

Creación de un dashboard para usuarios del ticket digital de Mercadona con consultas visuales a la evolución de 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

Índice general

1. Introducción	3
1.1. ¿Qué es el ticket digital de Mercadona?	3
1.2. Identificación de necesidades	3
1.3. Objetivos del proyecto	3
2. Diseño del proyecto	5
2.1. Requisitos Funcionales	5
2.1.1. Requisitos de la aplicación	5
2.1.2. Requisitos de los usuarios	6
2.2. Stack tecnológico	6
2.2.1. Front-End: HTML, CSS y Javascript	6
2.2.2. back-end: Java (SpringBoot) y Python	6
2.2.3. BBDD: MySQL y MongoDB	7
2.3. Diagramas de la aplicación	7
3. Desarrollo del proyecto	8
3.1. GitHub del proyecto	8
3.2. Entornos de desarrollo	8
3.3. desarrollo del front-end	9
3.3.1. Manejar vistas en función de Autenticación y autorización	9
3.3.1.1. Protegiendo las vistas	9
3.3.1.2. Recibir el Access Token desde el back-end	10
3.3.2. validacion de datos (Formularios entrada)	11
3.4. desarrollo back-end (Spring Boot)	11
3.4.1. Estructura de la aplicación	11
3.4.2. Autenticacion y Autorización	12
3.4.2.1. método utilizado: JWT	12
3.4.2.2. ¿Qué compone un JWT?	13
3.4.2.3. Implementación de JWT en java SpringBoot	14
3.4.2.4. Enviar por primera vez el Access Token hacia el front-end	16
3.4.2.5. Recibir el JWT enviado por el front-end, interpretarlo desde el back-end y con ello securizar un endpoint de la API desde el back-end	16
3.4.3. validación de datos (End-points back)	19
3.5. desarrollo back-end (microservicio con Python)	20
4. Evaluación y Conclusiones Finales	21

5. ANEXO	22
5.1. Flujo de trabajo habitual en git	22
5.2. Diferencias de seguridad: JWT vs SESSIONID en cookies seguras	23
5.3. Clases para crear y verificar JWTs	24
5.3.1. Clase JWT	24
5.3.2. Clase Refresh Token	24
5.3.3. Clase Access Token	24
5.4. CODIGO 5	25
7. Bibliografía	25

Introducción

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 son evidentes: no se pierden los tickets de cara a devoluciones y el cliente del supermercado no debe esperar a la impresión del ticket.

1.2. Identificación de necesidades

Los tickets de cada usuario se acumulan de forma recurrente en el correo electrónico y con un formato estructurado (los asuntos son predecibles e incluyen las fechas) y dentro de cada correo de un ticket digital se encuentra un PDF con el desglose de la compra (producto, unidades vendidas, establecimiento, etc).

Esta información se acumula en el correo del usuario pero a pesar de ser una información estructurada su acceso para el usuario no es simple: no puede visualizar lo que ha gastado, ni el precio de los productos y de su evolución, ni los supermercados en los que ha comprado, ni las veces que lo ha hecho, etc.

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” front-end 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.

Los **Objetivos principales** del proyecto son mostrar al usuario:

- **Evolución de precios** (inflación) a lo largo del tiempo en los productos

habitualmente comprados en el mismo establecimiento¹.

- **Evolución del gasto** total del usuario a lo largo del tiempo.

¹La 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

Para implementar los objetivos principales de los que hemos hablado en la sección [1.3](#) hemos proyectado una serie de requisitos funcionales de la aplicación.

2.1. Requisitos Funcionales

NOTA: Los requisitos aquí presentes se suman a los requisitos que se sobreentiende que tiene la aplicación de un proyecto final de grado superior y que esta aplicación, por supuesto, cumple: tener un front-end, un back-end con sistema de registro de usuarios y un login con buenas prácticas en materia de seguridad y una base de datos.

2.1.1. 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: 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 queso (o de sectores) desglosando qué porcentaje del dinero se ha destinado a cada una de las siguientes categorías: verdura y hortalizas, frutas, huevos y lácteos, agua y bebidas, aceite y especias, carnes, pescado, hogar e higiene personal. 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.

REQUISITO D: Un botón “Actualizar” permitirá al usuario refrescar los datos desde el servidor cuando haya añadido nuevas compras. También podríamos permitir

que los PDFs descargados en el servidor se almacenen en una carpeta local del usuario para que pueda verificar la extracción de los datos.

REQUISITO E: Hacer un sistema front-end y back-end que permitan redirigir a los usuarios rápidamente a un registro de forma inteligente. Nos inspiraremos en el sistema de registro e iniciar sesión de Netflix (**POSAR DIAGRAMA A L'ANEX SOBRE EL SISTEMA DEL NETFLIX I EL NOSTRE**)

2.1.2. 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.

2.2. Stack tecnológico

Hemos escogido un stack tecnológico que permite que seamos fieles a los requisitos funcionales que nos hemos marcado para la aplicación:

2.2.1. Front-End: HTML, CSS y Javascript

Se ha usado HTML, CSS y JavaScript. - Para la visualización de gráficos se usará una librería javascript: <https://www.chartjs.org/>. Entre otras cosas se utilizará para hacer los gráficos de la evolución de precios por producto.

2.2.2. back-end: Java (SpringBoot) y Python

- Back-end con Java (springboot para el login y la autenticación de usuarios: con este framework guardaremos datos en la BBDD mySQL).

- 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.

2.2.3. 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 por que 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. La primera vez que consulte sus datos, estos se descargarán y almacenarán en localStorage del cliente. En consultas posteriores, los datos se obtendrán directamente de localStorage sin necesidad de hacer peticiones al servidor. Evaluaremos la viabilidad de este sistema durante el desarrollo; Esto es la fase de diseño y como tal, **PUEDE QUE EN LA FASE DE DESARROLLO CAMBIE**, en caso de no ser factible, las consultas se harán directamente en MongoDB.

2.3. Diagramas de la aplicación

Fer un diagrama guapo de tots els components de l'aplicació.

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” de sus componentes en vuestro ordenador personal si así lo deseáis.

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

Cuadro 3.1: Enlaces importantes 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.

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).

3.3. desarrollo del front-end

3.3.1. Manejar vistas en función de Autenticación y autorización

Podemos considerar que cada archivo HTML y su CSS asociado es una “vista” de nuestra aplicación. La autenticación y la autorización se hace en el backend y el control de las vistas (en función de si el usuario está autorizado y autenticado) se hará en el front-end mediante JavaScript puro.

Si bien SpringBoot permite servir los archivos estáticos¹ de dentro del mismo back-end de Spring Boot y utilizar un sistema de plantillas (Thymeleaf) esto realmente no es, para nada, lo ideal. Lo ideal es definir un front-end y un back-end separados y así lo hemos hecho en este proyecto².

Al tener el front-end y el back-end desacoplados esto también aumenta la complejidad inicial en el desarrollo: la protección de las vistas se hace más compleja. Así las cosas, del mismo modo que los endpoints de nuestra API del back-end en SpringBoot 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), ciertas páginas del front-end no podrán obtener la información deseada: En este caso 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 de login, por ejemplo.

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.

3.3.1.1. Protegiendo las vistas

TO DO: POSAR EXEMPLE DE CASOS PARTICULARES

¹HTML, CSS y JS son archivos estáticos.

²Si 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.

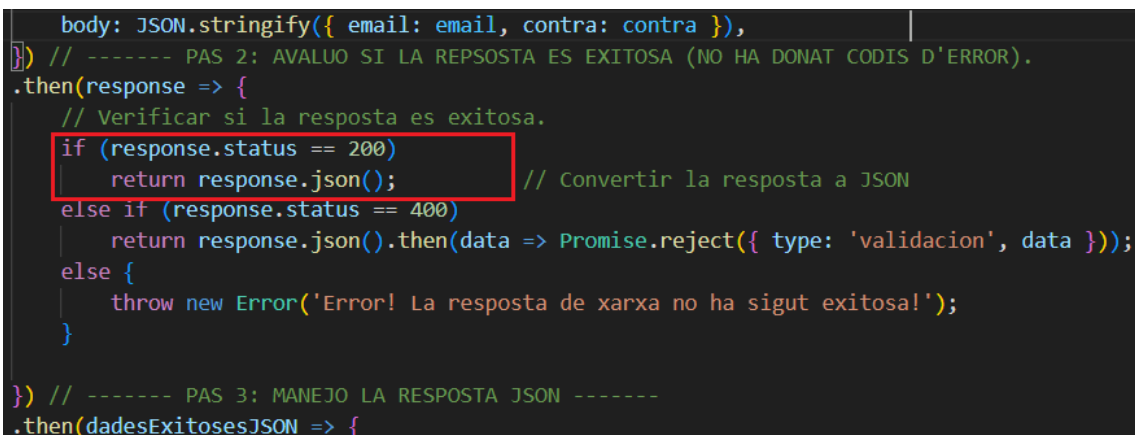
3.3.1.2. Recibir el Access Token desde el back-end

El token de acceso se recibe por primera vez en el cliente cuando el usuario se **loguea con éxito** mediante la llamada `fetch()` hacia el endpoint del back-end `"/api/login"`.

Esta recepción se produce cuando, por un lado, en el JavaScript embebido, dentro de *pas2Clogin.html*, obtenemos el código 200 (OK) del servidor, pero también se cumple que usuario y contraseña son correctos. Si y solo si se cumplen ambas condiciones, se extrae el token de acceso del body que obtenemos, que será uno como este, y podremos guardarlo en el LocalStorage del navegador³ (véase figuras y 3.1 y 3.2):

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

Figura 3.1: Fragmento de código en *pas2C_login.html* dentro del código javascript para manejar códigos de error. Cuando el back-end de SpringBoot devuelve el código 200 significará que podremos extraer los datos del body de la. 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 respuesta es exitosa.
  if (response.status == 200)
    return response.json(); // Convertir la respuesta a JSON
  else if (response.status == 400)
    return response.json().then(data => Promise.reject({ type: 'validacion', data }));
  else {
    throw new Error('Error! La respuesta de xarxa no ha sigut exitosa!');
  }
}) // ----- PAS 3: MANEJO LA RESPUESTA JSON -----
.then(dadesExitosesJSON => {
```

³Esto lo hacemos para luego poder mandarlo de vuelta al servidor en la subsecuentes solicitudes que requieran autentización y autorización.

Figura 3.2: Fragmento de código en pas2C_login.html dentro del código javascript para obtener el token de acceso (detalle en rojo).

```

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

    console.log(dadesExitosesJSON); // Imprimir la respuesta en la consola (treure a produccio: conte el

    if (dadesExitosesJSON.existeixUsuari) {
        if (dadesExitosesJSON.contrasenyaCorrecta) {
            if (dadesExitosesJSON.teAccesArecursos) {
                let tokenAccesJWT = dadesExitosesJSON.AccessToken; //extrec el token que anava al body
                localStorage.setItem("AccessToken", tokenAccesJWT); //guardem token al localStorage

                bannerAlerta([], "bienvenidoAlaApp", "var(--verdAlerta)");
                setTimeout(() => {window.location.href = "/dashboard.html";}, tEspera); //ENVIO USUARI
            } else {} //NO TE ACCES A RECURSOS (USUARI TE CONTA I CONTRASNEYA)

            bannerAlerta([email], "usuariExisteixPeroNoteAccesArecursos", "var(--lilaAlerta)");

```

Para entender de dónde viene el token desde el back-end redirigimos al lector a la sección 3.4.2.4, donde se trata ese aspecto. En la presente sección nos ocuparemos de JavaScript en el front.

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”.

3.3.2. 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 3.4.3).

TO DO FER-HO

3.4. desarrollo back-end (Spring Boot)

3.4.1. Estructura de la aplicación

TO DO pom.xml application.properties, usuari, repository, service, controller.

TO DO Parlar de les validacions i les anotacions

Parlar dels apartats de seguretat (security filter chain i JWT pero només de passada perquè els mencionem després)

3.4.2. Autenticacion y Autorización

3.4.2.1. método utilizado: JWT

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

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

A este sistema lo llamamos JSON Web Token (JWT) y toda la información que contiene está encriptada o **firmada digitalmente** con SHA256 mediante una clave privada que solo tenemos nosotros en el servidor. Esta clave es igual para todos los tokens que generemos: la firma digital que emana de esta clave estará embebida, por así decirlo, en cada uno de esos tokens y *será inválida* si un atacante ha modificado el token y nos lo devuelve al servidor tratando de suplantar la identidad de algún usuario; con ello, el servidor rechazará la integridad del token y 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!). La solución que se propone con JWT es tener, por lo tanto, dos tokens almacenados en el cliente: el “access token” que es el que permite autenticar y autorizar, del que hemos hablado hasta ahora; y el “refresh token”, que tiene la función de validar si

el token de refresco sigue siendo válido con el servidor, y si lo es, entonces permite que el servidor emita un nuevo “access token” para el usuario. De este modo el usuario no deberá estar generando nuevos tokens constantemente. En este trabajo solo utilizaremos “access tokens” y ya está. Cuando un usuario se desloguee, lo que haremos será borrar el access token de la cookie, local storage o session storage y ya está.

En resumen, **las ventajas** que tiene JWT vs uso de sesiones (assumiendo que tanto JWT como SESSID se guarden en una cookie segura) son las siguientes:

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

Y la **desventaja** más evidente que tiene JWT, es, en nuestra opinión, su complejidad⁴, 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**

3.4.2.2. ¿Qué compone un JWT?

El JWT se compone de tres partes. **los headers, el payload y la signatura**. En la página <https://jwt.io/>, como veremos después, se puede ver si los tokens son válidos, observar su estructura interna, etc. [1]. A saber:

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

⁴Se puede ver una tabla de diferencias más en profundidad, especialmente en materia de seguridad en el anexo 5.2)

secreta que solo conoce el servidor (es lo que permitirá al servidor rechazar el token si no es válido -i.e. ha sido manipulado).

Podéis observar estas tres partes en colores en la figura 3.3 que veremos después.

3.4.2.3. Implementación de JWT en java SpringBoot

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

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

En el proyecto se han creado tres clases dentro de sus respectivos archivos en la ruta `src/main/java/ miApp.app/seguretat/jwtseguretat` 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:

```
@Component
public class AccessToken extends JwtUtil {

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

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

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

```

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

```

Con la función **genera()** de la clase `AccessToken` arriba mostrada, y con los parámetros necesarios que serán necesarios para autorizar (`idUsuari`) y autenticar (permisos), podemos generar un token de acceso en “`accesJWT`”:

```

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

```

Si vemos la figura 3.3 que tenemos a continuación, veremos en la mitad izquierda un token de acceso generado por la función anterior. Fijémonos que internamente ese token está estructurado en las tres partes de la parte derecha de la imagen, siendo la payload la más importante:

Figura 3.3: Decodificación de un token de acceso generado con la función `genera` de la clase `AccessToken` mediante jwt.io. La Payload con las claims en flecha verde.

Encoded PASTE A TOKEN HERE

```

eyJhbGciOiJIUzI1NiJ9.eyJwZXJtaXNncyI6MSwiaWRVc3VhcmkiOiJIsInN1YiI6InNhbnRvQGdtYWlsLmNvbSIsImIhdCI6MTc0MzE1MjM1MCwiZXhwIjoxNzQzMjUyOTUwZmQ.ixC6NKKuG99zrxLcrgxfjRRKi8NgYbUrirFgPMEcUC0

```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256"
}
```

PAYLOAD: DATA

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



VERIFY SIGNATURE

```

HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-a8f7d9g0b6c3r
) ☐ secret base64 encoded

```



```

{
    "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 [3]. 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). Se recomienda encarecidamente al lector optar por esta última opción⁵.

3.4.2.4. Enviar por primera vez el Access Token hacia el front-end

Cuando el usuario consigue poner la contraseña correcta en la página del front-end **pas2C_login.html** se manda de vuelta del back-end (desde el controlador de endpoint `/login`) hasta el front-end en el `localStorage` (¡por ahora lo hemos hecho así por simplicidad, pero en producción debería estar en una cookie segura!).

La parte de recepción del token en el front-end y de su manejo podéis encontrarla en el apartado 3.3.1.2

TO DO: Referenciar endpoint login springboot (que cal millorar)

3.4.2.5. Recibir el JWT enviado por el front-end, interpretarlo desde el back-end y con ello securizar un endpoint de la API desde el back-end

Después de crear las tres clases en Java de las que hemos hablado en el apartado 3.4.2.3 anterior y haber mandado el token de acceso al front, 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 acaba

⁵Al 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.

de recibir su token y quiere acceder al dashboard de visualización (a través de la heather “Authorization”). La solicitud HTTP con el susodicho token se hace via GET a (`'usuaris/id/ticketsVERIFICARSIEXISTEIX'`) (ver subseccion 3.4.1, en `ControladorUsuari.java`).

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

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

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

Estos dos pasos irán en función del valor de la variable que haya emanado de la base de datos al conceder el token mediante **idUsuari** para el caso de la **Autenticación** -ver [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` [2], que como dice el propio nombre de la clase implementa un filtro que se desarrollará una y solo una vez por cada petición al servidor.

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

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

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

“Authorization” : “Bearer OJALWQ03P1WNOEGBO...”

La programación necesaria para conseguir lo mencionado en el párrafo anterior queda recogida en este rango de líneas de GitHub ([ver rango](#)).

Luego una vez tenemos el token dentro de Spring Boot tratamos de sacar las Claims del Payload, es decir, la carga útil del token ([ver rango](#)). Y con ello ya podemos asignar roles a partir de la variable permisos del payload (es decir, 0 1 o 2 en función de si no tiene ninguna autorización todavía, 1 si ya tiene acceso a los recursos de la web como podría ser consultar el dashboard de la aplicación y 2 si es superusuario - [Ver rango](#) -). En definitiva, lo de éste párrafo (pos si se imprimió la memoria y no hay acceso a github) sería esto:

```
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 la el 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 código](#)).

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

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

Este constructor nos permitirá restringir permisos para las APIs según el **idUsuari** al que esté vinculado su login (autenticación) y también según el valor de **permisos** que tenga (autorización)⁶.

```
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

lorem ipsum

PASO 3: Restricciones en el controlador

lorem ipsum

3.4.3. validación de datos (End-points back)

NOTA: Los datos validados en el back-end siguen las mismas expresiones regulares y restricciones que las validaciones hechas en el front-end (ver sección 3.3.2).

⁶Cuidado! 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.

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

3.5. desarrollo back-end (microservicio con Python)

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 SESSION en cookies seguras

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

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

5.3. Clases para crear y verificar JWTs

5.3.1. Clase JWT

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

    //es la clau privada de 256 bits com a minim per encriptar el token (tant el d'accés com el de refresh)
    //veure debat http://bit.ly/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'accés que genera generaAccesToken().
    public String generaRefreshToken(String correu, int idUsuari) {
        Map<String, Object> dadesExtraApayload = new HashMap<>();
        dadesExtraApayload.put("idUsuari", idUsuari);
        //posar mes dades al payload si es necessari

        return Jwts.builder()
            .setClaims(dadesExtraApayload) //dades customitzades
            .setId(String.valueOf(UUID.randomUUID().toString())) //id 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'accés.
    public String genera(String correu, int idUsuari, byte permisos) {
        Map<String, Object> dadesExtraApayload = new HashMap<>();
        dadesExtraApayload.put("permisos", permisos);
        dadesExtraApayload.put("idUsuari", idUsuari);
    }
}
```

```

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

```

5.4. CODIGO 5

Bibliografía

- [1] jwt.io. Json web tokens - jwt.io. <https://jwt.io/>. Accedido el 27 marzo de 2025.
- [2] 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.
- [3] 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-secret>. 2020. Accedido el 28 marzo de 2025.