

Creación de un dashboard para usuarios del ticket digital de Mercadona con visualización de la evolución temporal de precios en productos habitualmente adquiridos, costes de compras por intervalos temporales y gastos por áreas de producto.

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, que esperemos permita consumar un cambio profesional del ámbito de la Psicología a la informática.

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 insta al lector a considerarla el resultado del esfuerzo y trabajo constante a lo largo de los meses de marzo a mayo de 2025 (en paralelo con las prácticas a tiempo completo en Lâberit) y valorarla con su idiosincrasia -incluyendo sus imperfecciones-.

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 durante registro de usuario	9
2.5.2. Camino durante inicio de sesión	9
3. Desarrollo del proyecto	10
3.1. GitHub del proyecto	10
3.2. Entornos de desarrollo	10
3.3. Despliegue	11
3.4. Desarrollo back-end (Spring Boot)	12
3.4.1. Estructura de la aplicación	12
3.4.1.1. src/main/java: las clases del proyecto	13
3.4.1.2. src/main/resources: archivos de configuración	14
3.4.1.3. pom.xml: dependencias de maven	14
3.4.2. Autenticación y Autorización	14
3.4.2.1. método utilizado: JWT	14
3.4.2.2. ¿Qué compone un JWT?	16
3.4.2.3. Implementación de JWT en java SpringBoot	16
3.4.2.4. Enviar por primera vez el Access Token hacia el front-end (registro)	19
3.4.2.5. 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.	21
3.4.3. Validación de datos (End-points back)	26

3.4.4. Hasheado de contraseñas	29
3.5. Desarrollo back-end (microservicio con Python)	30
3.5.1. Contenerización	30
3.5.2. estructura de la aplicación	32
3.5.3. Solicitud de subida y parseo de datos	32
3.5.3.1. PARTE 1: FastAPI recibe la POST request con los tickets adjuntos desde el cliente	32
3.5.3.2. PARTE 2: FastAPI parsea todos los tickets con un algoritmo de extracción	33
3.5.3.3. PARTE 3: FastAPI manda los tickets parseados a MongoDB	34
3.5.3.4. PARTE 4: FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot: ahí se actualiza la variable de permisos en la bbdd mySql, y se recibe como respuesta un nuevo token de acceso fresco con los permisos actualizados a 1	34
3.5.3.5. PARTE 5: FastAPI manda la RESPONSE con el nuevo token de acceso con permisos a 1 al front-end, a la página pas4 (el pas4 ya redirigirá automáticamente al dashboard)	34
3.5.4. Gestión de solicitudes: token de acceso	34
3.5.5. Validación de datos (end-point entrada PDFs)	35
3.6. Desarrollo del front-end	36
3.6.1. Contenerización	36
3.6.2. Estructura de la aplicación	36
3.6.3. Enrutamiento de vistas	36
3.6.4. Manejar vistas en función de Autenticación y autorización	40
3.6.4.1. Protegiendo vistas “privadas” ante usuarios no “logueados”: cuando el token ha expirado	41
3.6.4.2. Protegiendo vistas “públicas” para usuarios “logueados”: cuando el token no ha expirado	43
3.6.4.3. Protegiendo vistas “privadas” ante usuarios “logueados”: cuando el token no ha expirado.	44
3.6.4.4. Salir voluntariamente de las páginas privadas: botón “cerrar sesión”	46
3.6.5. Recibir el Access Token desde el back-end	46
3.6.6. Validación de datos (Formularios entrada)	49
3.6.7. Diseño (UX/UI): páginas publicas	49
3.6.8. Diseño (UX/UI): páginas privadas	50
3.6.8.1. Diseño del dashboard	50
3.6.8.2. Diseño del “pas4”	56
3.6.9. Arquitectura (JavaScript): páginas privadas	60

3.6.9.1. Arquitectura del dashboard	60
3.6.9.2. Arquitectura pas4	61
3.6.10. Arquitectura (JavaScript): páginas públicas	63
3.6.11. Diseño de iconos	63
3.6.11.1. Áreas de producto	63
3.6.11.2. Icono mercApp pequeño	65
3.7. Desarrollo Cloud: configuración google API client	66
4. Evaluación y Conclusiones Finales	74
5. ANEXO	75
5.1. Flujo de trabajo habitual en git	75
5.2. Diferencias de seguridad: JWT vs SESSID en cookies seguras	76
5.3. Clases para crear y verificar JWTs	77
5.3.1. Clase JWT	77
5.3.2. Clase Refresh Token	77
5.3.3. Clase Access Token	77
5.4. Clases de seguridad	78
5.4.1. Clase ConfiguracioSeguretat.java	78
5.4.2. Controlador con restricciones aplicadas	79
5.5. Diagrama réplica netflix	80
5.6. Diagrama enrutamiento mercApp	81
5.7. Aspectos ventajosos de separar front-end y back-end (SoC)	81
5.8. Captura proyecto desarrollo interfaces	82
5.9. Creación del icono mercapp pequeño: IA con Gimp	83
7. Bibliografía	83
5.10. Google Cloud: descargar tickets digitales mediante cloud	85
5.11. Google cloud: cálculo de unidades de cuota	87

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 comprados 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 como framework (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 ejercitarno puede permitir iniciar una carrera enfocada en una tecnología que no se prevé que desfallezca en un futuro cercano.

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 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 clasificar.

².

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.

²Para ello, dado que no tenemos categorizados todos los productos de Mercadona ni podríamos hacerlo por falta de una lista exhaustiva y de 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.

REQUISITO D³: 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 E⁴: 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](#). Ambas 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 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.

³Requisito añadido después de la presentación del proyecto.

⁴Requisito añadido después de la presentación del proyecto.

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

- Back-end con Java (springboot para el login y la autenticación de usuarios: con este framework guardaremos datos en la BBDD mySQL).
- fastAPI parseará el contenido de los mismos: con sklearn, numpy y spacy que luego se podrán pasar a la BBDD mongoDB mediante Spring Boot.

*Spring Boot se encargará de la **persistencia** de los datos de los tickets, hacia mongoDB y de su lectura. Python solamente extraerá o parseará el contenido de los tickets mandando sus datos hacia el back-end de spring boot directamente.*

2.3.3. Cloud: Google API Client

- **Google API Client** nos permitirá acceder a los tickets de los usuarios y permitirle al usuario descargarlos a su ordenador mediante 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 estamos bien versados en ello.

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 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: en cuyo caso se borrarán los datos del localStorage.

2.4. Secuenciación de tareas

El desarrollo del proyecto ha empezado entorno el **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 y 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 3.15 principalmente) y todo lo redactado de la parte de SpringBoot del proyecto (ver apartado 3.4).

Durante el horario de las prácticas de Lāberit (iniciadas el 10 de marzo) hasta el viernes 11 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). También se ha creado el enlace del back-end con java hasta el back-end de fastAPI donde haremos la explotación de tickets.

Fuera del horario de prácticas desde el **21 de abril** hasta el **24 de mayo** (día previo a la entrega de la memoria) se programarán 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 saca datos de una API de Spring Boot que a su vez extraerá los datos de la API de FastAPI, que a su vez se conectará con MongoDB.

2.5. Diagrama de sistemas de la aplicación



Figura 2.1: Diagrama de sistemas de la aplicación mercApp. Tenemos el back-end principal (Spring Boot) donde se mandan a guardar y leer tanto los usuarios (mySql) como los tickets (MongoDB). El back-end secundario, con fastAPI, parsea los tickets y los pasa a formato estructurado JSON para hacer que Spring Boot lo persista en MongoDB. Google API Client permite extracción de tickets del gmail del usuario. El front-end con "Vanilla" JavaScript muestra las vistas al usuario en función del token de acceso.

2.5.1. Camino durante registro de usuario



Figura 2.2: Simplificación de los caminos activados en el diagrama de sistemas durante el registro de un usuario, hasta que este tiene acceso al dashboard de visualización con éxito en todas las partes del proceso: a) Usuario pone correo y contraseña. b) guardamos correo y hash contraseña. c) Mandamos vista al usuario para que nos dé acceso a tickets digitales junto con token de acceso con permisos = 0. d) usuario se autentica con **google API client**. e) tickets regresan al navegador. f) fastAPI recoge los tickets y los parsea a formato JSON estructurado. g) fastAPI delega en SpringBoot la persistencia de datos. h) la persistencia de los tickets se hace en MongoDB. i) Spring Boot manda token de acceso con permisos = 1 para que frontend muestre el análisis de los tikets en el dashboard.

2.5.2. Camino durante inicio de sesión

TO DO

Desarrollo del proyecto

3.1. GitHub del proyecto

Para desarrollar este proyecto se ha trabajado con GitHub y git. Dado que no ha habido trabajo en equipo no se han utilizado pull requests a la rama main sino simplemente se ha seguido la estrategia de crear ramas de característica y, una vez son satisfactorias, hacer un merge en la rama main en local.

Un flujo de trabajo habitual es mediante ramas de característica (puede verse anexo 5.1). También puede verse el GitHub del proyecto a continuación. Dentro del readme del proyecto encontraréis instrucciones para su descarga y clonado.

Las instrucciones para correr los componentes del proyecto en sus respectivos entornos de desarrollo están en el apartado 3.2. El despliegue de la aplicación mediante contenedores se encuentra explicado en 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**: se han utilizado extensiones necesarias para correr el proyecto que permiten sacar provecho de Lombok sin las cuales correr el proyecto en IntelliJ fallará¹.

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 con fast-api se ha utilizado el servidor embedido en el propio framework, expuesto en el puerto **8000**.

Para la base de datos mySQL se ha utilizado el editor mySQL workbench que corre en el puerto **3306**.

¹Para 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!

3.3. Despliegue

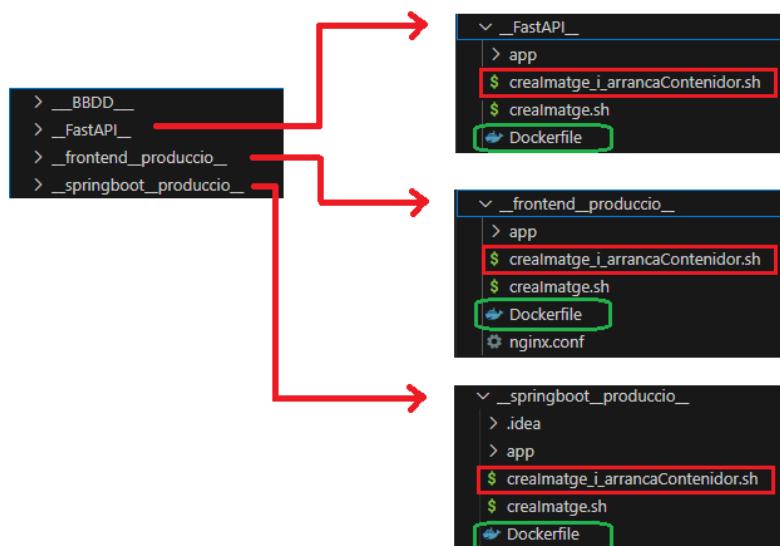
Los componentes del apartado 3.2 de entornos desarrollo se han *dockerizado* de modo que cada uno de ellos sea un microservicio, a excepción (por ahora), de las bases de datos. Hemos sido cuidadosos de que los microservicios en Docker corran en los mismos puertos que los servidores embedidos con los 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*³ 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 git bash en Windows:

```
bash creaImatge_i_arrancaContenedor.sh
```

Así, no tenemos que preocuparnos de parar el contenedor, borrarlo y luego crear la imagen (ver imagen 3.1).

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.



³La explicación pormenorizada de sus comandos, para el caso de fastAPI por ejemplo, puede verse en el apartado 3.5.1

3.4. Desarrollo back-end (Spring Boot)

3.4.1. Estructura de la aplicación

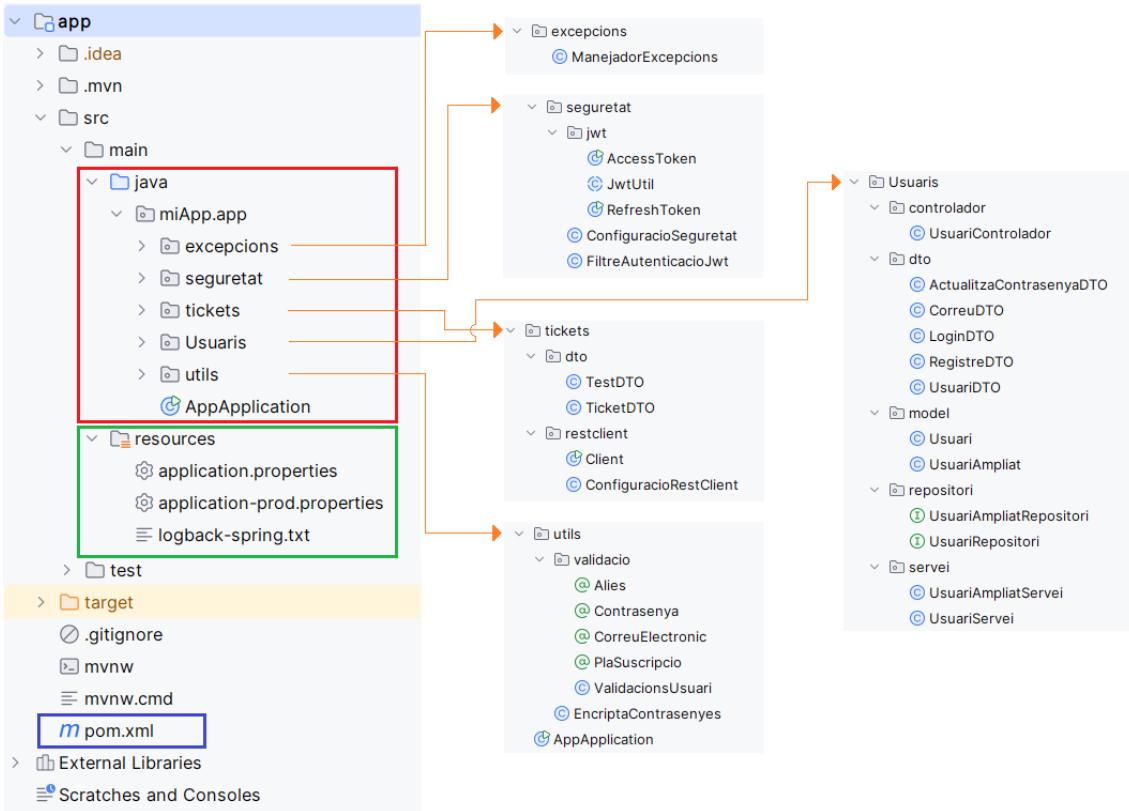
La parte de mercApp programada con Java (Spring Boot) se puede ver en el repositorio de GitHub del proyecto abriendo esta carpeta: [aquí](#) (eecomendamos al lector que abra y corra el proyecto con IntelliJ abriendo la carpeta mencionada del GitHub, pero en local). Se podrá ver entonces *tres* rutas importantes (que son las tienen todos los proyectos Spring Boot):

- **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`. 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) y `logback-spring.xml` que, aunque no está definido por defecto en una aplicación Spring Boot, si lo añadimos lo que hace es que los logs del proyecto se guarden en `logs/spring.log` en lugar de salir a terminal⁴. Aquí aparece como un `.txt` en lugar de un `.xml` porque por ahora no queremos que se guarden los errores.
- **pom.xml** → Es un arhivo donde encontramos el árbol de dependencias de Maven⁵ (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.

⁴La ruta se crea automáticamente

⁵Maven 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.

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})



3.4.1.1. 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 [3.4.3](#).

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 [3.4.2.3](#); 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 [3.4.2.5](#).

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 [3.4.4](#).

TO DO EXPLICAR	usuari, repository, service, controller de
TO DO EXPLICAR	classes per a fer la unio amb el backend de fast API mitjançant restClient.

3.4.1.2. src/main/resources: archivos de configuración

TO DO EXPLICAR application.properties

3.4.1.3. pom.xml: dependencias de maven

TO DO EXPICAR pom.xml: parlar de la versio i link a la pagina on la versio.

3.4.2. Autenticación y Autorización

3.4.2.1. 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 servidor⁶. 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

⁶El token está firmado, pero no cifrado: todo el mundo puede ver su contenido.

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 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 un alcance idéntico o más limitado -es decir, duración más corta- [6]
- 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, en caso que sea robado, el servidor rechazará la petición no permitiendo acceso a recursos [7]).

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 la **desventaja** más evidente que tiene JWT, es, en nuestra opinión, su com-

plejidad⁷, pues para tener un buen equilibrio entre facilidad de uso y seguridad es necesario almacenar los tokens en el cliente para conseguir que uno se renueve (el token de acceso y el de refresco, como comentamos):

- Caducidad de tokens irrevocable
- Renovación de tokens de acceso con uno de refresco

3.4.2.2. ¿Qué compone un JWT?

El JWT se compone de tres partes separadas por *puntos*: **el header**, **el payload** y **la firma**. En la página <https://jwt.io/>, como veremos después, se puede ver si los tokens son válidos, observar el contenido de su payload, etc. [8]. 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 firma:** 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 3.3 que veremos después.

3.4.2.3. Implementación de JWT en java SpringBoot

Para poder implementarlo añadimos la dependencia **jjwt** en **pom.xml** que es la que nos permite definirlo.

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

⁷Se puede ver una tabla de diferencias más en profundidad, especialmente en materia de seguridad en el anexo 5.2)

En el proyecto 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**. A continuación podéis, de estas tres, la más importante (la 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):

```
@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 que serán necesarios para autorizar (`idUsuari`) y autenticar (`permisos`), podemos generar un token de acceso en “accesJWT” que es el que usamos en la aplicación para permitir acceder a los endpoints o no:

```
AccessToken accessToken = new AccessToken();
String accesJWT = accessToken.genera(
```

```

    "santo@gmail.com", //campoSub
    2, //idUsuario
    1 //permisos
);

```

Si vemos la figura 3.3 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 de la parte derecha de la imagen, siendo la payload la más importante:

Figura 3.3: 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

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MS
wiaWRVc3VhcmkiOjIsInN1YiI6InNhbnRvQGdtYW
WlsLmNvbSIsImlhCI6MTc0MzE1MjM1MCwiZXhw
IjoxNzQzMTUyOTUwfQ.1xC6NKkuG99zrxLcrgxf
jRRK18NgYbUrirFgPMEcUC0
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE	
{ "alg": "HS256" }	
PAYLOAD: DATA	
{ "permisos": 1, "idUsuario": 2, "sub": "santo@gmail.com", "iat": 1743152350, "exp": 1743152950 }	
VERIFY SIGNATURE	
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256- 88f7d99bb6c34) □ secret base64 encoded	



```
{
  "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 [9]. 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 o en el GitHub del proyecto ([link](#))⁸⁹.

3.4.2.4. 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 electrónico¹⁰ mediante una petición POST al controlador de endpoint `api/registraUsuari` de nuestro back-end de springboot. 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[10] 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 seguridad¹¹: si todo va bien deberemos obtener la figura 3.4:

⁸Se recomienda encarecidamente al lector optar por esta última opción

⁹Al 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.

¹⁰el correo electrónico lo teníamos guardado en el localStorage de la página de registro inicial.

¹¹Mínimo 8 caracteres, una minúscula y una mayúscula y sin caracteres peligrosos.

HTTP <http://localhost:8080/api/registraUsuari>

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

Params Authorization Headers (9) **Body** ● Scripts Settings

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

```
1 {"correoElectronico": "nuevoUsuario@gmail.com", "contrasenya": "12345678Mm"}
```

Body Cookies (1) Headers (14) Test Results | ⏪

{ } JSON ▾ ▷ Preview ⚡ Visualize | ▾

```
1 {
2   "existiaUsuari": false,
3   "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MCwiaWRVc3VhcmkiOjQsInN1YiI6I
4     EU8EXuvV2_WbkR1-iymJOVaeIn5DVcw1Kn6e08r0kEc",
5   "usuariShaRegistrat": true
}
```

Figura 3.4: Creación de un nuevo usuario llamando con una solicitud POST al endpoint “api/registraUsuari” cuando las validaciones del objeto RegistreDTO del back-end lo permite. 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).

Tenemos otro endpoint que también expide tokens de acceso, mucho más habitualmente que el anterior: es el endpoint que se consume cuando iniciamos sesión, en `pas2C_login.html`: el endpoint ubicado en la URI¹² `/api/login`. Si intentamos iniciar sesión con un usuario ya existente en la tabla de usuarios obtendremos algo como esto:

¹²Uniform Resource Identifier

```

POST http://localhost:8080/api/login
Body
1 {"correuElectronic": "acces@gmail.com", "contrasenya": "12345678Mm_"}
2
3 {
4     "usuari": {
5         "alias": "blackcub3s",
6         "permisos": 1,
7         "idUsuari": 2
8     },
9     "existeixUsuari": true,
10    "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNvcyI6MSwiוארVCv3VhcmkiOjIsInN1YiI6ImFjY2VzQGdtYWlsLmNvMqcDhr3cjnaAetzBuKnn_DpbYsLkdtZtYF8VUcxeZbM",
11    "teAccesArecursos": true,
12    "contrasenyaCorrecta": true
13 }

```

Figura 3.5: 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.

POTSER POSAR LES FUNCIONS DEL CONTROLADOR I DEL SERVICE QUE HO FAN ENLLAÇANT CODIS DE GITHUB.

La parte de recepción del token en el front-end y de su manejo podéis encontrarla en el apartado [3.6.5](#)

3.4.2.5. 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 [3.4.2.3](#) 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 de endpoints con JWT.

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 acceder al dashboard de visualización (a través de la header “Authorization”).

POSA EL CAS REAL QUAN L'HAGIS PROGRAMAT QUE AQUEST ENDPOINT ENCARA NO EXISTEIX: La solicitud HTTP con el susodicho token se hace via GET a ('usuaris/id/tiketsVERIFICARSIEEXISTEIX') (ver sub-

sección 3.4.1, en ControladorUsuari.java).

En javascript puro, desde el cliente, esta solicitud HTTP la pondríamos conseguir de la función fetch(), poniéndole uno de los pares clave valor con el inicio “Bearer” (por convenio) tal que así:

```
fetch("http://localhost:8080/usuaris/{id}/endpointTikets", {
    method: "GET",
    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 de la variable que haya emanado de la base de datos al conceder el token mediante **idUsuari** para el caso de la **Autenticación** -ver [línea github](#)- , y de la variable **permisos** del model de la @Entity class **Usuari** - ver [línea github](#)- para el caso de la **autorización**). 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 Autenticacion** dentro de **FiltreAutenticacio.java**
- **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 *FiltreAuntenticacioJwt.java* extiende de OncePerRequestFilter [11], 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 en [github](#) del proyecto [FiltreAuntenticacioJwt.java](#)

Lo primero que hay que tener en cuenta al diseñar esta clase es que tenemos que hacer una inyección de dependencias: debemos incluir la clase que hemos diseñado AccessToken para implementar el token de acceso. Lo haremos simplemente incluyéndola en el constructor como un parámetro.

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.

0 en la variable permisos de la tabla usuaris de mysql se asigna cuando un usuario ya se ha registrado dando correo y contraseña, pero no ha dado acceso a tickets digitales todavía; 1 en la variable permisos se da cuando el usuario en cuestión ya tiene acceso a la consulta del dashboard de la aplicación (ya se ha registrado y, además, concedido acceso a tickets digitales); y 2 se da cuando el usuario en cuestión es superusuario y tiene acceso a consultar todos los recursos de la aplicación, tickets de los demás usuarios, etc. ([ver rango](#)). En definitiva, lo que estamos mencionando (por si el lector se imprimió la memoria y no tiene acceso directo al link de GitHub) las líneas relevantes del código que permiten esa asignación son estas:

```
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 SpringBoot. 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)¹³.

```
UsernamePasswordAuthenticationToken authentication =
new UsernamePasswordAuthenticationToken(
    idUsuari,
    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. La tres cosas que hay que hacer en la clase `ConfiguracioSeguretat.java` para llevar a término las mencionadas configuraciones son:

¹³Cuidado! 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.

- 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](#))¹⁴.
- 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 administrador (*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 no está hecho todavía. Ahora mismo cualquier usuario con roles “USER” (*permisos = 1*) puede cambiar la contraseña de cualquier 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` donde le permitimos a ese controlador en específico afinar quien puede acceder a él. Con esta anotación, pasándole los parámetros adecuados, conseguiremos restringir, si no se es ADMIN, que un solo usuario con un solo *idUsuari* sea el que pueda cambiar la contraseña (concretamente, este usuario será el del *principal*¹⁵, 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`

¹⁴JWT 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.

¹⁵El principal era el *idUsuari* que pasamos al crear el objeto *UsernamePasswordAuthentificacionToken* dentro del paso1 en *FiltreAutenticacio.java*.

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](#)

3.4.3. Validación de datos (End-points back)

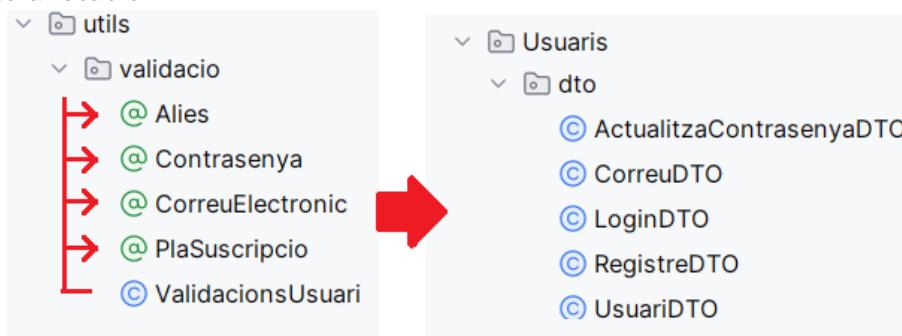
NOTA: Los datos validados en el back-end siguen las mismas expresiones regulares y restricciones que las validaciones hechas en el front-end (ver sección [3.6.6](#)).

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`¹⁶ pero se produce parte de validación a través de `Usuaris → dto`. Una primera aproximación o esquema a entender como se interrelacionan estos dos tipos de clases está en la figura [3.6](#).

Figura 3.6: **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 al endpoint que recibe la llamada del front-end para el inicio de sesión, es decir, el endpoint “[api/login](#)” (muy importante ver esta línea) del archivo `ControladorUsuari.java` de Spring Boot.

¹⁶podéis ver la imagen completa de la estructura del proyecto SpringBoot si queréis, de nuevo, en [3.2](#)

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

```
{"correoElectrónico" : "acces@gmail.com", "contraseña" : "12345678Mm\_" }
```

Según los términos esperados las clases utilizadas y activadas para validar estos datos de entrada serán las de la figura 3.7:

Figura 3.7: 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 3.8 podréis ver exactamente qué es lo que está pasando para la llamada a este endpoint. Podréis ver en esencia como existe una clase LoginDTO¹⁷ que permite hacer la validación de datos cuando un usuario inicia sesión, a través de anotaciones personalizadas (@CorreuElectronic y @contraseña) 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):

¹⁷Los 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.

Figura 3.8: 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 **@Contrasenya** 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



Podría parecer complejo hacer lo que hemos hecho en la figura 3.8, 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*.

3.4.4. 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[12] y se ha hecho una clase `EncriptaContrasesnyes.java` que ha permitido envolver con nombres más pedagógicos las funciones de bcrypt que han sido usadas (ver figura 3.9).

Figura 3.9: Clase `EncriptaContrasesnyes.java`, utilizada para encriptar contraseñas y comparar hash encriptados.

```
public class EncriptaContrasesnyes {  
    private final BCryptPasswordEncoder passwordEncoder = new BCryptPasswordEncoder();  
  
    //PRE: una contrasenya.  
    //POST: el hash de la contrasenya.  
    public String hashejaContrasesnya(String contrasenya) {  
        return passwordEncoder.encode(contrasenya);  
    }  
  
    //PRE: una contrasenya plana i una contrasenya hashejada  
    //POST: retorna true si la contrasenya plana, un cop hashejada coincideix amb la contrasenyaHash.  
    public boolean verificaContrasesnya(String contrasenyaPlana, String contrasenyaHash) {  
        return passwordEncoder.matches(contrasenyaPlana, contrasenyaHash);  
    }  
}
```

Específicamente, cada vez que un usuario inicie sesión o se registre, se utilizarán una u otra función de la clase `EncriptaContrasesnyes.java`. Al instanciar un objeto de la misma, se instanciará también un objeto BCryptPasswordEncoder y se usarán las funciones de librería `encode()`¹⁸ y `matches()`¹⁹, respectivamente (convenientemente guardadas con un nombre más agradable como vemos en la figura 3.9).

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

¡Es interesante hacer notar que una misma contraseña encriptada varias veces por la función `encode()` (envuelta en `hashejaContrasesnya()`) produce hash distintos! Es decir, el hashing no solo impide ver las contraseñas de los usuarios, sino que también impide ver si dos usuarios tienen la misma contraseña. Esto también hace que 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 nos vemos obligados a usar el metodo `matches()`.

¹⁸Para obtener el hash de una contraseña creada.

¹⁹Para verificar si una contraseña plana es coincidente con el hash que se guardó en base de datos en el momento del registro.

3.5. Desarrollo back-end (microservicio con 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`²⁰ descargada del registry de docker (ver nota sobre instalación de docker²¹).

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 3.10):

Figura 3.10: 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	8cffea9ac6	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 ...*), debemos usar esta imagen para derivar de ella un contenedor (es decir, crear una instancia de esa imagen). Crearemos el contenedor de nombre *contFastApi* a partir de la imagen anterior y le pediremos que exponga el puerto 8000²² -puerto con el que se comunica fastAPI en el interior del contenedor- y haremos que exponga ese puerto también al exterior del contenedor²³. 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

²⁰Las 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.

²¹Instalar 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.

²²Hacer referencia al 8000 que va a la **derecha** de los dos puntos.

²³Es el 8000 emplazado a la **izquierda** de los dos puntos.

usar las flags `-a`²⁴ e `-i`²⁵:

```
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 3.11 siguiente lo que puede hacer nuestro contenedor con esta configuración y como se construye, de forma más pormenorizada:

Figura 3.11: definición de un endpoint de fastAPI 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 contenedores que se creen con la misma fuera de esos contenedores (B) rectángulo rojo). También podemos ver la configuración de que fastAPI correrá en el puerto 8000 dentro del contenedor instanciado (B) rectángulo verde). Finalmente, podemos ver también el comando que crea finalmente contenedor desde la imagen, pidiéndole que que escuche en el puerto 8000 de la red interna del contenedor (C) rectángulo verde) y lo de ahí lo saque al puerto 8000 (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 (D), rectángulo naranja).

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

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

Permite que la app esté accesible desde fuera el contenedor

C) \$ creatimatge_1_arancaContenidor.sh
11 #creo contenedor contFastApi des de la imatge back-end-fastapi recent creada.
12 #redireccio port 8000 (dreta dels dos punts) al port XXXX (esquerra dels dos punts)
13 docker create -p 8000:8000 --name contFastApi back-end-fastapi
14 docker start contFastApi

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

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 `creatimatge_1_arancaContenidor.sh`, que incluye todos los comandos necesarios (inclusive aquellos para parar y destruir contenedores antiguos) ubicado aquí (ver figura)

²⁴Simplemente veremos lo que imprima la terminal

²⁵Nos permitiría interaccionar con el programa a través de la terminal si fuese necesario

3.5.2. estructura de la aplicación

TO DO

3.5.3. Solicitud de subida y parseo de datos

NOTA: la petición *POST* con *JavaScript* desde el front-end hacia el endpoint de subida de datos se mostró en la PARTE 2 del apartado anterior [3.6.9.2](#), sobre front-end

Al recibir el servidor de FastAPI una solicitud POST con los PDFs adjuntos y el Token de acceso con permisos a 0 en el endpoint /api/subir-tickets-pdf vamos a empezar la extracción de datos de los tickets. A continuación mostramos exhaustivamente los 5 pasos o partes, que deberán ejecutarse con éxito todas y cada una de ellas, y de forma SECUENCIAL, para que el proceso funcione correctamente y devuelva una RESPONSE con código de éxito:

- **PARTE 1:** FastAPI recibe la POST **REQUEST** con los tickets adjuntos desde el cliente junto con el token con permisos a 0 [\[3.5.3.1\]](#)²⁶.
- **PARTE 2:** FastAPI delega en el service de Python (PyPDF2) el parseo de todos los tickets con un algoritmo de extracción [\[3.5.3.2\]](#).
- **PARTE 3:** FastAPI manda los tickets parseados a MongoDB [\[3.5.3.3\]](#).
- **PARTE 4:** FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot: ahí se actualiza la variable de permisos en la bbdd mySql, y se recibe como respuesta un nuevo token de acceso fresco con los permisos actualizados a 1 [\[3.5.3.4\]](#).
- **PARTE 5:** FastAPI manda la **RESPONSE** con el nuevo token de acceso con permisos a 1 al front-end, a la página pas4 (el pas4 ya redirigirá automáticamente al dashboard)[\[3.5.3.5\]](#).

Después de todo esto, al recibir en el front-end el nuevo token de acceso con permisos a 1, el pas4 del front-end redirigirá automáticamente al dashboard sin que tengamos que hacer nada más (vuélvase a ver la lógica de redirecciones en la figura [3.18](#), que es la que nos lo gestionará automáticamente)

3.5.3.1. PARTE 1: FastAPI recibe la POST request con los tickets adjuntos desde el cliente

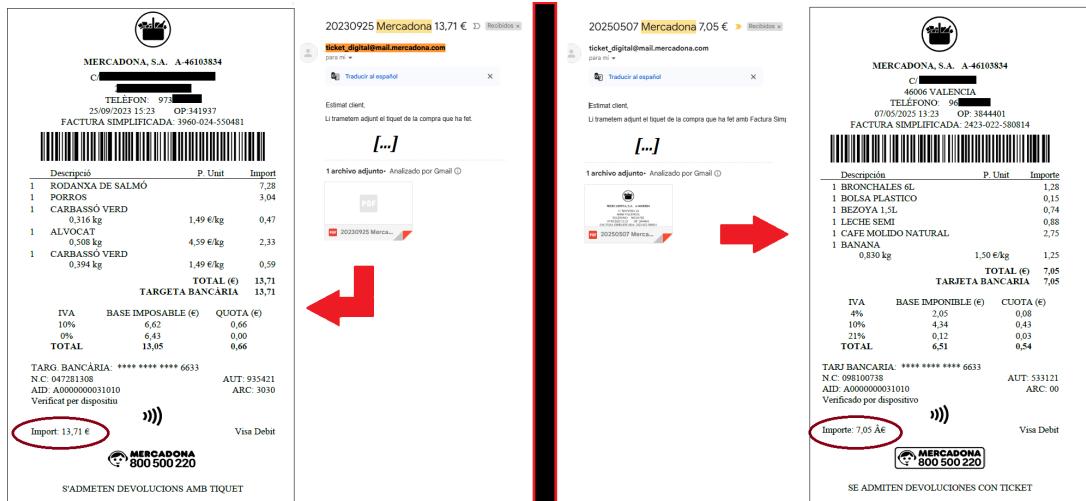
TO DO

²⁶0 es el caso de uso más frecuente, pero también podría ser 2 si fuese admin

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

El correo electrónico más antiguo y más nuevo prácticamente no difieren, con lo cual se puede reaprovechar un mismo algoritmo para todos los tickets como podemos ver en la imagen 3.12:

Figura 3.12: A la **izquierda**: la primera compra en Mercadona hecha con ticket digital por mi parte: en un supermercado en Catalunya; a la **derecha** la última compra que se ha hecho por mi parte: 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. La dirección desde la que se mandan los tickets digitales no ha cambiado en dos años y el formato general del ticket es exactamente igual a términos de parseo: Hay que prestar atención a posibles problemas con el unicode (círculo marrón).



CONTINUAR TO DO TO DO TO DO TO DO TO DO

- 3.5.3.3. PARTE 3: FastAPI manda los tickets parseados a MongoDB
- 3.5.3.4. PARTE 4: FastAPI manda el token de acceso (con permisos a 0) hacia Spring Boot: ahí se actualiza la variable de permisos en la bbdd mySql, y se recibe como respuesta un nuevo token de acceso fresco con los permisos actualizados a 1
- 3.5.3.5. PARTE 5: FastAPI manda la RESPONSE con el nuevo token de acceso con permisos a 1 al front-end, a la página pas4 (el pas4 ya redirigirá automáticamente al dashboard)

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 y procesamos: **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 `jwtUtils.py`. Este archivo Python, al que convenientemente le hemos puesto el mismo nombre que la clase en Java donde guardábamos el secret con el que generábamos los tokens, no es casual: justamente aquí también guardamos una copia del secret!

Ese “secret” es indispensable para validar si el token entrante en este y otros endpoints de fastAPI es válido o no. 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) el mismo secret.

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` [13]²⁷ se hace todo muy rápido.

²⁷un nombre aparentemente muy español pero cuyo acrónimo significa *JavaScript Object Signing*

3.5.5. Validación de datos (end-point entrada PDFs)

Figura 3.13: 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 | Debajo: PDFs subidos al sistema de archivos del servidor con fastAPI.

The diagram illustrates the validation process for uploaded PDFs across three main components:

- Postman Screenshot (Left):** Shows a POST request to `localhost:8000/api/subir-tickets-pdf`. The "Body" tab displays form-data entries for "arrius" containing various files. A green box highlights two successful uploads: "20230925 Mercadona 1..." and "20231004 Mercadona 2...". A red box highlights four rejected files: "pdfMassaGran.pdf", "pdfMassaPetit.pdf", "un audio.mp3", and "20250515 Carrefo...". An orange arrow points from the Postman response (containing JSON output) to the browser screenshot.
- Browser Screenshot (Center):** Shows a POST request to the same endpoint. The JSON response indicates 2 tickets were saved and 4 were rejected. A callout box highlights the message: "2 subidos | 4 rechazados total: 2 tickets facilitados".
- File System View (Bottom):** Shows the contents of the `tickets\3` directory in a terminal window. It contains two files: "20230925 Mercadona 13,71 €.pdf" and "20231004 Mercadona 27,40 €.pdf".

Al subir PDFs al sistema de archivos del servidor hemos tenido mucho cuidado, como vemos en la figura 3.13. 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). Se ha hecho trabajo en este aspecto tanto en [controlador.py](#) como en

and Encryption (JOSE)

[serveiValidaciones.py](#).

Sabemos que Será el usuario el que 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 (véase detalle de líneas en [este commit](#)).

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

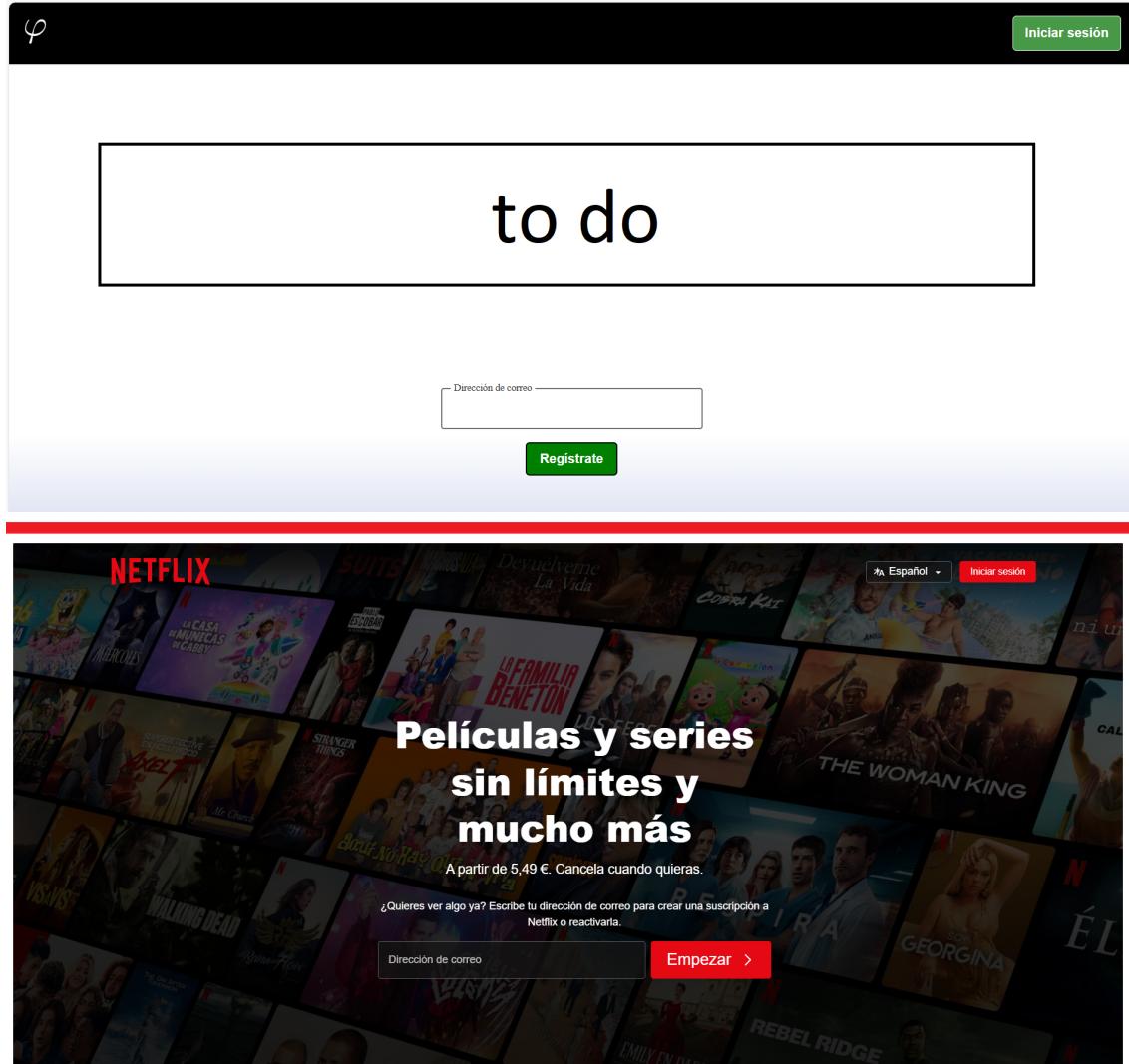
TO DO

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 autenticados se cubre en otra sección: apartado [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*.

Figura 3.14: **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 los tickets digitales al usuario (la página `pas4_ConcedirAccesGmail.html`); de nuevo análogamente a NetFlix, donde al usuario que ya ha pagado se le concede inmediatamente el acceso a las películas y series, en nuestro caso se le dará acceso al usuario al tablón de visualización de análisis de datos de los tikets 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 3.15, cuyas nomenclaturas explicamos en los siguientes *bullet points*:

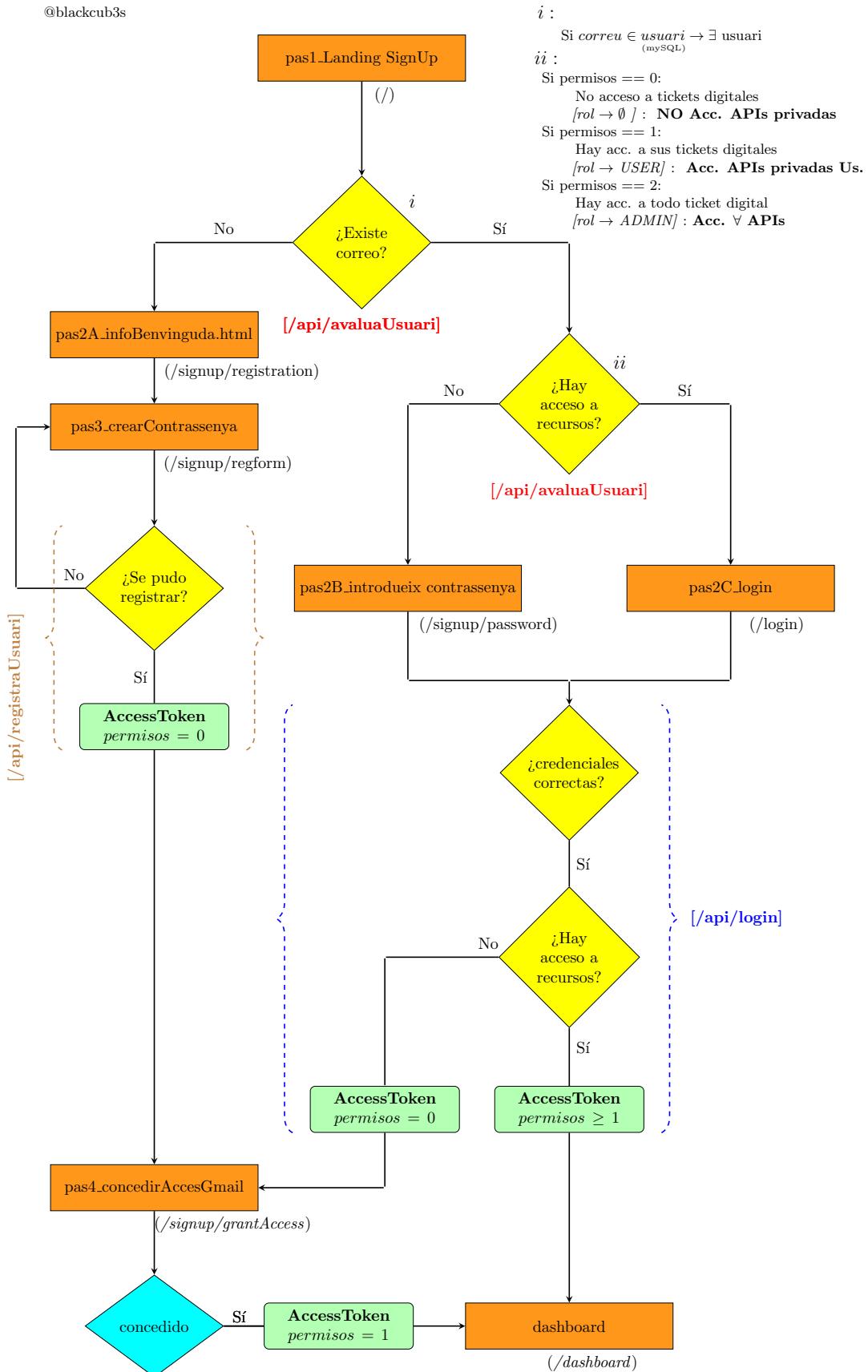
- Decisiones del back-end de Spring Boot al llamar a APIs: rombos amarillos.
- Las APIs que consume el front-end van entre corchetes (`[]`) y con color:
 - `[/api/avaluaUsuari]`: *end-point* que evalúa si el correo electrónico introducido pertenece a un usuario registrado y qué permisos tiene.
 - `[/api/registraUsuari]`: *end-point* que registra un nuevo usuario en el sistema y expide su AccessToken con permisos a 0 en las *claims* de su *payload*²⁸.
 - `[/api/login]`: *end-point* que gestiona el proceso de autenticación y generación del *JWT Access Token* con tres niveles de permisos posibles (0, 1 y 2).
- 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 las APIs: rectángulos naranja.
- Expedición de tokens de acceso (JWT) desde el endpoint del que emanan y enviados al front-end se representan con un rectángulo de fondo azul y bordes redondeados²⁹.

²⁸El lector puede consultar la explicación sobre lo que son las claims y el payload de un JWT en el apartado 3.4.2.2.

²⁹Consultad la estructura de los mismos en la figura 3.3.

Figura 3.15: Diagrama de flujo del enrutamiento completo del sistema *front-end* durante el proceso de registro, desde que el usuario introduce su correo en `pas1_LandingSignUp.html` (`index.html`) hasta obtener acceso al `dashboard`.

———— NOTA: Acceso a recursos \Leftrightarrow permisos ≥ 1 ————



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 estáticos³⁰ de dentro del mismo back-end de Spring Boot 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*³¹, y así lo hemos hecho en este proyecto³². 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 este³³.

Por ejemplo, del mismo modo que los endpoints de nuestra API del back-end en Spring Boot están protegidos 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 end-point 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.

³⁰HTML, CSS y JS son archivos estáticos.

³¹Separation of concerns

³²Si 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.

³³A 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 a los usuarios.

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 3.15, 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 descifrable³⁴.

Si está expirado, cuando hagamos la diferencia entre el valor “exp” del payload³⁵ y la función `Date.now()/1000`³⁶ 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 visionado de cualquiera de las dos páginas (se cargará el script que contiene esta función

³⁴Ojo, describirable no significa validable. Validable es lo que hacemos en el back-end con el secret, y no se muestra jamás en el cliente.

³⁵Segundos en que el token expira o expiró, desde la epoch.

³⁶Segundos actuales del navegador, desde la epoch.

antes de que se cargue el DOM³⁷). 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 en los párrafos anteriores (ver figura 3.16):

Figura 3.16: 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
```

³⁷El 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 3.15 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	Acceso a tickets como usuario (USER)
2	Acceso a tickets como administrador (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**³⁸. 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”³⁹ dentro del *payload* del token de acceso guardado en el *local storage*⁴⁰, 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:

³⁸Si 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!

³⁹No hace falta mencionar que se saca del campo permisos de la tabla usuariis de la base de datos que tenemos en mySQL.

⁴⁰Si queréis ver a qué me refiero con el payload, revisad de nuevo la figura 3.3.

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 3.16 previa, y lo mostramos a continuación en la figura 3.17:

Figura 3.17: 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 3.16 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 ha concedido acceso a tickets digitales) pueda pedir de nuevo acceso a esos 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 0 (e.g., ha puesto correo y contraseña y se ha registrado, pero no ha dado acceso a tickets digitales todavía) pueda tratar de visualizar el `dashboard` a la espera de obtener una información de unos tickets que él todavía no ha proporcionado.

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 ha concedido acceso a tickets digitales) 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 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 3.18:

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

Figura 3.18: Script `restringeixVistesPrivades_USUARI_NO_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. Recordemos 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”⁴¹.

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 de figura [3.16](#)). 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 3.19: 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 `mercApp`⁴², el token de acceso lo recibe por primera vez en el

⁴¹A 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.

⁴²Sea que ya estuvo registrado -permisos 0-, dio acceso a sus tickets digitales -permisos 1- o es superusuario -permisos 2-.

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: ⁴³ Para más información sobre el código JavaScript del front-end que lo permite véase figuras y [3.20](#) y [3.21](#)):

```
{
  "usuari": {
    "aliases": "the protein kingdom",
    "permisos": 2,
    "idUsuari": 1
  },
  "existeixUsuari": true,
  "AccessToken": "eyJhbGciOiJIUzI1NiJ9.eyJwZXJ [...]",
  "teAccesArecursos": true,
  "contrasenyaCorrecta": true
}
```

Figura 3.20: Fragmento de código en `pas2C_login.html` dentro del código javascript para manejar códigos de error. Cuando el back-end de Spring Boot devuelve 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 DE LA RESPUESTA JSON -----
.then(dadesExitosesJSON => {
```

⁴³Esto lo hacemos para luego poder mandarlo de vuelta al servidor en las siguientes solicitudes que requieran autenticación y autorización.

Figura 3.21: Fragmento de código en pas2C_login.html dentro del código javascript para obtener el token de acceso (detalle en rojo).

```

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

    console.log(dadesExitosesJSON); // Imprimir la respuesta en la consola (tiremos a producción)
    if (dadesExitosesJSON.existeixUsuari) {
        if (dadesExitosesJSON.contrasenyaCorrecta) {
            let tokenAccesJWT = dadesExitosesJSON.AccessToken; //extraemos el token
            localStorage.setItem("AccessToken", tokenAccesJWT); //guardamos token al localStorage

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

```

Ahora bien, si el usuario nunca se ha registrado en nuestra aplicación⁴⁴, 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 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 3.4.2.4, 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 embedados que pueden hacer llamadas asíncronas y obtener un token de acceso del servidor y guardararlo en el localStorage (un detalle de los formularios de estas páginas se encuentra en la figura 3.22):

⁴⁴Es decir, no tenemos su correo electrónico en BBDD.

Figura 3.22: Detalle de los formularios que permiten generar llamadas a los dos endpoints generadores de tokens de acceso (/api/login y /api/registraUsuario). De izquierda a derecha las páginas que los contienen: pas2C_login.html, pas3_crearContrasenya.html y pas2B_introducirContrasenya.html

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 3.4.3).

TO DO FER-HO

3.6.7. Diseño (UX/UI): páginas publicas

TO DO FER-HO: parlar disseny responsive

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 es⁴⁵.
- **Persistencia de datos:** las vistas de la página del dashboard permiten visualizar datos extraídos de una base de datos de mongoDB.

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 superusuario (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 3.23, 3.24 y 3.25).

Además se ha utilizado un sistema (figura 3.26) para conseguir que al hacer hover en cada link cambie su color a gris y aparezca una línea por debajo con una transición

⁴⁵en la medida de lo posible, dado que no todas las librerías admiten responsividad: por ejemplo, chart.js no lo es.

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 3.23: 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 3.24: Detalle barra de navegación con el drop-down desplegado.

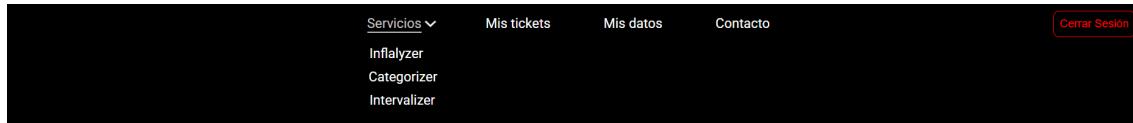


Figura 3.25: Detalle del código que hizo posible el drop-down del menú. Cada elemento drop down se ha animado mediante animate.css definiendo distintas duraciones de la animación *fadeInDown* para conseguir un efecto persiana. El uso del selector descendiente y el no abuso de los IDs a menos que sea necesario es una metodología seguida a lo largo del diseño del dashboard.

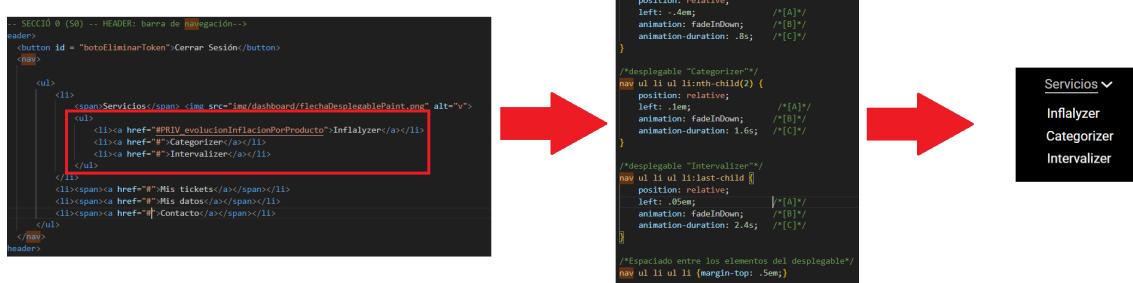


Figura 3.26: Transiciones suaves de propiedades *border-bottom* y *color* al hacer hover en los spans que envuelven los links tanto en la barra de navegación como en la footer.



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 3.27.

Figura 3.27: 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 tikets de cada usuario, que se extraen también de la BBDD (ver detalle figura 3.28). El lector puede consultar el fichero `estils.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 [14]. 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 3.29. 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 [3.25](#)) de modo que en lugar de que aparezcan con un temporizador desde la carga de la página como nos permite la librería animate.css, sino 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 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.

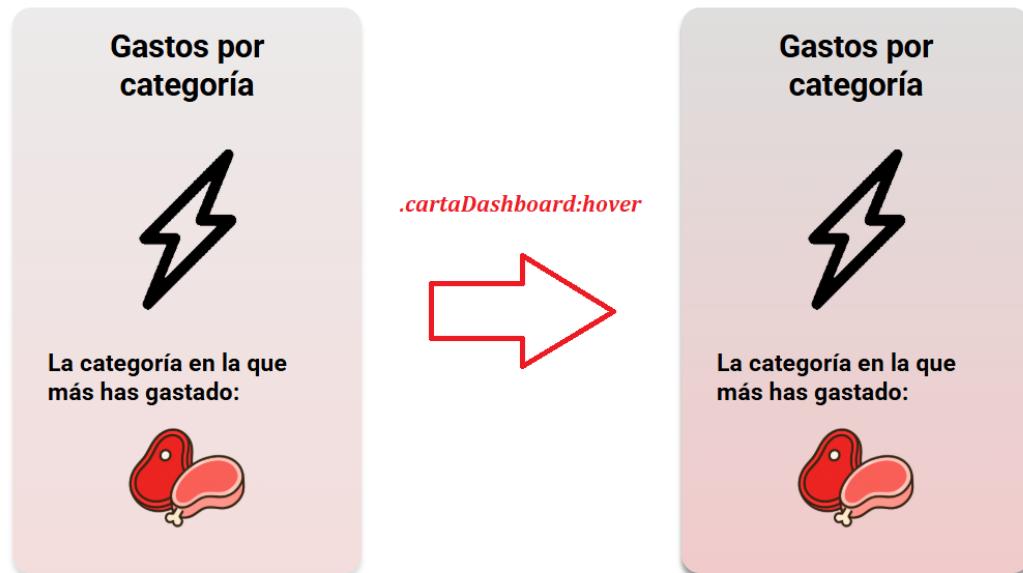
Figura 3.28: Detalle de la sección con las tres cards: características resumidas del dashboard. Los textos en color son placeholders para lo que se extraerá de la BBDD.

¡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 3.29: 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 3.30). Este “swiper” artesanal, se creó tomando los mismos iconos de swiper, eliminado 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 3.30: Detalle de la parte superior del intervalizer: obsérvese el paginador creado manualmente

Inflalyzer: evolución de los precios por producto ("inflación")



A nivel de UX/UI el paginador de la figura 3.30 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 3.31).

Figura 3.31: 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 [3.30](#) 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): “categoryzer”

TO DO

SECCIÓN 5 (S5): “intervalizer”

TO DO

3.6.8.2. Diseño del “pas4”

El diseño del pas4 no entraña mucha más dificultad que el diseño del dashboard. Se ha seguido un procedimiento parecido para la generación de iconos que en el dashboard (los iconos generados se hicieron con IA y se extrajo el canal alpha con gimp: podéis verlos en la carpeta de [edición de iconos](#)).

El ícono para cerrar sesión es el mismo que en el dashboard y 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 3.32: Vista del grid de dos columnas del pas4 en la versión desktop



Paso 1: Descarga tus tickets

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

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

Un aspecto diferencial con el **dashboard** es que el diseño responsive de la parte clave 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 texto (h1 y h2): nótese que en el HTML la imagen de descarga de tickets y la de “relájese” aparecen en después del contenedor de h1 y h2 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 produce que al ponerlo en vertical en un grid de una sola columna para la versión mobile tanto el icono de “descarga tickets” como el de “relájese” 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 3.33 (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 3.33: 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.

```
/*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 imágenes 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, 3.34). Con el fondo blanco ya queda perfecto, como hemos visto en la figura 3.33; 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 3.34: Gradiente lineal a 135 grados para el diseño del pas4

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.

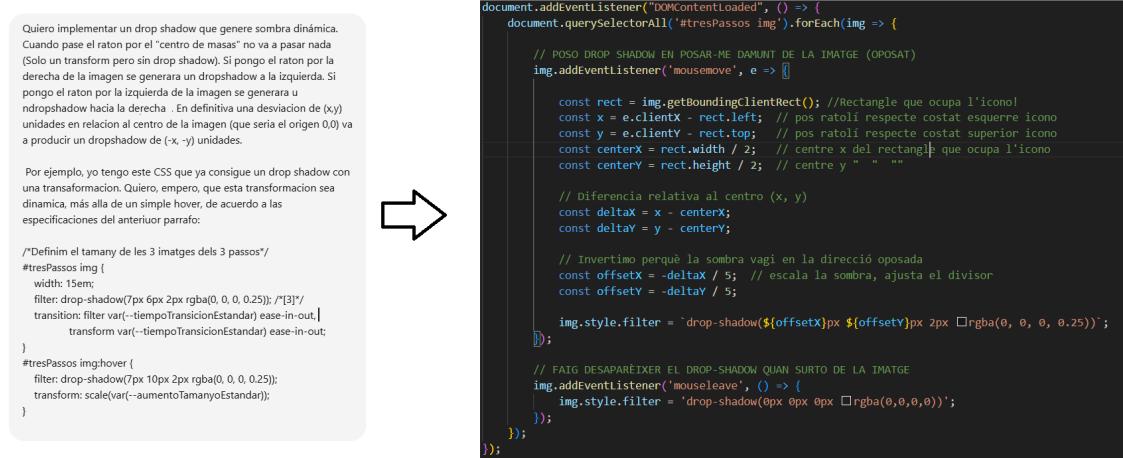
Espera a que los guardemos en la base de datos el contenido de tus tickets. Una vez guardados te redirigiremos automáticamente a la página definitiva. Guarda tus tickets en PDF en una carpeta: serán de utilidad luego

Finalmente lo que sí es una diferencia en relación al dashboard es que se ha automatizado la propiedad dropshadow utilizando JavaScript en lugar de haciéndolo con 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 3.35. 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 3.35: Desde el prompt inicial al código final



The diagram illustrates the transition from a user's initial thought to the final implemented code. On the left, a light gray box contains handwritten text in Spanish about implementing a dynamic drop shadow. An arrow points from this text to the right, where the final JavaScript code is shown in a dark gray box.

```

document.addEventListener("DOMContentLoaded", () => {
  document.querySelectorAll('#tresPasos 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

El **paginador customizado** que vimos en la figura 3.30 requiere JavaScript. El código que lo consigue orquestrar se encuentra en el archivo [paginadorInflacio.js](#).

TO DO PARLAR DEL GRAFIC D'INFLACIO I DEL SEU JAVASCRIPT (continuacio inflalyzer)

TO DO PARLAR DEL DIAGRAMA DE SECTORS I DEL JS

TO DO PARLAR DEL CATEGORIZER I DEL JS

TO DO PARLAR DEL INTERVALIZER I DEL JS

La extracción de datos de la base de datos se hace principalmente desde [extractorDadesPersistencia_enCarregarPagina.js](#)

HI HA MES ARXIUS DE PERSISTENCIA? REAVALUAHO?? HI HA MES ARXIUS DE PERSISTENCIA? REAVALUAHO??

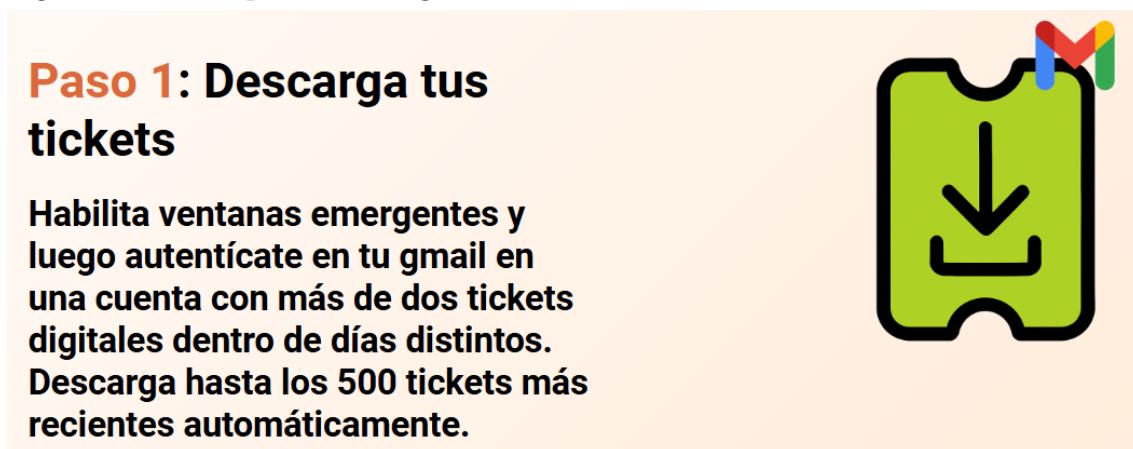
3.6.9.2. Arquitectura pas4

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 permitá que los usuarios de nuestro sitio web (en el [pas4_concedirAccesGmail.html](#)), siempre que tengan una cuenta de google con tickets digitales de Mercadona enviados por el supermercado, puedan descargárselos a la carpeta de descargas del ordenador en el que están navegando mediante **un solo click** y una autenticación a su cuenta de Google: concretamente en la carpeta de descargas de su ordenador. Esto lo haremos mediante Javascript puro, cuando el usuario pulse el icono mostrado en la figura [3.36](#):

Figura 3.36: Pulsando sobre el icono iniciamos la llamada a la api de google: con ello autenticamos el usuario y éste concede acceso a su correo donde están los tikets digitales dentro, 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.js⁴⁶](#).

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

⁴⁶Luego explicamos por qué este script pertenece a un repositorio auxiliar

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 ([repo 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 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 externa⁴⁷:

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

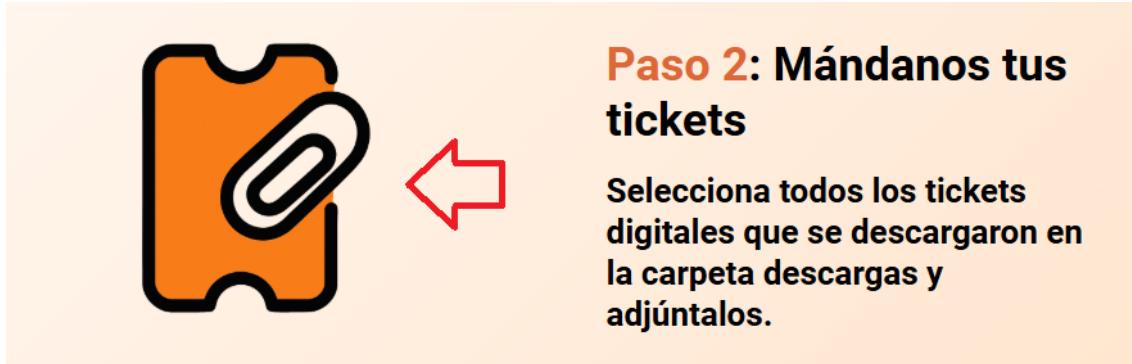
PARTE2: Subir tickets digitales del ordenador del usuario con JavaSce- ript hacia el back-end de fastAPI para su procesamiento

NOTA: *El procesado que involucra servidor lo mostraremos en otra sección: la parte de la solicitud de subida y parseo de datos en fastAPI: sección 3.5.3.*

La subida de tickets digitales desde el front-end la hacemos como vemos en la figura 3.37:

⁴⁷Solo que la CDN la hemos creado expresamente para la ocasión.

Figura 3.37: 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.



TO DO TO DO POSAR A LA IMATGE ANTERIOR UN FRAGMENT DEL JAVASCRIPT QUE FA LA CRIDA EN PAS4 CAP A ENDPOINT API/suvir-tickets-pdfs

PARTE 3: Navegador espera que termine el proceso y recibe token de acceso con permisos 1

Al termi

3.6.10. Arquitectura (JavaScript): páginas públicas

TO DO FER-HO

3.6.11. Diseño de iconos

3.6.11.1. Áreas de producto

Los iconos de las 8 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 3.38, 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 3.39). La utilidad de ese último procedimiento en relación al anterior puede verse en la figura 3.40 donde comparamos la aplicación de ambos procedimientos al mismo ícono.

Figura 3.38: Se ha pedido a chat gpt la creación del ícono de verduras y hortalizas. Chat gpt 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*.

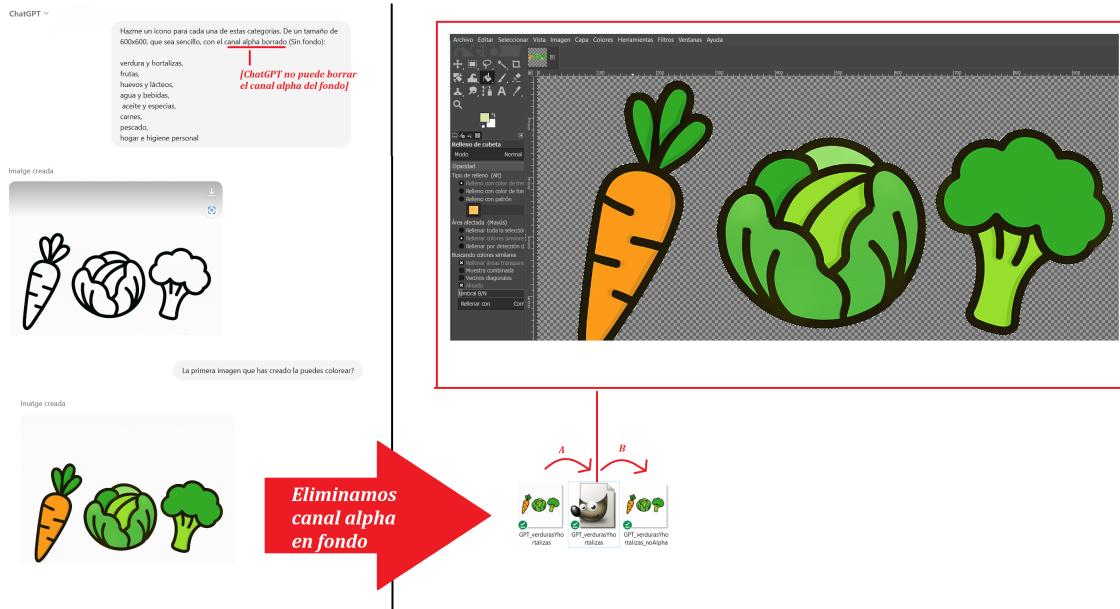


Figura 3.39: En determinados casos no es suficiente con la estrategia sencilla del paso anterior y debemos seguir este procedimiento, que incluye el uso de la varita mágica, y de los pasos *Seleccionar → invertir* (invertir selección) y *Ctrl + X* seguido de *Ctrl + V* como mostramos en la figura.

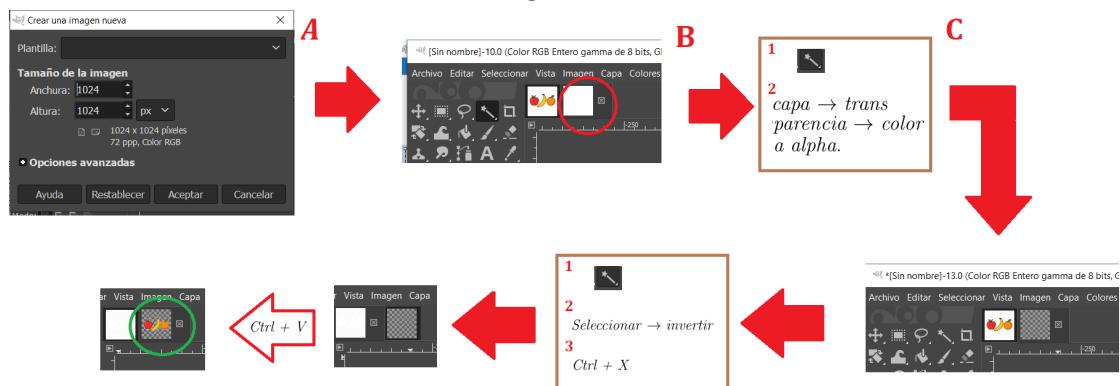


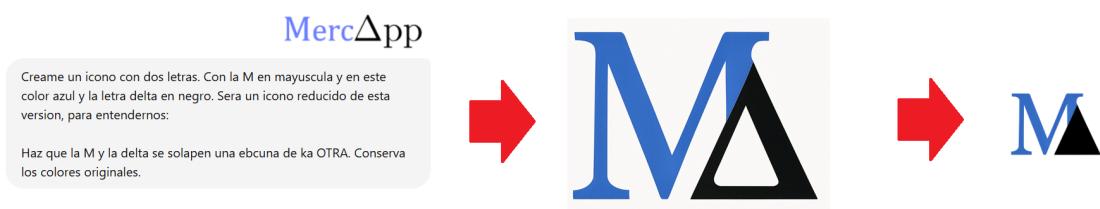
Figura 3.40: El procedimiento de Gimp mostrado en la figura 3.38 genera un ícono que en contexto se verá como la imagen de la izquierda: la card central del dashboard mostraría un ícono con bordes rectangulares no intencionados. El procedimiento de la figura 3.39, en cambio, nos da la imagen de la derecha que no tiene fondo alguno y permite que el ícono se asiente a la perfección en la card.



3.6.11.2. 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: 3.41.

Figura 3.41: El prompt pasado a ChatGPT para obtener una versión reducida del ícono de nuestra APP. Incluso con tipos 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/>.

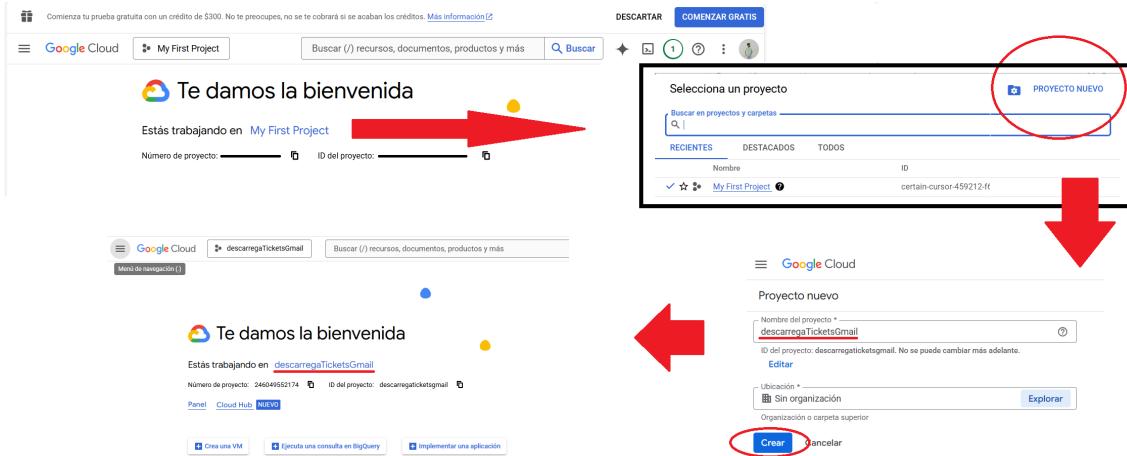
Estas configuraciones son necesarias por varios motivos: el *primero*, es que tenemos que obtener el (**Client ID**) para que se identifique nuestra web en relación a Google Cloud; el *segundo*, es configurar los “*scopes*” que le dicen a Google qué tratamos de conseguir del Gmail de los usuarios que se autoricen ante google mediante su mecanismo de autenticación denominado OAuth2 (aquí el Scope es solo para lectura de gmaile -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 “descargaTicketsGmail” (ver figura 3.42):

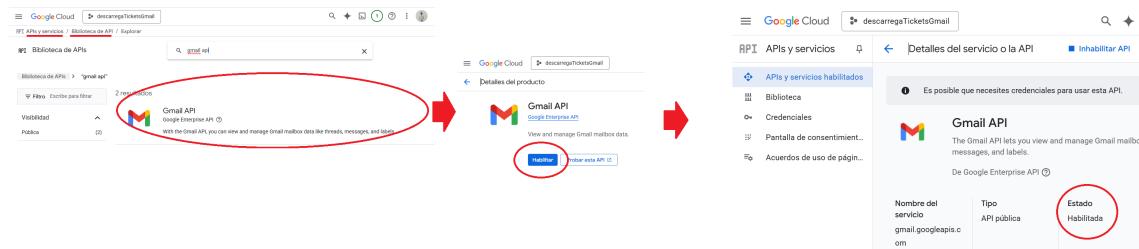
Figura 3.42: Pasos para crear un nuevo proyecto en la consola de google cloud.



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 3.43).

Figura 3.43: Habilitación de la GmailAPI en el proyecto “descarregaTicketsGMAIL”



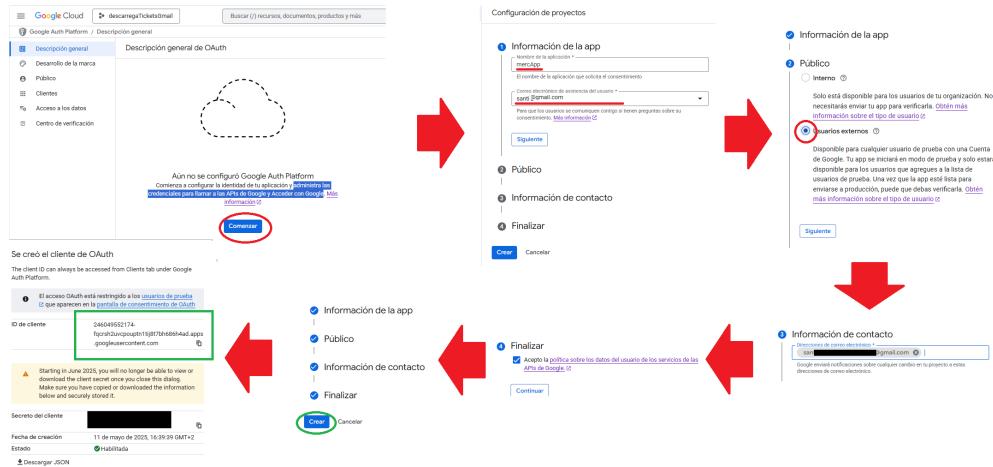
PASO 3: Configurar la pantalla de consentimiento OAuth

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 Gmail⁴⁸.

Análogamente al paso anterior seguimos los menús de la izquierda “*APIs y servicios*” → *Pantalla de consentimiento OAuth* (ver figura 3.44).

⁴⁸¡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 3.44: 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 3.44 obtenemos una ventana de configuración⁴⁹ que nos permite modificar aquello para lo que sirve el *ClientID*⁵⁰ como podemos ver en la imagen 3.45. A través de esta configuración tenemos que decirle a Google de qué dominios y de qué puertos puede recibir llamadas para autenticarse.

⁴⁹Podemos llegar a esa ventana de configuración -es decir al contenido de la figura 3.45- del mismo modo si optamos por seguir una ruta por menús: *Apis y servicios* (buscador) → *credenciales* → *+ crear credenciales* → *ID de cliente Oauth*.

⁵⁰El 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 3.45: 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 3.45 esta será la vista que tendremos si entramos en el buscador, en las **credenciales** y vemos los ID de clientes OAuth2.0:

Figura 3.46: Vista de la aplicación cloud después de la creación del ClientID

Nombre	Fecha de creación	Tipo	ID de cliente	Acciones
mercAppWebPas4	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 client ID. Para ello o bien hacemos lo del paso 6 (que luego 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 tiquets digitales (ver figura 3.47) y con eso la autenticación será posible, por ahora, solo para este correo.

Figura 3.47: 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 [?](#)

+

Add users

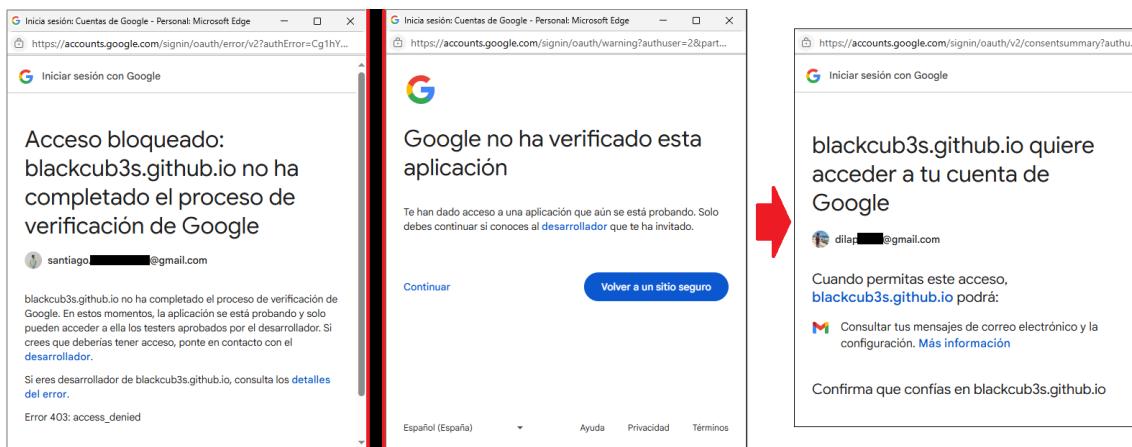
Filtro Ingresar el nombre o el valor de la propiedad [?](#)

Información del usuario

dilar._____@gmail.com

A continuación podemos ver en la figura 3.48 qué pasa cuando tratamos de acceder a autenticarnos en google:

Figura 3.48: **IZQUIERDA**: tratamos de loguearnos con un usuario que no esta 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.



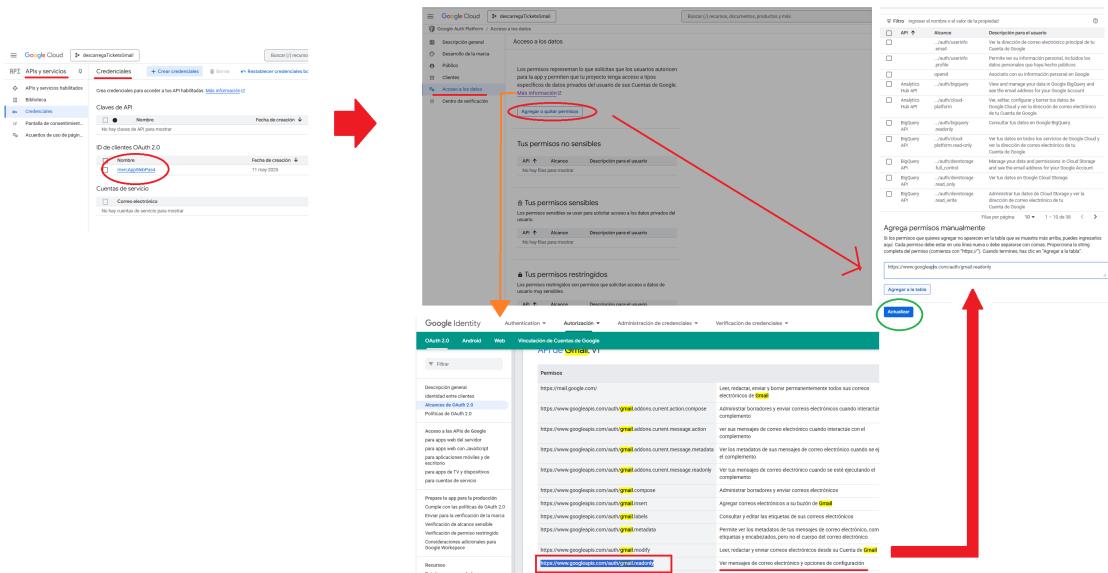
PASO 6: Validar la aplicación descargaTicketsGmail para que cualquier usuario externo pueda usarla (solo en producción y DESPUÉS de autorización expresa de Google).

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.httponly* y deberemos verificar aplicación en el cloud. Si no lo hicieramos, 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 paso 4 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 web⁵¹. Para solucionarlo tenemos que añadir el scope necesario como vemos en la figura 3.49:

Figura 3.49: 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 3.49 tenemos que guardar esta configuración. Para ello seguimos los pasos de la imagen 3.50:

⁵¹Si os fijáis en la figura 3.44 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 3.50: 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.

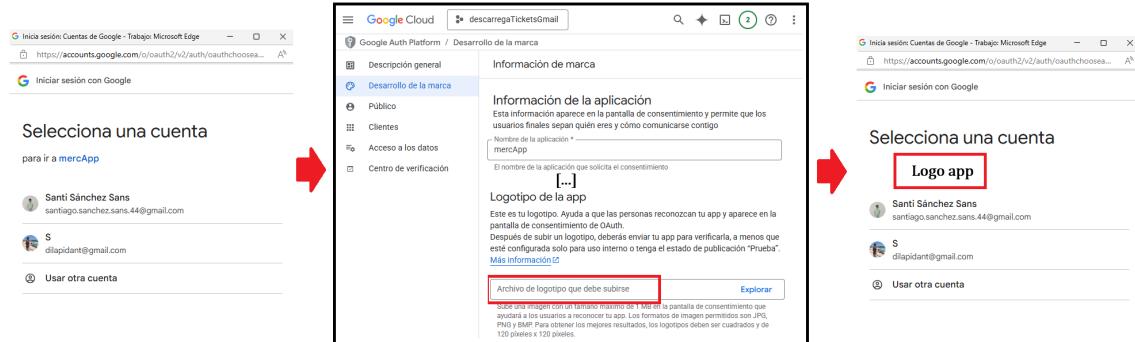
The screenshot shows the Google Auth Platform interface. On the left, there's a sidebar with icons for 'Descripción general', 'Desarrollo de la marca', 'Público', 'Clientes', and 'Acceso a los datos'. The 'Acceso a los datos' item is selected and highlighted in blue. The main content area is titled 'Acceso a los datos' and features a small icon of a person with three dots above them. Below this, it says 'Tus permisos restringidos' (Your restricted permissions) and describes them as 'permisos que solicitan acceso a datos de usuario muy sensibles' (permissions that request access to very sensitive user data). A table titled 'Permisos de Gmail' (Gmail permissions) lists one permission: 'Ver mensajes de correo electrónico y parámetros de configuración' (View emails and configuration parameters), which corresponds to the API scope './auth/gmail.readonly'. At the bottom of the page are two buttons: 'Save' (highlighted with a green circle) and 'Descartar cambios' (Discard changes).

Y el último caso es validar. Que google tiene que dar el vistobueno. Sin embargo asumo que esto no va a pasar, al menos no en plazo, con lo cual, directamente solo podremos usar los correos (por ahora solo uno) en el listado de usuarios permitidos mostrado en el PASO 5, para descargar los tickets digitales.

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 3.51. Como podemos ver en esta entrada de Stackoverflow [16], 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 3.51: Definición de icono saliente para mostrar a los usuarios que se autentican “Apis y servicios” → “Credenciales” → “MercAppWebPas4” → “Desarrollo de la marca”



Evaluación y Conclusiones Finales

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();
}

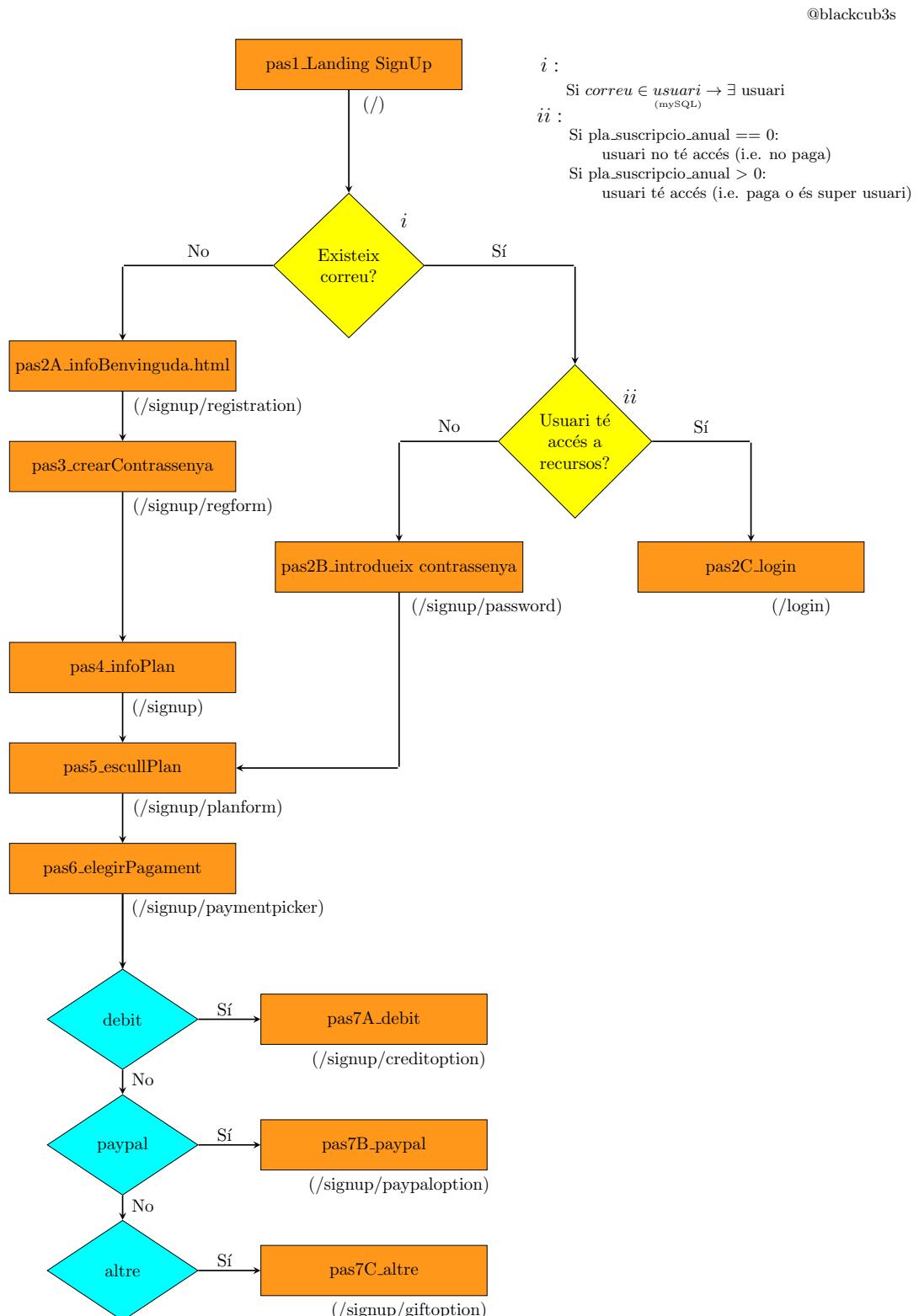
```

5.4.2. Controlador con restricciones aplicadas

La restricción aplicada es @PreAuthorize. Solamente usuarios que tengan rol administrador 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);
    }
}
```

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)

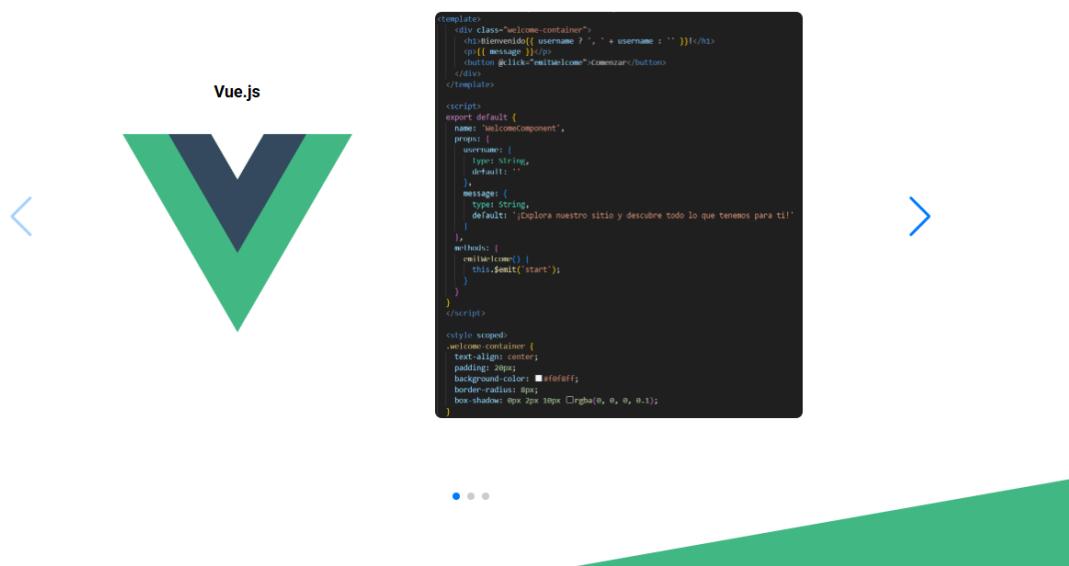
- **Responsabilidad única (SRP):** Cada módulo debería hacer una sola cosa (`{Frontend → interfaz}`, `{Backend → procesamiento}`).
- **mantenibilidad:** Al usar una arquitectura modular cada parte puede evolucionar por separado. Podemos desplegar solo el front o el back. En el futuro podremos cambiar 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 [17] (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.

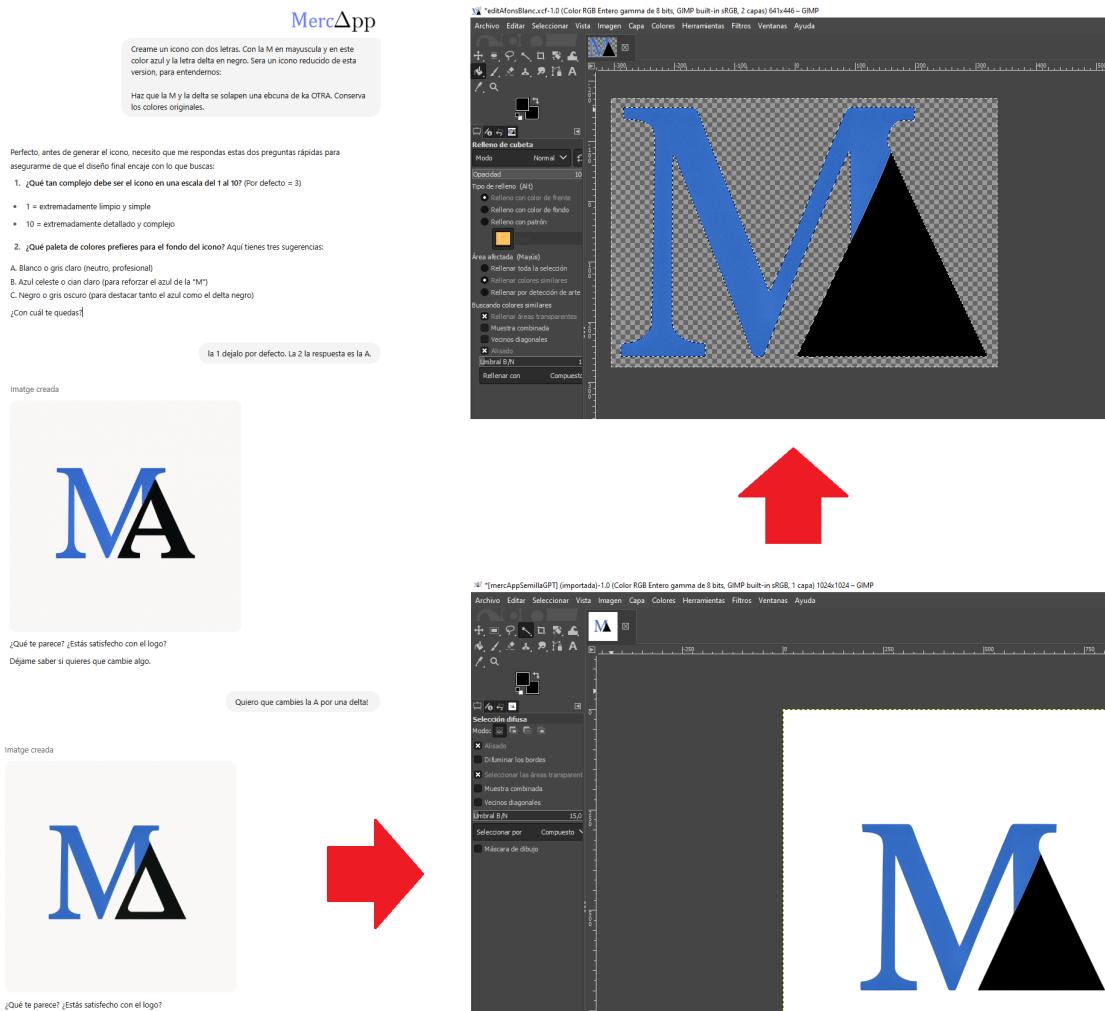
¿Qué diferencias fundamentales hay entre los tres frameworks?



5.9. Creación del icono 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: [3.6.11.2](#)

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.



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] 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.
- [7] 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.
- [8] jwt.io. Json web tokens - jwt.io. <https://jwt.io/>. Accedido el 27 marzo de 2025.
- [9] 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.
- [10] Postman. Postman api platform. <https://www.postman.com/>. Accedido el 9 abril de 2025.

- [11] 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.
- [12] 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.
- [13] Michael Davis. python-jose: A jose implementation in python. <https://pypi.org/project/python-jose/>, 2025. Version 3.4.0, available on PyPI.
- [14] Wikipedia. Número áureo - wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/N%C3%BAmero_%C3%A1ureo. Accedido el 6 mayo de 2025.
- [15] 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.
- [16] 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.
- [17] NGINX. Nginx — high performance load balancer, web server, & reverse proxy. <https://www.nginx.com/>. Accedido el 8 abril de 2025.

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

The screenshot shows the 'Free cloud features and trial offer' section of the Google Cloud Free Program documentation. It highlights the 90-day, \$300 Free Trial period, which includes \$300 in free Cloud Billing credits. A sidebar on the right provides links to various topics related to the trial offer.

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)

The figure compares the free tiers of App Engine and Cloud Run. App Engine's free tier is limited to 28 hours per day of F1 instances, 9 hours per day of B1 instances, and 1 GB of outbound data transfer per day. In contrast, Cloud Run offers 2 million requests per month, 360,000 GB-seconds of memory, 180,000 vCPU-seconds of compute time, and 1 GB of outbound data transfer from North America per month. A red circle with a cross marks App Engine as 'dangerous' due to its lack of scaling to zero. A green checkmark marks Cloud Run as a better choice.

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 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 \[15\]](#) ([link](#)) podemos ver que cada una de las funciones que usamos en el script `scriptExtraccionBoto.js` usado para extraer los tickets de los correos electrónicos del Gmail de un usuario consumen ciertas unidades de quota de la API de Google. Nosotros usamos `messages.attachments.get`, `users.messages.list` y `messages.attachments.get`, que tienen un uso de quotas por cada llamada cada uno de 5 unidades. 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- supondrá 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](#)

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

Figura 5.5: Unidades de cuota de procesamiento que nuestra aplicación no puede exceder al hacer llamadas a gmailAPI sin incurrir en excesos [15].

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). [15].

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

A diagram consisting of a vertical bracket with three segments, one under each row of the table. Above the first segment is the number '5'. Above the second segment is the number '5'. Above the third segment is the number '5'. An orange arrow points from the right side of the bracket to the text '15 uc/ticket'.