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 7X. XX de Junio 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 Lãberit) y valorarla con su idiosincracia -incluyendo sus imperfecciones-.

Índice general

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-A[1] en los tickets de compra de distintos paises [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 consifgue 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

[1] Es 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 establecimiento[2].
- **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

[2]La 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 Lãberit 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 Lãberit 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 Politécnica 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 Europa[1]. No es un lenguaje que vaya por modas y ejercitarlo puede permitir iniciar una carrera enfocada en una tecnología que no se prevé que desfallezca en un futuro cercano.

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 el ***coste total*** de la cesta de la compra del usuario a lo largo de distintas ventanas temporales (por meses, períodos de 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 9 categorías: verdura y hortalizas, frutas, huevos y lácteos, agua y bebidas, aceite y especias, carnes, pescado, hogar e higiene personal[2].

REQUISITO D[3]: Podremos permitir que los PDFs descargados del correo del

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

[2] Para 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.

[3] Requisito añadido después de la presentación del proyecto.

usuario se almacenen en una carpeta local del mismo para que pueda verificar la extracción de los datos.

REQUISITO E[4]: 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.js[4]; y una librería de css que permite aplicar transiciones, animate.css[5]. 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 3.5.1). 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.

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).

[4]Requisito añadido después de la presentación del proyecto.

- Python dentro de un contenedor docker (o python a secas, para descargar los pdfs del correo) y parsear el contenido de los mismos: con sklearn, numpy y spacy que luego se podrán pasarlos a la BBDD mongoDB.

2.3.3. 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 tikets, estos se descargarán de mongoDB y almacenarán en localStorage del cliente. En consultas posteriores, los datos se obtendrán directamente de localStorage sin necesidad de hacer peticiones al servidor, hasta que se termine su token de acceso: en cuyo caso se borrarán los datos del localStorage. Evaluaremos la viabilidad de este sistema durante el desarrollo; Esto es la fase de diseño y como tal, **AQUEST PARAGRAF POTSER EL PODRAS ELIMINAR SI FINALMENT NO HO FAS I FAS CRIDES DIRECTES A BBDD**, en caso de no ser factible, las consultas se harán directamente en MongoDB.

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 sesion (páginas públicas) del proyecto, en comunión 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 3.7 principalmente) y todo lo redactado de la parte de SpringBoot del proyecto (ver apartado 3.3).

Durante el horario de las prácticas de Lãberit (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 3.1, 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. Diagramas de la aplicación

2.5.1. diagrama de sistemas

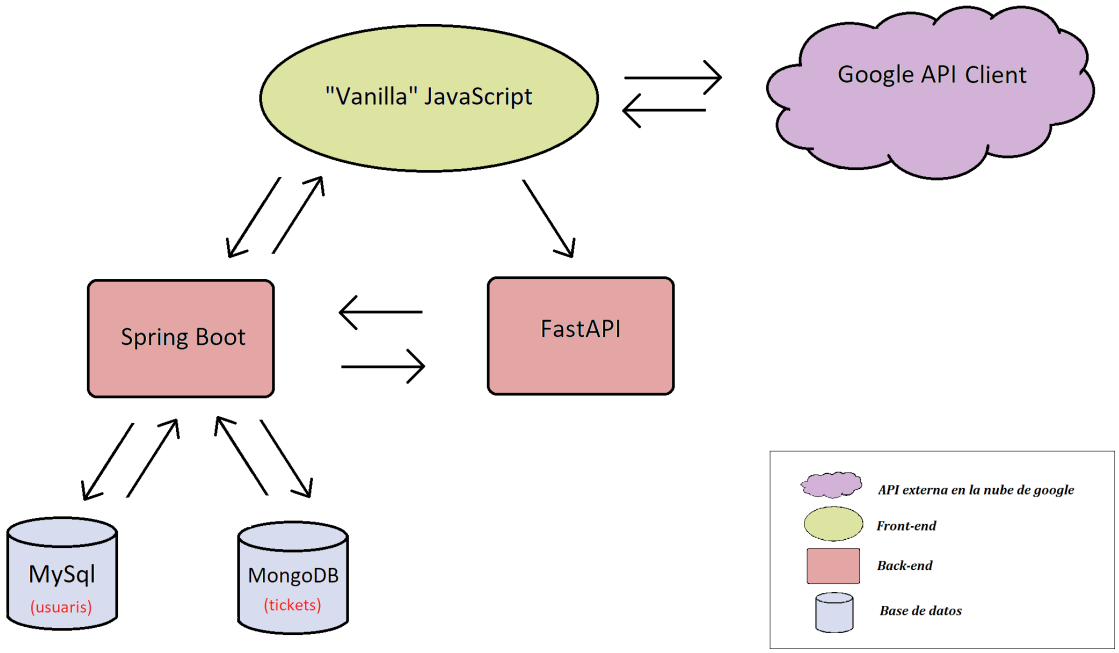


Figura 2.1: Diagrama de sistemas de la aplicación mercApp. Tenemos el back-end principal (Spring Boot) donde se mandan a guardar y leer tanto los usuarios (mySql) como los tickets (MongoDB). El back-end secundario, con fastAPI, parsea los tickets y los pasa a formato estructurado JSON para hacer que Spring Boot lo persista en 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.

2.5.2. Camino durante registro de usuario

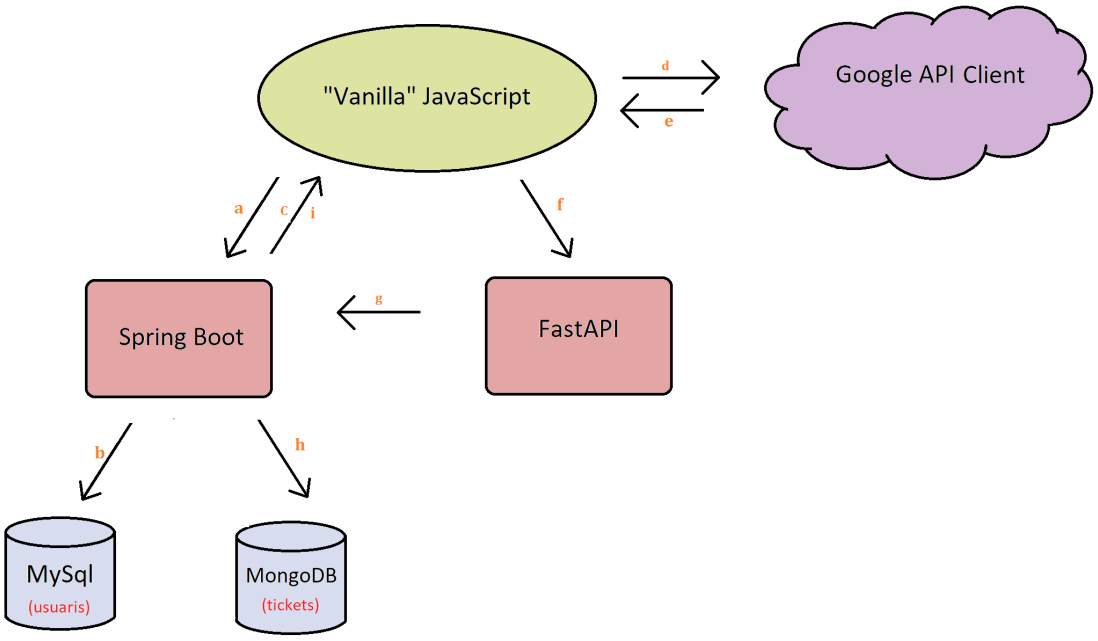


Figura 2.2: Simplificación de los caminos activados en el diagrama de sistemas durante el registro de un usuario, hasta que este tiene acceso al dashboard de visualización con éxito en todas las partes del proceso: a) Usuario pone correo y contraseña. b) guardamos correo y hash contraseña. c) Mandamos 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.

2.5.3. Camino durante inicio de sesión de usuario con pleno acceso a tikets digitales

TO DO TO DO

O si n o el fas borra el **Camino durante registro de usuario** i deixa el diagrama de sistemes només.

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, clonado y "despliegue" (en local) de sus componentes en vuestro ordenador personal utilizando los servidores embedidos en los entornos de desarrollo de workbench, vscode e IntelliJ **PONER OTRO SI HACE FALTA** si así lo deseáis.

Link al repositorio	https://github.com/blackcub3s/mercApp
Página desplegada	TO DO

Cuadro 3.1: Enlaces del proyecto.

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 fallará[1].

Para el frontend se ha utilizado `VScode`, con la extensión live server para poder hacer llamadas al back-end directamente desde el puerto **5500**. Esto podría hacer las veces de una CDN donde podrían estar alojados los archivos estáticos (HTML, CSS y JavaScript)[2].

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

[1] ¡Para correr el proyecto back-end con *Spring Boot* abrir con intelliJ la carpeta app!

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

3.3. Desarrollo back-end (Spring Boot)

3.3.1. Estructura de la aplicación

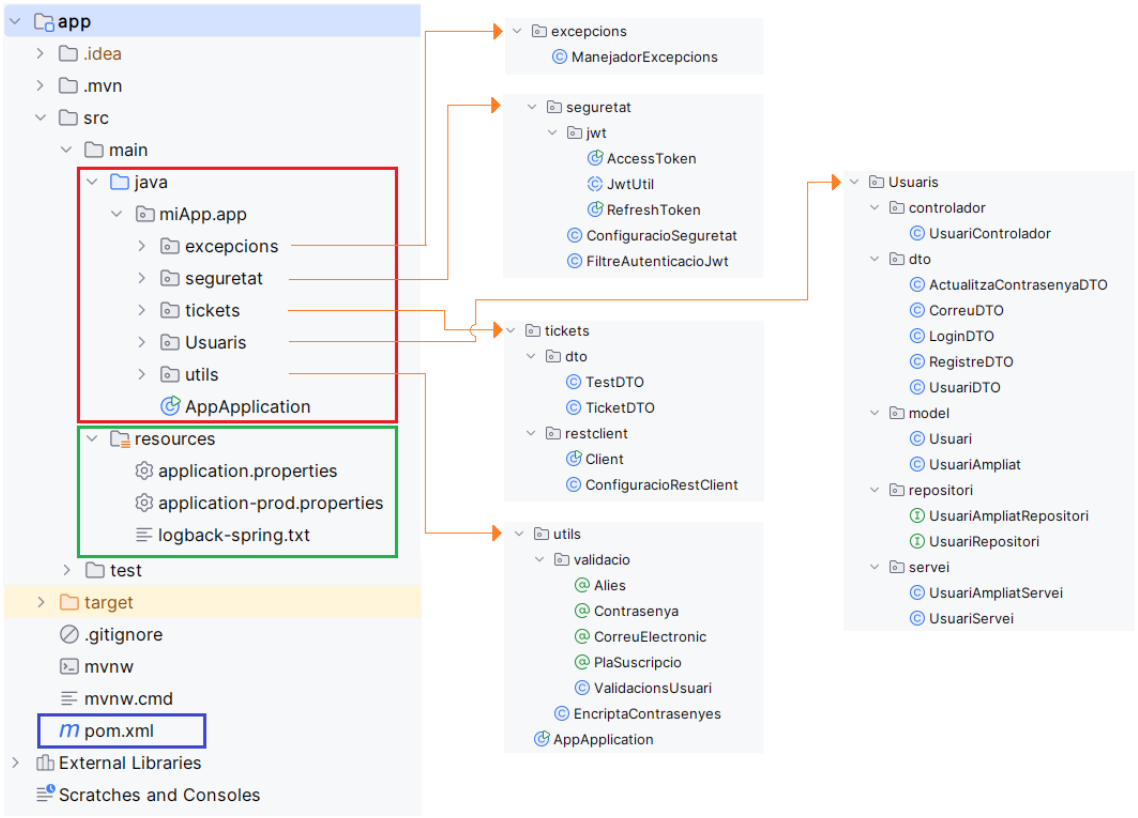
La parte de mercApp programada con Java (Spring Boot) se puede ver en el repositorio de GitHub del proyecto abriendo esta carpeta: aquí (eecomendamos al lector que abra y corra el proyecto con IntelliJ abriendo la carpeta mencionada del GitHub, pero en local). Se podrá ver entonces *tres* rutas importantes (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 3.1 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 conexionado con la base de datos (ver figura 3.1 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 terminal[3]. 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 Maven[4] (ver figura 3.1, 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.

[3] La ruta se crea automáticamente

[4] Maven 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 3.1: Estructura del back-end de Spring Boot en la aplicación mercApp. (Cajas de color → {clases del proyecto, archivos de configuración, dependencias de Maven})



3.3.1.1. 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 3.3.3.

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 3.3.2.3; 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 3.3.2.5.

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 3.3.4.

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.

3.3.1.2. src/main/resources: archivos de configuración

TO DO EXPLICAR application.properties

3.3.1.3. pom.xml: dependencias de maven

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

3.3.2. Autenticación y Autorización

3.3.2.1. 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 servidor[5]. 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

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

usuario; con ello, el servidor rechazará la integridad del token y evitará 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-

plejidad[6], 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 aceso y el de refresco, como comentamos):

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

3.3.2.2. ¿Qué compone un JWT?

El JWT se compone de tres partes separadas por *puntos*: **el header**, **el payload** y **la signatura**. En la página https://jwt.io/, como veremos después, se puede ver si los tokens son válidos, observar el contenido de su payload, etc. [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 toquen ("iat", o "issued at") y la fecha de expiración ("exp" o "expiration time"). También podemos tener ahí otros pares clave valor que podremos querer definir, por ejemplo, que contengan el id del usuario y sus roles o permisos que son los que nos permitirán dejar que un determinado usuario pueda consultar o no ciertos recursos.
- **La signatura**: Es la parte que garantiza la integridad del token y evita que sea alterado por terceros. Se genera aplicando un algoritmo de hash (en nuestro caso el SHA256) a la combinación del header y el payload, junto con la clave secreta que solo conoce el servidor (es lo que permetirá al servidor rechazar el token si no es válido -i.e. ha sido manipulado).

Podéis observar estas tres partes en colores en la figura 3.2 que veremos después.

3.3.2.3. 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>
```

[6]Se puede ver una tabla de diferencias más en profunidad, 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 continuacion 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 exprirar el token

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

  //FINALITAT: Generar un JWT d'acces.
  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();
}
```

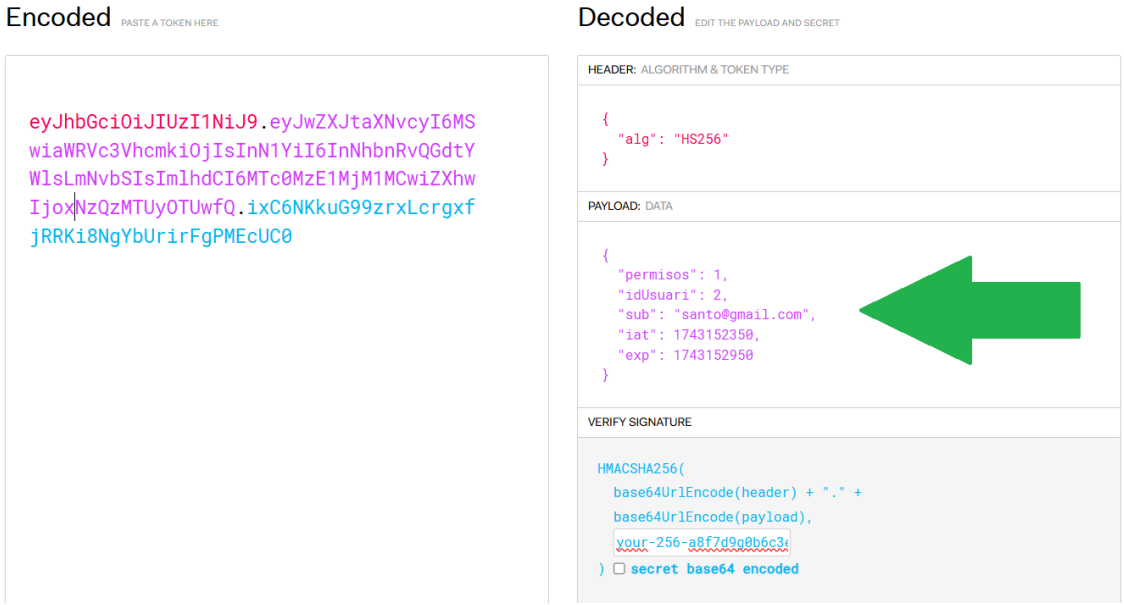
Con la función **genera()** de la clase AccessToken arriba mostrada, y con los parámetros necesarios que serán necesarios para autorizar (idUsuari) y autenticar (permisos), podemos generar un token de acceso en "accesJWT" que es el que usamos en la aplicación para permitir acceder a los endpoints o no:

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

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

Si vemos la figura 3.2 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 3.2: 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.




```
{
    "permisos": 1,
    "idUsuari": 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)[7][8].

3.3.2.4. 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 electrónico[9] 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 nuestro requisitos de seguridad[10]: si todo va bien deberemos obtener la figura 3.3:

[7] Se recomienda encarecidamente al lector optar por esta última opción

[8] Al poner las tres clases en anexo se omitieron las funciones main donde se testeaban las funciones de creación de tokens con control de excepciones, comentarios e imports por falta de espacio en el DIN A4.

[9] el correo electrónico lo teníamos guardado en el localStorage de la página de registro inicial.

[10] Mínimo 8 caracteres, una minúscula y una mayúscula y sin caracteres peligrosos.

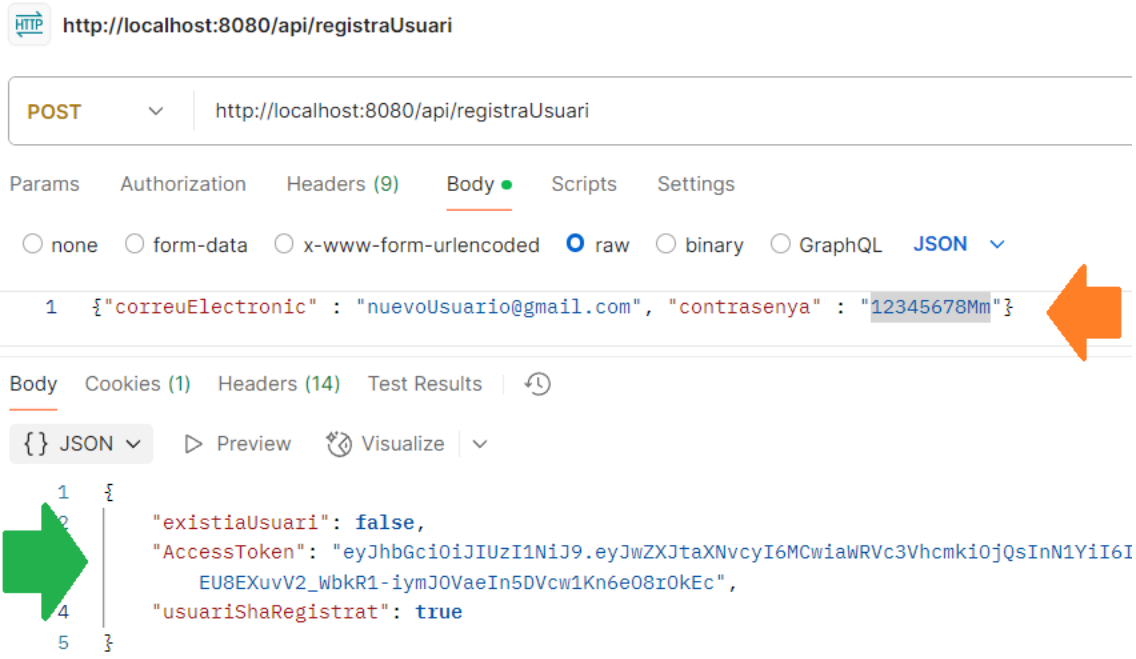


Figura 3.3: Creación de un nuevo usuario llamando con una solicitud POST al endpoint "api/registraUsuari" cuando las validaciones del objeto RegistreDTO del back-end lo permite. En naranja se muestra el body de la petición (lo que el cliente envía al servidor) y en verde el body de la respuesta (lo que 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 URI[11] */api/login*. Si intentamos iniciar sesión con un usuario ya existente en la tabla de usuarios obtendremos algo como esto:

[11] Uniform Resource Identifier

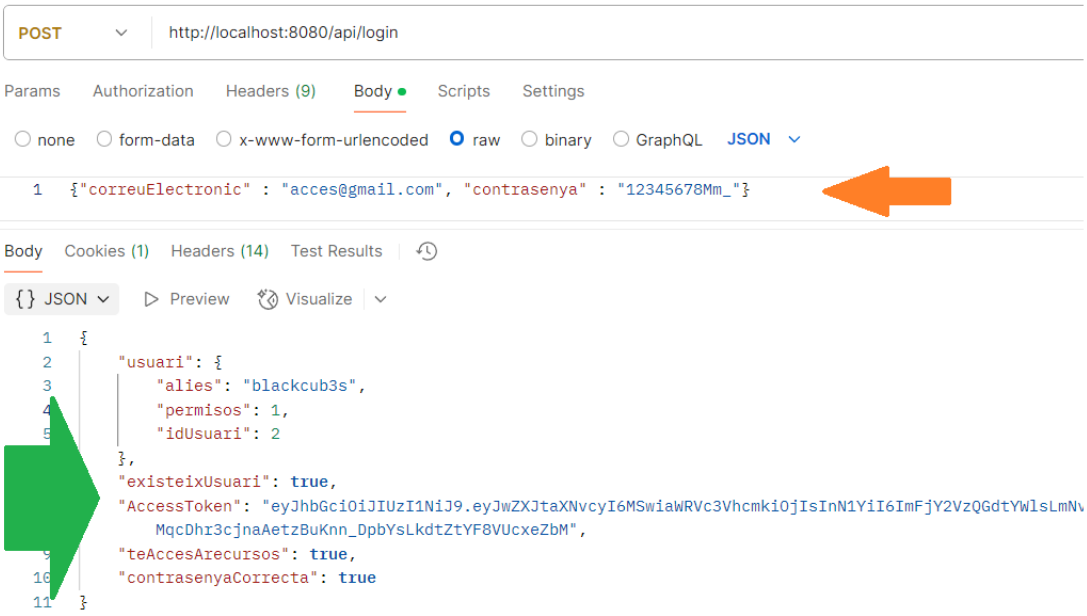


Figura 3.4: Iniciando sesión con un usuario ya existente mediante “api/login” mandando los datos de “correuElectronic” y “contrasenya” que leerá y validará el LoginDTO del back-end. En naranja es el body de la petición y en verde el body de la respuesta.

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 3.5.3

3.3.2.5. 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 3.3.2.3 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 heather “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/tiketsVERIFICARSIEXISTEIX'`) (ver sub-

seccion 3.3.1, 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 linea github-, y de la variable permisos del model de la @Entity class Usuari - ver linea 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 lineas 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 *UsernamePasswordAuthentication-Token* 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)[12].

```
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:

[12] Cuidado! 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 *FiltreAutenticacioJwt.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)[13].
- 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 usuari será el del *principal*[14], 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

[13]JWT 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.

[14]El principal era el idUsuari que pasamos al crear el objeto *UsernamePasswordAuthentitacionToken* 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

3.3.3. 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 3.5.4).

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) a través de **llamadas directas a la API**. Hay que tener mucho cuidado con esto! **TO DO**

3.3.4. 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 EncriptaContrasenyes.java que ha permitido envolver las funciones de bcrypt usadas con nombres más pedagógicos (ver figura 3.5).

Figura 3.5: Clase `EncriptaContrasenyes.java`, utilizada para encriptar contraseñas y comparar hash encriptados.

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

    //PRE: una contrasenya.
    //POST: el hash de la contrasenya.
    public String hashejaContrasenya(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 verificaContrasenya(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 EncriptaContrasenyes.java. Al instanciar un objeto de la misma, se instanciará también un objeto BCryptPasswordEncoder y se usarán las funciones de librería *encode()*[15] y *matches()*[16], respectivamente (convenientemente guardadas con un nombre más agradable como vemos en la figura 3.5).

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

¡Es interesante hacer notar que una misma contraseña encriptada varias veces por la función *encode()* (envuelta en *hashejaContrsenya()*) produce hash distintos! Es decir, el hashing no solo imposibilita 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()*.

3.4. Desarrollo back-end (microservicio con Python)

3.5. Desarrollo del front-end

3.5.1. Enrutamiento de vistas

Cuando un usuario introduzca su correo en el formulario de registro de la página principal de la web (`pas1_LandingSignUp.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 3.5.2.3).

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*.

[15] Para obtener el hash de una contraseña creada.

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

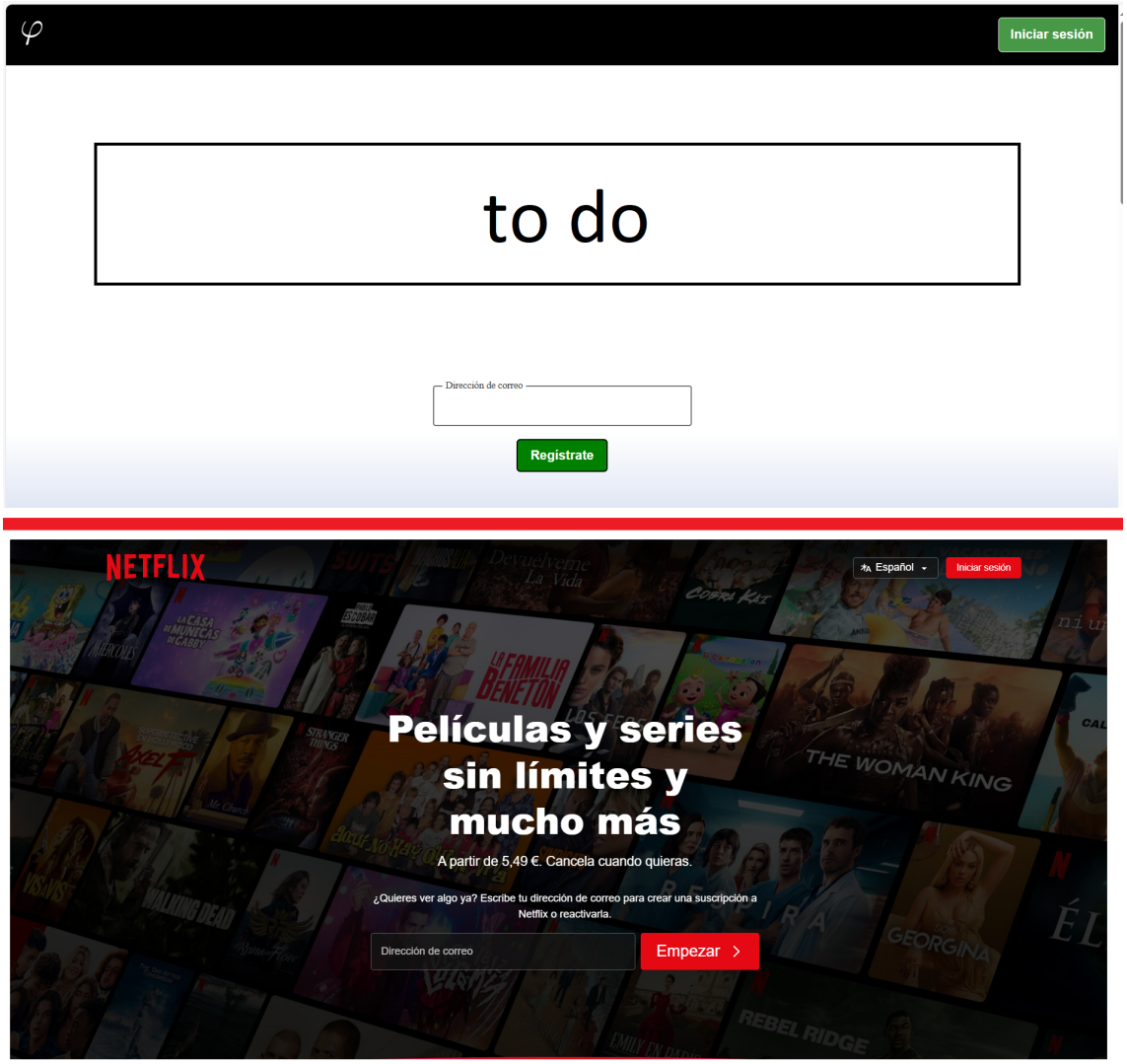


Figura 3.6: **Imagen superior**: Detalle de la landing page `pas1_LandingSignUp.html` donde el usuario introducirá inicialmente su correo para registrarse -aunque ese mismo formulario en realidad nos servirá para todo gracias al enrutamiento de vistas- || **Imagen inferior**: la página de Netflix en 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 tikets.

El proceso de enrutamiento de los usuarios desde que introducen el correo en el formulario de `pas1_LandingSignUp.html` hasta que acceden al *dashboard* se encuentra recogido en el diagrama de la figura 3.7, cuyas nomenclaturas explicamos en los siguientes *bullet points*:

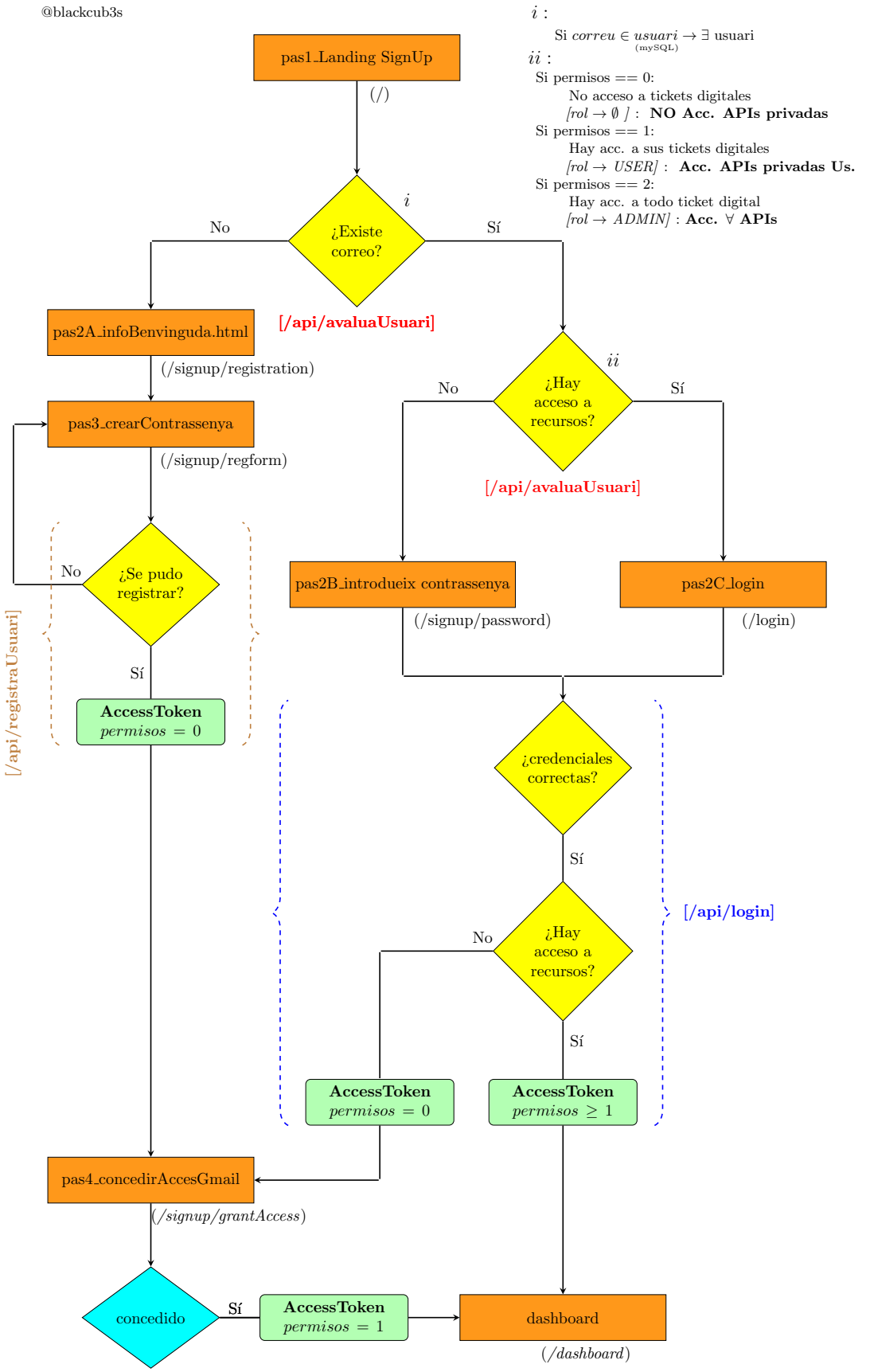
- Decisiones del back-end de Spring Boot al llamar a APIs: **rombos amarillos**.
- Las APIs que consume el front-end van entre corchetes ([]) y con color:
  - [`/api/avaluaUsuari`]: *end-point* que evalúa si el correo electrónico introducido pertenece a un usuario registrado y qué permisos tiene.
  - [`/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*[17].
  - [`/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 redondeados [18].

[17] El lector puede consultar la explicación sobre lo que son las claims y el payload de un JWT en el apartado 3.3.2.2.

[18] Consultad la estructura de los mismos en la figura 3.2.

Figura 3.7: 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` hasta que finalmente obtiene acceso al `dashboard`.

NOTA: **Acceso a recursos $\iff$ permisos ≥ 1**



3.5.2. 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 estáticos[19] 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*[20], y así lo hemos hecho en este proyecto[21]. 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 este[22].

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.

[19] HTML, CSS y JS son archivos estáticos.

[20] Separation of concerns

[21] Si 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.

[22] A 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.

3.5.2.1. 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 3.7, 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 descifrable[23].

Si está expirado, cuando hagamos la diferencia entre el valor "exp" del payload[24] y la función *Date.now()/1000*[25] 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

[23] Ojo, descrifrable no significa validable. Validable es lo que hacemos en el back-end con el secret, y no se muestra jamás en el cliente.
[24] Segundos en que el token expira o expiró, desde la epoch.
[25] Segundos actuales del navegador, desde la epoch.

antes de que se cargue el DOM[26]). 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 3.8):

Figura 3.8: Script `restringeixVistesPrivades_USUARI_NO_LOGUEJAT.js`, utilizado 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'exp
}

//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); //comparem 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(); //aixi la primera crida a la funcio no espera 1 segon
setInterval(redirigeixAlandingPage, 1000); //repetim crides subsequents cada segon
```

[26]El 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*; 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.

3.5.2.2. 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 enrutamiento del front-end de la aplicación (su impacto podemos verlo en la parte superior derecha de la figura del enrutamiento 3.7 y más resumidamente en el cuadro 3.2), 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 3.2: Significado de la variable `permisos` en la tabla `usuaris` 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**[27]. Estas páginas vetadas a los usuarios "logueados" son las siguientes:

- *A)* `pas1_landingSignUp.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"[28] dentro del *payload* del token de acceso guardado en el *local storage* [29], 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:

[27] ¡Si 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!

[28] No hace falta mencionar que se saca del campo permisos de la tabla usuaris de la base de datos que tenemos en mySQL.

[29] Si queréis ver a qué me refiero con el payload, revisad de nuevo la figura 3.2.

página accedida	Permisos token	Redirigimos automáticamente a
A), B, C), D) o *E)*	0	`pas4_concedirAccesGmail.html`
	1||2	`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 3.8 previa, y lo mostramos a continuación en la figura 3.9:

Figura 3.9: 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`.

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

        if (!tokenHaExpirat) {
            if (payload.permisos >=1)
                window.location.href = "dashboard.html";
            else if (payload.permisos == 0)
                window.location.href = "pas4_concedirAccesGmail.html";
        }
    } catch (error) {
        console.log("NO HI HA TOKEN :). Per tant, no cal fer redirecció i deixem l'usuari en la pàgina pública :)");
    }
}

redirigeixApaginaProtegida(); //aixi la primera crida a la funcio no espera 1 segon
setInterval(redirigeixApaginaProtegida, 1000); //repetim crides subsequents cada segon (si queda oberta una pàgina en una altra pestanya ens redirigirà
```

3.5.2.3. 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 visionado de usuarios no logueados (usarios que no tengan token expedido): estos no las podrán ver nunca, porque hicimos el script de la figura 3.8 para redirigilos 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 tikets; 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 3.10:

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

Figura 3.10: Script `restringeixVistesPrivades_USUARI_LOGUEJAT.js`, utilizado para redirigir a un usuario logueado hacia `pas4_concedirAccesGmail.html` o `dashboard.html`, según proceda, en caso que el usuario entre en una de estas dos páginas privadas sin el permiso correspondiente.

```
function redirigeix_a_dashboard_o_pas4() {
    try {
        const payload = decodificaPayloadJWT(localStorage.getItem("AccessToken"));
        let secFinsExpiracio = payload.exp - (Date.now()/1000); //comparem 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 3.5.3 que viene a continuación, donde especificamos como se recibe el token del back-end y se guarda en el front-end.

3.5.2.4. 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 sustitye 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"[30].

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 3.8). 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 3.11: 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");
    });
});
```

3.5.3. Recibir el Access Token desde el back-end

Cuando un usuario del que ya tenemos su correo electrónico en BBDD intenta "loguearse" en mercApp[31], el token de acceso lo recibe por primera vez en el

[30] A 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.

[31] Sea 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_introdueix contrassenya.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: [32] Para más información sobre el código JavaScript del front-end que lo permite véase figuras y 3.12 y 3.13):


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


Figura 3.12: Fragmento de codigo en `pas2C_login.html` dentro del codigo javascript para manejar codigos 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 REPSOSTA ES EXITOSA (NO HA DONAT CODIS D'ERROR).
.then(response => {
    // Verificar si la resposta es exitosa.
    if (response.status == 200)
        return response.json();          // Convertir la resposta a JSON
    else if (response.status == 400)
        return response.json().then(data => Promise.reject({ type: 'validacion', data }));
    else {
        throw new Error('Error! La resposta de xarxa no ha sigut exitosa!');
    }

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

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

Figura 3.13: Fragmento de codigo en pas2C_login.html dentro del codigo javascript para obtener el token de acceso (detalle en rojo).

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

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

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

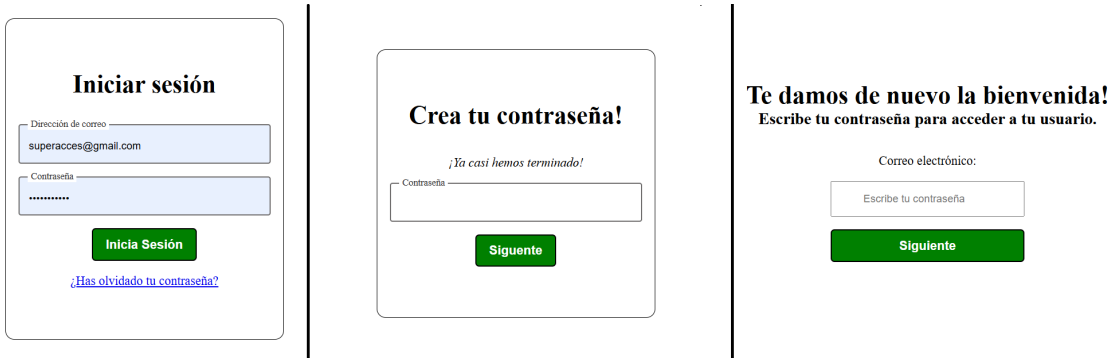
Ahora bien, si el usuario nunca se ha registrado en nuestra aplicación[33], cuando lo haga, lo hará por el proceso de registro y no por el proceso de inicio de sesión. Una vez introduzca su contraseña en `pas3_crearContrasenya.html` y se tome el correo electrónico insertado por el usuario en páginas 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 ahi 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 3.3.2.4, 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 embedidos que pueden hacer llamadas asíncronas y obtener un token de acceso del servidor y guardarlo en el localStorage (un detalle de los formularios de estas páginas se encuentra en la figura 3.14):

[33]Es decir, no tenemos su correo electrónico en BBDD.

Figura 3.14: Detalle de los formularios que permiten generar llamadas a los dos endpoints generadores de tokens de acceso (/api/login y /api/registraUsuari). De izquierda a derecha las páginas que los contienen: `pas2C_login.html`, `pas3_crearContrasenya.html` y `pas2B_introduirContrasenya.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.

3.5.4. validacion 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 3.3.3).

TO DO FER-HO

3.5.5. Diseño responsive: paginas publicas

TO DO FER-HO

3.5.6. Diseño responsive: paginas privadas

3.5.7. arquitectura dashboard: llamada fetch inicial a back-end y posteriores a localStorage

TO DO FER-HO

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 `HttpOnly` y `Secure`, ya que no es accesible desde JavaScript.	Más seguro si la cookie tiene `HttpOnly` y `Secure`, ya que no es accesible desde JavaScript.
Seguridad contra CSRF	Puede ser vulnerable si la cookie no tiene `SameSite=Strict`.	Menos vulnerable si la cookie tiene `SameSite=Strict`.
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 `HttpOnly`.	No accesible desde JavaScript si la cookie tiene `HttpOnly`.

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'acces com el de refresh)
        //veure debat http://bit.ly/3RmBGIK
        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'acces 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 unic per a token. Per traSSSabilitat
                .setSubject(correu)           //guardo nom subjecte (dins "sub")
                .setIssuedAt(new Date())      //data creacio
                .setExpiration(new Date(System.currentTimeMillis() + tExpDies*86400*1000))  //expiracio
                .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 exprirar el token

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

        //FINALITAT: Generar un JWT d'acces.
        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() / 1000 + (tExpM * 60)) * 1000))
            .signWith(SignatureAlgorithm.HS256, clauSecreta.getBytes())
            .compact();
    }
}
```

5.4. Clases de seguridad

5.4.1. Clase ConfiguracioSeguretat.java

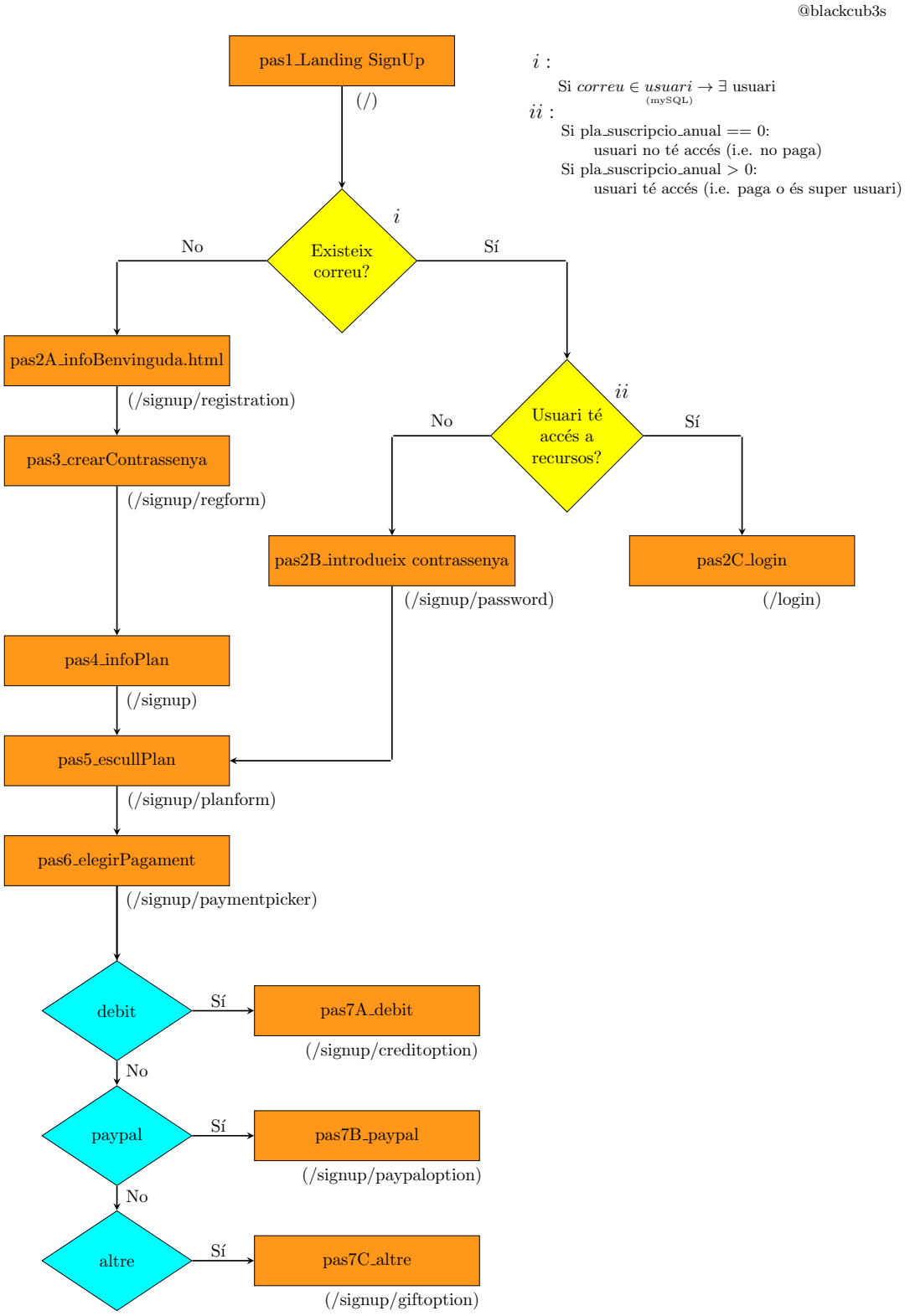
```
package miApp.app.seguretat;


@Configuration
@EnableMethodSecurity
@PreAuthorise
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 administardor 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>> actualitzaContrasenya(
        @PathVariable("id") int id, @Valid
        @RequestBody ActualitzaContrasenyaDTO dto) {
        Optional<Usuari> usuariActualitzatOPTIONAL =
         serveiUPP.actualitzaContrasenya(dto, id);
        HashMap<String, String> resposta = new HashMap<>();
        if (usuariActualitzatOPTIONAL.isPresent()) {
                resposta.put("mensaje", "Contrasena actualizada correctamente.");
                return new ResponseEntity<>(resposta, HttpStatus.OK); //200 OK
        } else {
                resposta.put("mensaje", "Usuario no encontrado.");
                return new ResponseEntity<>(resposta, HttpStatus.NOT_FOUND);
//404 NOT FOUND
        }
}
```

5.5. Diagrama réplica netflix



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

Este diagrama se puede entender del siguiente modo:

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

5.6. Diagrama enrutamiento mercApp

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

- **Responsabilidad única (SRP)**: Cada módulo debería hacer una sola cosa ({Frontend → interfaz}, {Backend → procesamiento}).
- **mantenibilidad**: Al usar una arquitectura modular cada parte puede evolucionar por separado. Podemos desplegar solo el front o el back. En el futuro podremos cambiar el front-end de archivos estáticos por un front-end con Angular, por ejemplo.
- **Escalabilidad**: Según la carga podemos escalar independientemente ambas partes del proyecto. Por ejemplo, poner los archivos estáticos (html, css y js del front-end) en un servidor para servirlos rápidamente como nGinx [13] (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.

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] NGINX. Nginx — high performance load balancer, web server, & reverse proxy. https://www.nginx.com/. Accedido el 8 abril de 2025.