE QUE PARSEJA EL TOKEN JWT COMPLET. VERIFICA LA FIRMA I EXTRAU LES CLAIMS (parells clau valor en el payload).
    protected Claims getClaims(String token) {
        return Jwts.parser()
            .setSigningKey(clauSecreta.getBytes())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }

}
```

5.3.2. Clase Refresh Token

```
@Component
public class RefreshToken extends JwtUtil {

    private static int tExpDies;

    public RefreshToken() {this.tExpDies = 7;}

    // FINALITAT DEL METODE: Refrescar el token d'accés que genera generaAccesToken().
    public String generaRefreshToken(String correu, int idUsuari) {
        Map<String, Object> dadesExtraApayload = new HashMap<>();
        dadesExtraApayload.put("idUsuari", idUsuari);
        //posar mes dades al payload si es necessari

        return Jwts.builder()
            .setClaims(dadesExtraApayload) //dades customitzades
            .setId(String.valueOf(UUID.randomUUID().toString())) //id únic per a token. Per traçabilitat
            .setSubject(correu) //guardo nom subjecte (dins "sub")
            .setIssuedAt(new Date()) //data creació
            .setExpiration(new Date(System.currentTimeMillis() + tExpDies*86400*1000)) //expiració
            .signWith(SignatureAlgorithm.HS256, clauSecreta.getBytes())
            .compact();
    }
}
```

5.3.3. Clase Access Token

```
@Component
public class AccessToken extends JwtUtil {

    private static int tExpM; //minuts per a exprimir el token

    public AccessToken() {this.tExpM = 10;}

    //FINALITAT: Generar un JWT d'accés.
    public String genera(String correu, int idUsuari, byte permisos) {
        Map<String, Object> dadesExtraApayload = new HashMap<>();
        dadesExtraApayload.put("permisos", permisos);
        dadesExtraApayload.put("idUsuari", idUsuari);
    }
}
```

```

        .setClaims(dadesExtraApayload) //dades customitzades
        .setSubject(correu)           //guardo nom subjecte (clau "sub")
        .setIssuedAt(new Date())      //data creacio (clau "iat" payload)
        .setExpiration(new Date((System.currentTimeMillis() / 1000 + (tExpM * 60)) * 1000))
        .signWith(SignatureAlgorithm.HS256, clauSecreta.getBytes())
        .compact();
    }
}

```

5.4. Clases de seguridad

5.4.1. Clase ConfiguracioSeguretat.java

```

package miApp.app.seguretat;

@Configuration
@EnableMethodSecurity
@PreAuthorize
public class ConfiguracioSeguretat {

    private final FiltreAutenticacioJwt jwtAuthenticationFilter;

    public ConfiguracioSeguretat(FiltreAutenticacioJwt jwtAuthenticationFilter) {
        this.jwtAuthenticationFilter = jwtAuthenticationFilter;
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
            throws Exception {
        http
                .csrf(csrf -> csrf.disable())
                .authorizeHttpRequests(auth -> auth
                        .requestMatchers("/api/correusUsuaris").hasRole("ADMIN")
                        .requestMatchers("/api/usuaris/*").hasAnyRole("USER", "ADMIN")
                        .requestMatchers("/api/usuaris").hasRole("ADMIN")
                        .requestMatchers("/api/nreUsuaris").hasAnyRole("USER", "ADMIN")
                        .requestMatchers("/api/*/contrasenya").hasAnyRole("USER", "ADMIN")
                        .requestMatchers("/api/**").permitAll()
                )
                .addFilterBefore(jwtAuthenticationFilter,
                        UsernamePasswordAuthenticationFilter.class);
    }

    return http.build();
}

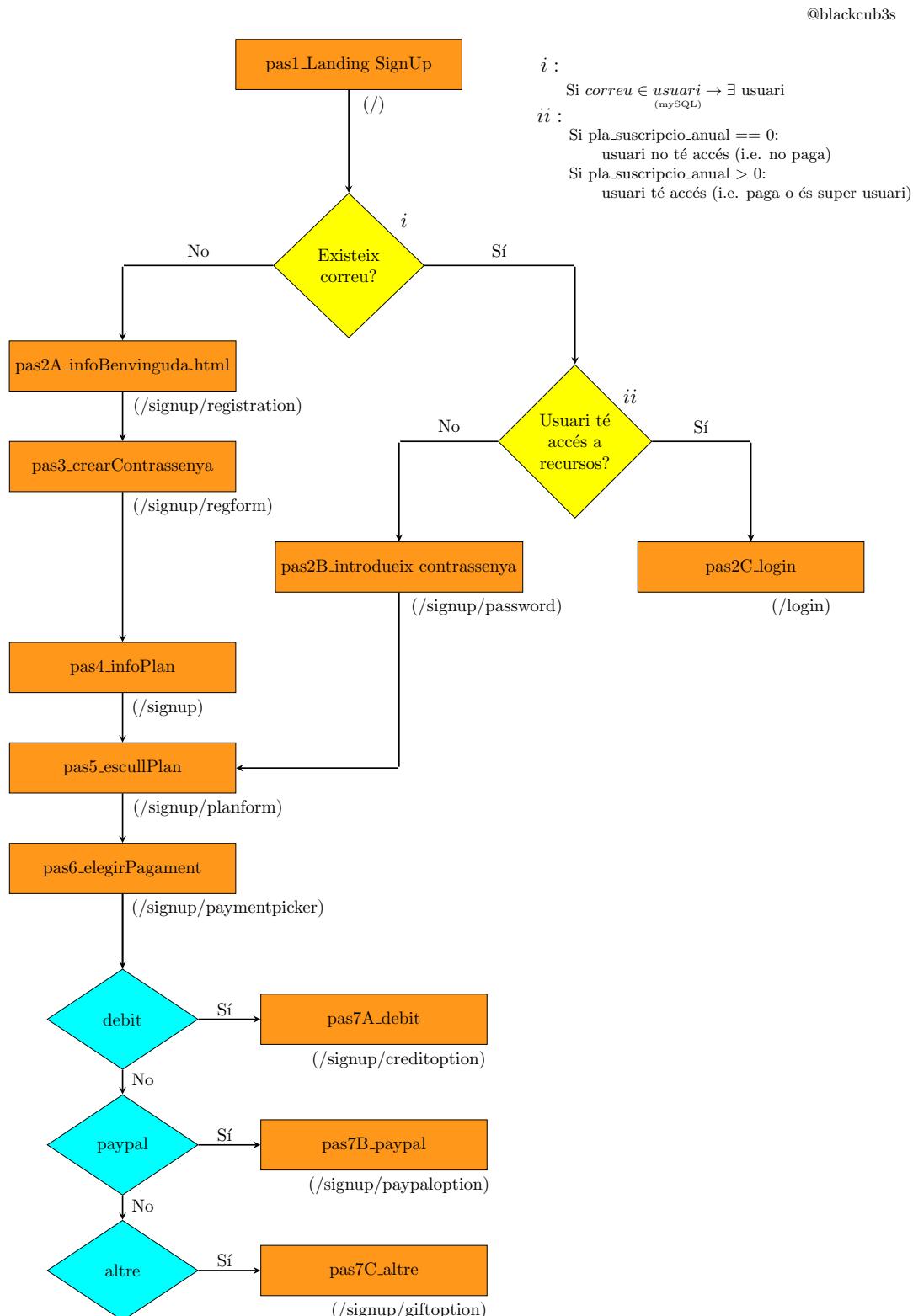
```

5.4.2. Controlador con restricciones aplicadas

La restricción aplicada es `@PreAuthorize`. Solamente usuarios que tengan rol administrador (ADMIN) o bien usuarios que contengan el id == principal (el propio id del usuario autenticado) podrán cambiar la contraseña:

```
@PatchMapping("usuaris/{id}/contrasenya")
@PreAuthorize("hasRole('ADMIN') or #id == principal")
public ResponseEntity<HashMap<String, String>> actualitzaContraseña(
    @PathVariable("id") int id, @Valid
    @RequestBody ActualitzaContraseñaDTO dto) {
    Optional<Usuari> usuariActualitzatOPTIONAL =
        serveiUPP.actualitzaContraseña(dto, id);
    HashMap<String, String> resposta = new HashMap<>();
    if (usuariActualitzatOPTIONAL.isPresent()) {
        resposta.put("mensaje", "Contrasena actualizada correctamente.");
        return new ResponseEntity<>(resposta, HttpStatus.OK); //200 OK
    } else {
        resposta.put("mensaje", "Usuario no encontrado.");
        return new ResponseEntity<>(resposta, HttpStatus.NOT_FOUND);
    }
//404 NOT FOUND
}
```

5.5. Diagrama réplica netflix



NOTA: El lector puede ver el proceso de creación de este diagrama en el repositorio [diagramaTikz](#). También puede ver una explicación del diagrama a continuación:

Este diagrama se puede entender del siguiente modo:

1. Cada rectángulo de color naranja es una página estática .html de lo que sería una réplica de la página de registro de netflix.
2. Cada rombo de fondo amarillo es una decisión que se hará dentro del back-end de Spring Boot, dado que requiere hacer consultas a la BBDD y contiene datos sensibles.
3. Los rombos de fondo azul se decidirán en el front-end en tanto que sus decisiones no requieren consultar información personal en la base de datos y no precisan, por lo tanto, del uso del back-end (y, además, no se explicarán en este *readme*).
4. El paréntesis que incluye la extensión de una URL debajo de cada rectángulo naranja es cada página de Netflix cuyo comportamiento y, en menor medida, aspecto, se ha intentado replicar en el archivo .html del rectángulo naranja que le es contiguo. Por ejemplo, el archivo `pas2A_infoBenvinguda.html` de este proyecto es una réplica de la página especificada en el paréntesis `netflix.com/signup/registration` y el usuario llegará a ella a través del proceso de registro gracias a la aplicación de una lógica de back-end similar a la que usa Netflix.

5.6. Diagrama enrutamiento mercApp

5.7. Aspectos ventajosos de separar front-end y back-end (SoC)

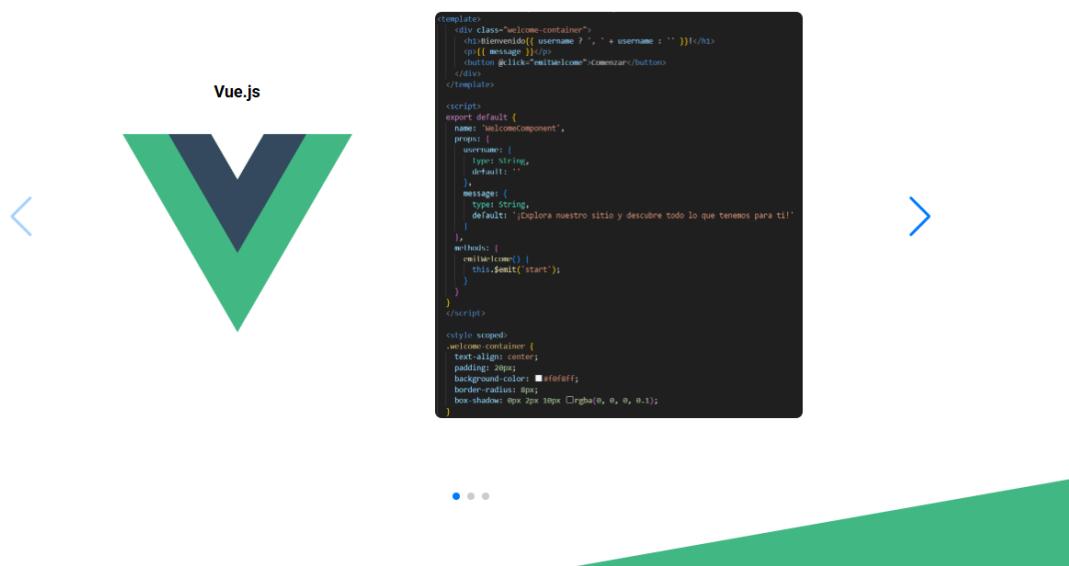
- **Responsabilidad única (SRP):** Cada módulo debería hacer una sola cosa (`{Frontend → interfaz}`, `{Backend → procesamiento}`).
- **mantenibilidad:** Al usar una arquitectura modular cada parte puede evolucionar por separado. Podemos desplegar solo el front o el back. En el futuro podremos cambiar el front-end de archivos estáticos por un front-end con Angular, por ejemplo.
- **Escalabilidad:** Según la carga podemos escalar independientemente ambas partes del proyecto. Por ejemplo, poner los archivos estáticos (html, css y js del front-end) en un servidor para servirlos rápidamente como nGinx [18] (supuestamente más rápido que Apache). Dejar en el tomcat embedido de springboot el procesamiento del back-end y si hay problemas de escalabilidad escalar este independientemente en AWS, AZURE, o en un servidor propio según sea más conveniente.

- **Reutilización:** Un backend puede servir varios front-ends (no solo web, sino móvil también). El mismo front-end podemos reutilizarlo luego para otra aplicación con un back-end en otro lenguaje por ejemplo.

5.8. Captura proyecto desarrollo interfaces

Figura 5.1: Detalle del paginador de swiper del proyecto de desarrollo de interfaces.

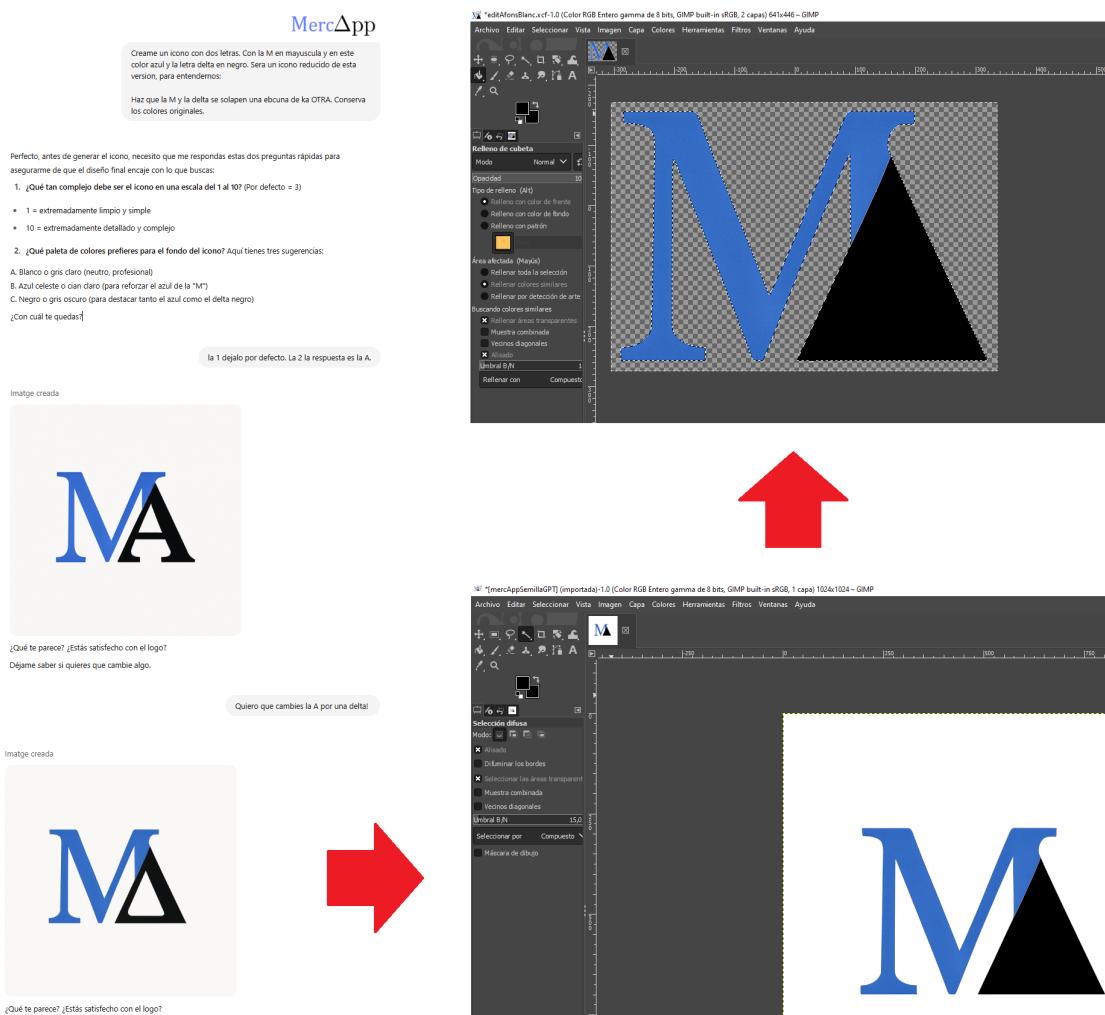
¿Qué diferencias fundamentales hay entre los tres frameworks?



5.9. Creación del ícono mercapp pequeño: IA con Gimp

NOTA: Podéis regresar al punto de la memoria del que os hemos redirigido clicando en el link de sección siguiente: [3.6.10.2](#)

Figura 5.2: Proceso completo de creación del ícono pequeño de mercApp. El proceso involucra IA generativa y pequeñas ediciones en Gimp para retirar canal alpha, pintar el fondo de la delta y añadir una parte redonda con fondo blanco para que se vea en la barra de navegación negra. Existe un ícono con la delta blanca para barra de navegación negra que aquí no mostramos.

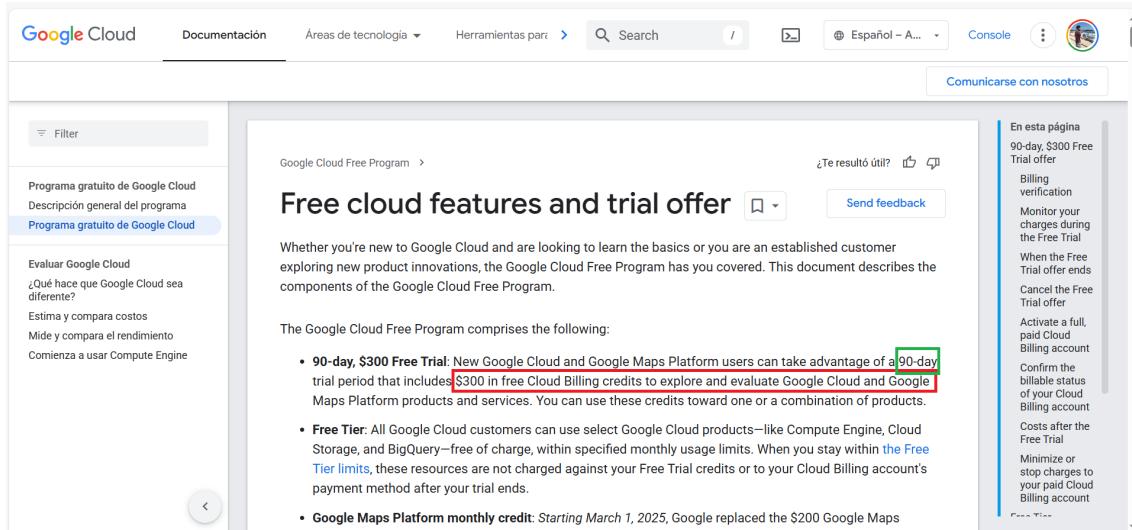


5.10. Google Cloud: descargar tickets digitales mediante cloud

NOTA: clica aquí [3.6.9.2](#) para regresar al texto que referenció este anexo.

Los pasos a seguir para obtener el ClientID necesario para habilitar usuarios a la descarga de tickets digitales es acceder a <https://cloud.google.com/> e introducir la información de pago (si ya tenemos pagos periódicos a Google por, por ejemplo, compra de almacenamiento a Google Drive no habrá que añadirlos). Fijémonos en lo que está disponible: 300 dólares de créditos gratuito durante tres meses. Como vemos en la figura [5.3](#).

Figura 5.3: Crear cuenta gratuita de google cloud



Si consultamos los servicios gratuitos en [esta página](#) tenemos muchos. De ellos los que pueden servirnos para nuestro caso particular podrían ser **App Engine** y **Cloud run**. Ambos tienen un free tier distinto. Después de extraer la información detallada de **Cloud run** y de **app engine** hay un claro ganador en cuanto a costes para nuestro caso de uso: el primero (ver figura [5.4](#)).

Figura 5.4: Dos servicios con free tier que nos pueden ser de utilidad para nuestra aplicación. El que a priori sería más económico sería *cloud run* porque asegura que el proceso escalará a cero cuando no esté en uso, y nos permite activarlo solamente con un evento web (cuando el usuario se acabe de registrar y pida acceso a los tickets)

App Engine	Cloud Run
<ul style="list-style-type: none"> • 28 hours per day of F1 instances • 9 hours per day of B1 instances • 1 GB of outbound data transfer per day <p>The Google Cloud Free Tier is available only for the Standard Environment.</p> <p>Learn more</p>	<p>No ofrece reducción de escala a cero (peligroso)</p> <p></p>
<p>Cloud Run</p> <ul style="list-style-type: none"> • 2 million requests per month • 360,000 GB-seconds of memory, 180,000 vCPU-seconds of compute time • 1 GB of outbound data transfer from North America per month <p>Learn more</p>	<p>Documentación de Cloud Run</p> <p>Ler la documentación del producto</p> <p></p> <p>Cloud Run es una plataforma de procesamiento administrada que te permite ejecutar contenedores que se pueden invocar a través de solicitudes o eventos. Cloud Run es una plataforma sin servidores, lo que significa que quita la complejidad de la administración de infraestructura, de modo que te puedas enfocar en lo que más importa: compilar aplicaciones excelentes. Obtén más información.</p> <hr/> <p>Servicios y trabajos: dos formas de ejecutar tu código</p> <p>Servicios de Cloud Run Se usan para ejecutar código que responde a solicitudes, eventos o funciones web.</p> <ul style="list-style-type: none"> • Trabajos de Cloud Run. Se usan para ejecutar código que realiza trabajo (un trabajo) y se cierra cuando el trabajo finaliza. <p></p> <hr/> <p>Precios de pago por uso para los servicios</p> <p>La reducción de escala a cero es atractiva por razones económicas, ya que se te cobran por la CPU y la memoria asignadas a una instancia con un nivel de detalle de 100 ms. Si no configuras un mínimo de instancias, no se te cobrará si tu servicio no se usa.</p> <p></p>

Cloud run requiere contenerizar una aplicación y subirla al cloud. Sin embargo, aunque podría funcionar para nuestro caso particular, nada gana el hacerlo desde el front-end en términos de costes, como finalmente se ha hecho.

5.11. Google cloud: cálculo de unidades de cuota

NOTA: Podéis regresar al origen de este anexo con un click aquí: [3.6.9.2](#)

En la siguiente página de [developers.google \[19\]](#) ([link](#)) podemos ver que cada una de las funciones que usamos en el script `scriptExtraccioBoto.js` usado para extraer los tickets de los correos electrónicos del Gmail de un usuario consumen ciertas unidades de cuota de la API de Google. Nosotros usamos `messages.attachments.get`, `users.messages.list` y `messages.attachments.get`, que tienen un uso de 5 unidades de cuota por cada llamada, cada uno. Por ello, cada ticket descargado para un usuario nos va a consumir un total de 15 unidades. Si ese mismo usuario se descarga 500 tickets digitales adjuntos de los correos en un segundo -asumiendo que la API fuese del orden de 100 veces más rápida de lo que lo es en realidad- supondrán 7500 unidades de cuota gastadas de golpe: esto ya sería suficiente para este proyecto. Si

el usuario, sin embargo, decide volver a autenticarse otra vez y quiere descargar de nuevo los tickets va a gastar las 15 000 unidades y va a excederse de la cuota si lo intenta por tercera vez, dándonos *userRateLimitExceeded*: Esto sí sería un aspecto que en una versión más definitiva deberíamos pulir (se puede añadir *batch processing*, para descargar por lotes si queremos aumentar el límite de tickets digitales descargables); pero para la versión que nos ocupa no vamos a hacerlo porque asumimos que dar capacidad de análisis a los últimos 500 tickets digitales de correo es más que suficiente. Asimismo, el límite de frecuencia global por proyecto son 1 200 000 unidades de cuota por minuto, lo cual nos daría la posibilidad de tener hasta 160 usuarios concurrentes en un minuto, suponiendo que cada uno de ellos hiciese como máximo una descarga de 500 tickets ¹ sin dar *rateLimitExceeded*. En la figura 5.5 siguiente podéis ver los límites de la página de desarrolladores de Google que enlazamos antes y en la figura 5.6 los gastos de cuotas por llamada a cada función usada.

Figura 5.5: Unidades de cuota de procesamiento que nuestra aplicación no puede exceder al hacer llamadas a gmailAPI sin incurrir en excesos [19].

Tipo de límite de uso	Límite	Motivo del exceso
Límite de frecuencia por proyecto	1,200,000 unidades de cuota por minuto	rateLimitExceeded
Límite de frecuencia por usuario	15,000 unidades de cuota por usuario por minuto	userRateLimitExceeded

Figura 5.6: La cantidad de unidades de cuota que consume una solicitud a la gmailAPI varía según el método al que se llama. En la siguiente tabla, se describe el uso exacto de las unidades de cuota por cada uno de los tres métodos de gmail API que utilizamos cuando un usuario descarga un ticket del correo. Concretamente, en este caso, un usuario gasta un total de 15 unidades de cuota (uc). [19].

Método	Unidades de cuota
<code>messages.get</code>	5
<code>messages.list</code>	5
<code>messages.attachments.get</code>	5



¹ $1\,200\,000 \text{ (uc/min)} / 7500 \text{ (uc/usuario)} = 160 \text{ usuarios/min}$

5.12. Correspondencia entre páginas del sistema de registro de NetFlix y el sistema de registro de mercApp

NOTA: Al terminar podéis regresar al apartado 3.6.3, que es donde proviene el link que enlaza a este anexo.

La siguiente tabla y las imágenes que vienen a continuación son un complemento al diagrama de enrutamiento de vistas de la aplicación mercApp (figura 3.23). Este enrutamiento corresponde en su lógica de redirecciones, al menos parcialmente, con la lógica utilizada por NetFlix en su aplicación web. Así las cosas, la correspondencia de la que hablamos en el título no es solo en su forma o aspecto.

Vista mercApp	Vista en NetFlix	Comparación vistas
pas1_Landing SignUp	/	ver 5.7
pas2A_infoBenvinguda.html	/signup/registration	ver 5.8
pas3_crearContrassenya	/signup/regform	ver 5.9
pas2B_introdueix contrassenya	/signup/password	ver 5.10
pas2C_login	/login	ver 5.11
pas4_concedirAccesGmail	signup/planform	ver 5.12
dashboard	App de streaming	ver 5.13

Cuadro 5.2: Correspondencia entre vistas del proyecto mercApp y rutas de navegación en NetFlix (estudio de algunas rutas se hizo en verano de 2024 y pueden haber cambiado, se recomienda ver las imágenes de comparación): es de la página de Netflix de la que se sacó la inspiración para hacer el enrutamiento de vistas del proyecto para captar usuarios fácilmente, como se puede ver en cada una de las figuras comparativas enlazadas en la tercera columna.

Ruta en Netflix	URL completa
/	https://www.netflix.com/
/signup/registration	https://www.netflix.com/signup/registration
/signup/regform	https://www.netflix.com/signup/regform
/signup/grantAccess	https://www.netflix.com/signup/grantAccess
/signup/password	https://www.netflix.com/signup/password
/login	https://www.netflix.com/login

Cuadro 5.3: Correspondencia entre rutas relativas y URLs completas en Netflix. ¡CUIDADO! Las páginas no siempre son visualizables: hay que iniciar un proceso de registro en muchos casos y/o entrar en modo incógnito si no queremos cerrar sesión en Netflix, en caso que tengamos un perfil. De ahí las imágenes comparativas adjuntas más abajo.

Figura 5.7: Vista de la landing page de mercApp (arriba) y de NetFlix (abajo)

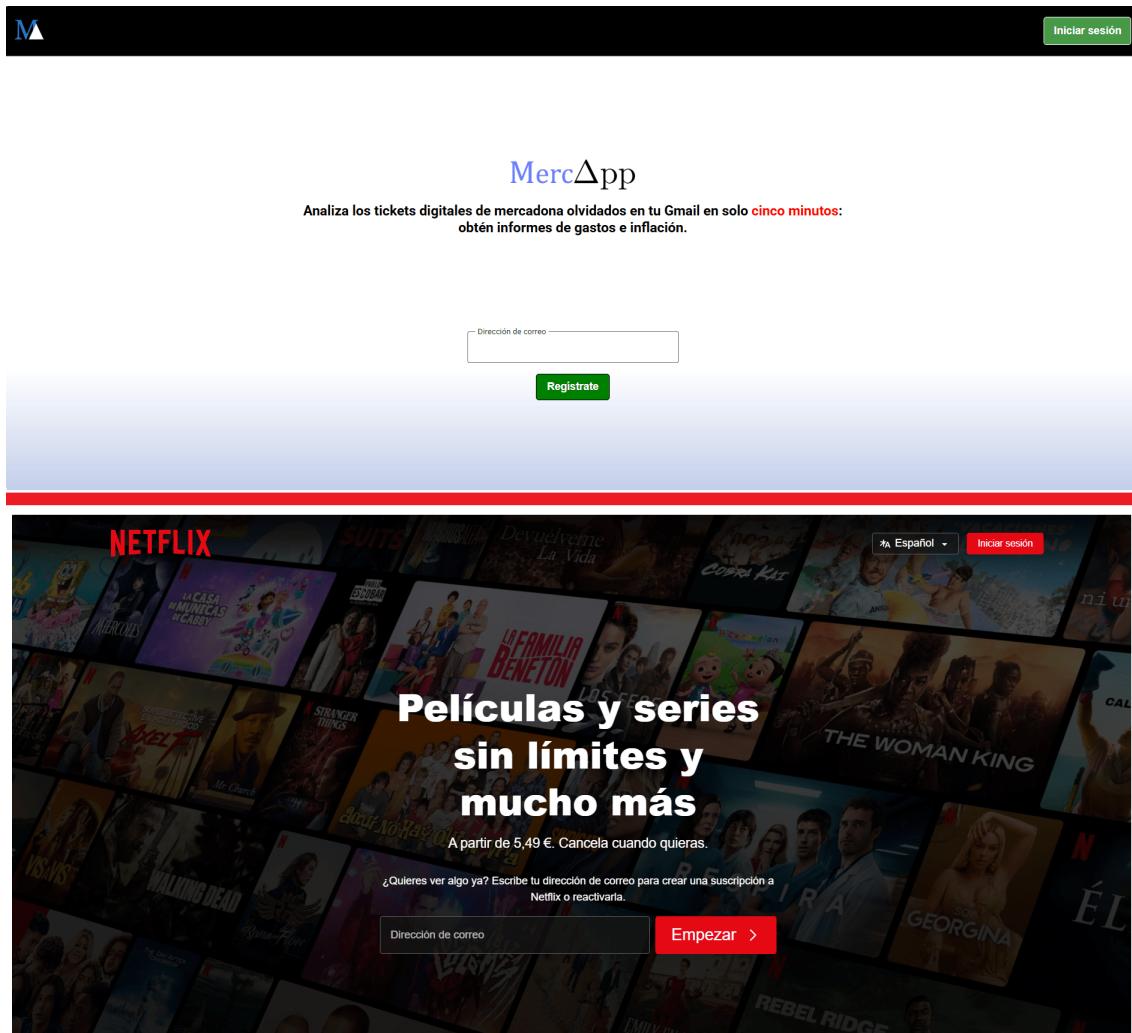


Figura 5.8: Pantalla de bienvenida al registro de mercApp `pas2A_infoBenvenida.html` (arriba) y página de Netflix homóloga con la misma función, con ruta `/signup/registration` (debajo)

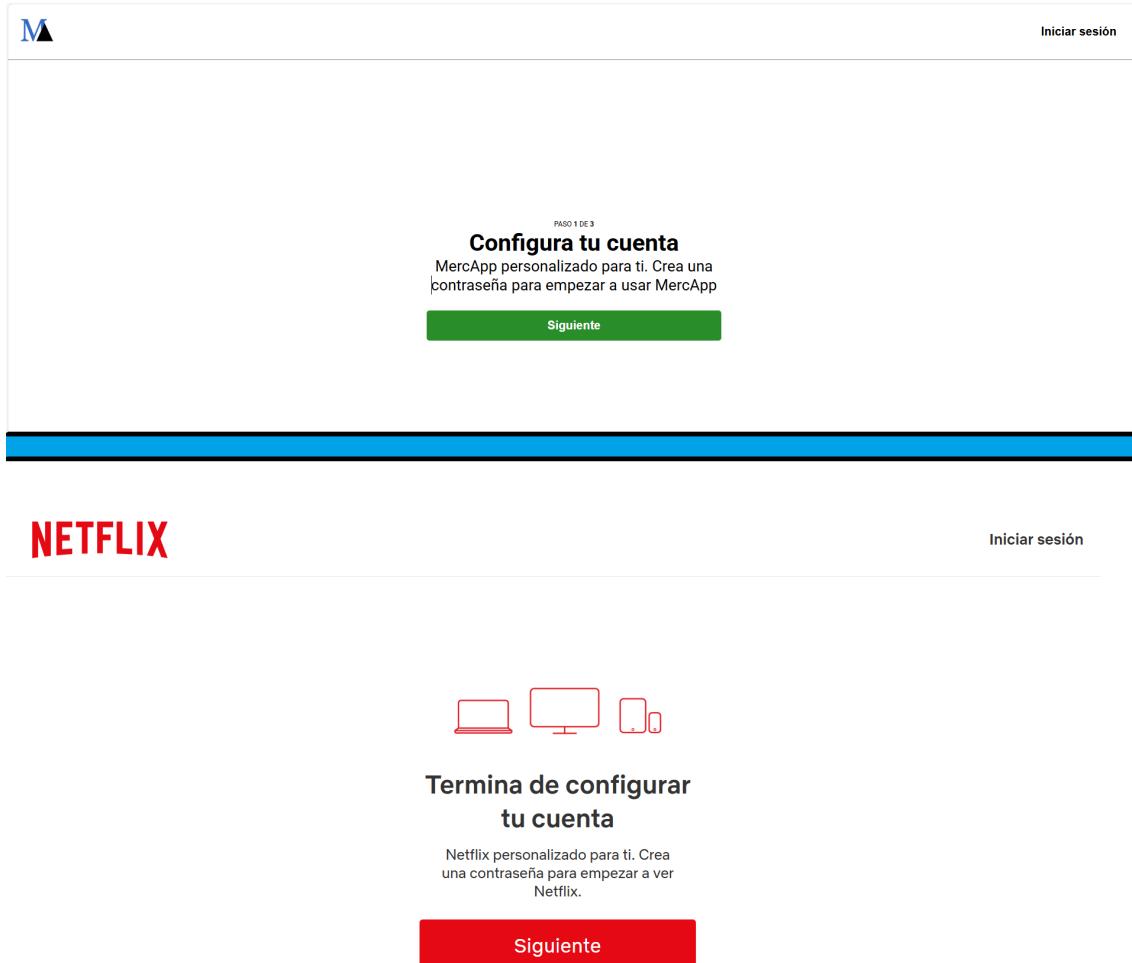


Figura 5.9: Formulario de creación de contraseña en mercApp pas3_crearContrassenya (arriba) y página equivalente en NetFlix con ruta /signup/regform (debajo)

The figure shows two side-by-side screenshots of password creation forms.

Top Form (mercApp):

- Paso 2 de 3**
- Crea tu contraseña!**
- ¡Ya casi hemos terminado!*
- Contraseña
- Siguiente**

Bottom Form (Netflix):

- NETFLIX** logo and **Iniciar sesión** link.
- Crea una contraseña para empezar la suscripción.**
- ¡Faltan solo algunos pasos!**
También odiamos el papeleo.
- Dirección de correo
- Añadir una contraseña
- No, no quiero ofertas especiales de Netflix por correo.
- Siguiente**

Figura 5.10: Pantalla para introducir manualmente la contraseña y completar el registro del usuario en mercApp **pas2B_introdueix contrassenya** (arriba) y página equivalente en NetFlix con ruta */signup/password* (debajo). En mercApp el correo se recupera del *localStorage*; en NetFlix probablemente utilizan un mecanismo similar.

The figure shows two side-by-side screenshots of password creation forms. The top form is from mercApp, featuring a large blue header with the letters 'M' and 'A'. The bottom form is from Netflix, with its red logo on the left and a blue header with the text 'Iniciar sesión' (Sign In).

mercApp (Top):

- Header:** M A
- Text:** ¡Te damos de nuevo la bienvenida!
- Subtext:** Escribe tu contraseña para acceder al usuario que empezaste a crear.
- Input:** Correo electrónico:
noacces@gmail.com
- Input:** Contraseña
- Button:** Siguiente
- Link:** Iniciar sesión

Netflix (Bottom):

- Header:** NETFLIX
- Text:** ¡Te damos de nuevo la bienvenida!
- Text:** Es fácil suscribirse a Netflix.
- Text:** Escribe tu contraseña para empezar a disfrutar.
- Input:** Dirección de correo
asd@gmail.com
- Input:** Escribir tu contraseña
- Text:** ¿Has olvidado tu contraseña?
- Button:** Siguiente
- Link:** Iniciar sesión

Figura 5.11: Pantalla de inicio de sesión con correo electrónico y contraseña en mercApp pas2C_login (arriba) y en NetFlix con ruta /login (debajo)

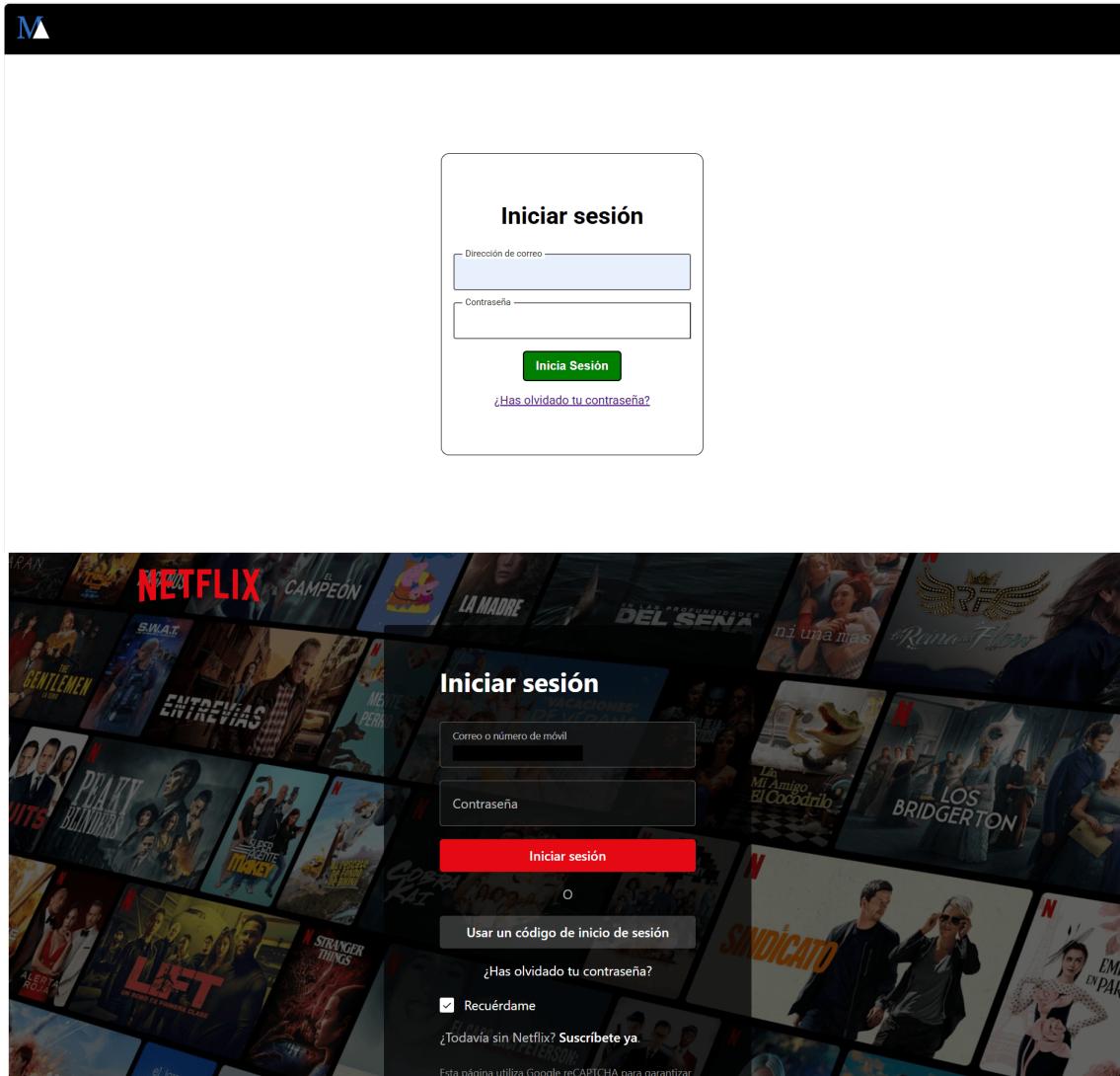


Figura 5.12: Pantalla **pas4_concedirAccesGmail** usada para conceder acceso a la cuenta de Google, descargar tickets y pedir su parseo (arriba) i página “equivalente” de NetFlix, que ellos utilizan para que el usuario haga pagos, con ruta *signup/planform* (debajo). NOTA: En mercApp se llega aquí cuando el usuario tiene permisos = 0 en el token de acceso.

Paso 1: Descarga tus tickets digitales

Habilita ventanas emergentes y luego autenticate en tu gmail en una cuenta con más de dos tickets digitales dentro de días distintos. Descarga hasta los 500 tickets más recientes automáticamente.

Paso 2: Mándanos tus tickets digitales

Selecciona todos los tickets digitales que se han descargado en la carpeta descargas de tu sistema en el paso anterior y adjúntalos (no cambies su nombre!).

has facilitado 7 tickets al sistema:

Paso 3: Inicia la extracción de datos.

Una vez conforme con los tickets subidos en el paso anterior, clica en el engranaje: entonces extrearemos el contenido de los tickets digitales en PDF y analizaremos sus datos. Serás redirigido a tu dashboard de visualización terminar.

© 2025 / Santiago Sánchez Sans [in](#)
Todos los derechos reservados

NETFLIX

Cerrar sesión

Elige el plan ideal para ti

Estándar con anuncios	Estándar	Premium
Prueba al mes 5,49 €	Prueba al mes 12,99 €	Prueba al mes 17,99 €
Ver todos los videos y audio Buena	Calidad de video y audio Buena	Calidad de video y audio Excepcional
Resoluciones 1080p (Full HD)	Resoluciones 4K (Ultra HD)	Resoluciones 4K (Ultra HD + HDR)
Dispositivos compatibles: Teléfono, ordenador, tabletas	Dispositivos compatibles: Teléfono, ordenador, tabletas	Dispositivos compatibles: Teléfono, ordenador, tabletas
Dispositivos de los que puedes verlos: Netflix en la web	Dispositivos de los que puedes verlos: Netflix en la web	Dispositivos de los que puedes verlos: Netflix en la web
2	2	6
Descarga en dispositivos	Descarga en dispositivos	Descarga en dispositivos
Anuncios	Sin anuncios	Sin anuncios
Mucha de lo que plasmas		

Mayores informaciones y plan con anuncios. Tuágina y sus términos y condiciones, incluye que Netflix te fija la fecha de renovación para la permanencia de anuncios y otras formas conformes con la Declaración de protección de datos.

Los términos y condiciones de Netflix y las políticas de privacidad están sujetos a cambios sin previo aviso. Netflix no se hace responsable de las modificaciones de las políticas de privacidad o de los términos y condiciones de Netflix. No todo el contenido es disponible en todas las resoluciones. Consulta los [Términos de uso](#) para obtener más información.

Para más información sobre los planes y precios de Netflix, consulta la [Página principal](#). Los precios de Netflix varían según el país y la tarifa. Puedes ver los precios actuales en la [Página principal](#).

Siguiente

Figura 5.13: Vista simplificada del **dashboard** tras iniciar sesión después de haber proporcionado tickets y haberlos parseado a mongoDB y ganar permisos a 1 o 2 (arriba) y la vista simplificada equivalente de la página con plenos derechos de acceso que el usuario de NetFlix ve cuando ha pagado y se ha logueado (debajo)



¡Hola Nombre de usuario!

MercApp ha encontrado **XXX** tickets digitales de mercadona en tu correo electrónico, desde **dd/mm/aa** hasta **DD/MM/AA**:
Resumen de todas estas compras a continuación:

