UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

STUDY OF AUTHENTICATION MECHANISMS IN DISTRIBUTED APPLICATIONS

Treball de fi de Grau

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
per
Oriol Raventós Plans

GRAU EN ENGINYERIA DE TECNOLOGIES I SERVEIS DE TELECOMUNICACIÓ

Especialitat de
Sistemes Telemàtics

Director: Marcel Fernández Muñoz

Barcelona, Juny 2019

Resum

En aquest projecte he realitzat un estudi sobre l'autenticació basada en tokens i he desenvolupat un sistema basat en microserveis que consta de tres servidors. Els usuaris s'autentifiquen utilitzant l'estàndard JWS JSON WEB SIGNED.

Aquest estàndard utilitza un format de JSON que consta de tres blocs: header, payload i signature. Crea dos JSON de les dues primeres parts, i les codifica en Base64 separades per un punt. Després utilitza tota aquesta cadena per crear una firma digital, utilitzant encriptació de clau pública / privada que serà la signature, i també és codificada en Base64.
Un token és una cadena codificada, seguint aquest patró:
headerOnBase64.payloadOnBase64.signatureOnBase64
El resultat és una cadena com aquesta:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

La part del front-end està feta amb React-Redux i escrita amb TypeScript. He intentat ser el màxim de fidel a la programació funcional.

Resumen

En este proyecto he realizado un estudio sobre la autenticación basada en tokens y he desarrollado un sistema basado en microservicios que consta de tres servidores. Los usuarios se autentican utilizando el estándar JWS JSON WEB SIGNED.

Este estándar utiliza un formato de JSON que consta de tres blocs: un header, un payload y un signature. Crea dos JSON en las dos primeras partes, y las codifica en Base64 separadas por un punto. Después utiliza toda esta cadena de caracteres para crear una firma digital, utilizando encriptación de clave pública / privada para crear la signature, que también la codifica en Base64.
Un token es una cadena codificada, siguiendo este patrón:
headerOnBase64.payloadOnBase64.signatureOnBase64
El resultado es una cadena como esta:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

El front-end está desarrollado con React-Redux y escrito en TypeScript. He intentado ser fiel a la programación funcional.

Abstract

In this project I have carried out a study on token-based authentication and I have developed a system based on micro services that consists of three servers. Users are authenticated using the JSON WEB SIGNED standard.

This standard uses a JSON format that has three blocks: a header, a payload and a signature. Create two JSONs in the first two parts, and encode them in Base64 separated by a point. Then it uses all this string of characters to create a digital signature, using public / private key encryption to create the signature, which also encodes it in Base64.
A token is an encoded string, following this pattern:
headerOnBase64.payloadOnBase64.signatureOnBase64
The result is a string like this:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

The front-end is developed with React-Redux and written in TypeScript. Trying to be the maximum of fidelity to functional programming.

Dedicació

A tots els companys de carrera que he anat coneixent durant aquests anys. També als companys de feina Egoitz, Jordi i Jonatan, ja que amb ells vaig aprendre molt en molt poc temps quan feia les pràctiques a Mango. I a l'Eric , que tot i no estar en el meu equip sempre em va ajudar en tot.

Agraïments

Un agraïment molt especial a l'Eric Torres de Codurance el qual vaig conèixer durant les pràctiques a Mango. Ell em va ajudar a plantejar bona part del back-end, que és on es creen i firmen els tokens.

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Historial de revisions i registre d'aprovació

Revision	Date	Purpose
0	05/03/2019	Document creation
1	11/04/2019	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Oriol Raventós Plans	o.raventos.89@gmail.com
Marcel Fernández Muñoz	marcelf@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	15/06/2019	Date	25/06/2019
Name	Oriol Raventós Plans	Name	Marcel Fernández Muñoz
Position	Project Author	Position	Project Supervisor

Índex

Abstract 4
Dedicació 5
Agraïments 6
Historial de revisions i registre d'aprovació 7
Índex 8
Índex de figures 9
1. INTRODUCCIÓ 10
2. ESTAT DE L'ART DE LES TECNOLOGIES APLICADES EN AQUEST PROJECTE 12
2.1 MAVEN 12
2.2 SPRING BOOT 13
2.3 MONGODB 16
2.4 TYPESCRIPT 16
2.5 NODE 17
2.6 REACT 18
2.7 REDUX 19
2.8 GITHUB 20
3. METODOLOGIA / DESENVOLUPAMENT DEL PROJECTE 22
3.1 FORMAT DEL JWS 22
3.2 SISTEMA 24
3.3 IDPROVIDER 25
3.3.1 Base de dades 25
3.3.2 Contrasenyes dels usuaris 26
3.3.3 Magatzem de claus 26
3.3.4 Creació del JWT 27
3.3.5 Seguretat web 30
3.3.6 API RESTful 34
3.3.7 JSON 35
3.4 GEOMETRIC-APP 37
3.4.1 React 39
3.4.2 Redux 42
3.4.3 Polimorfisme en TypeScript 49
3.5 GEOMETRIC-RESOURCES 51
3.5.1 Base de dades 51
3.5.2 API RESTful 52
4. RESULTATS 53
5. PRESSUPOST 54
6. CONCLUSIONS 55
7. BIBLIOGRAFIA 56
8.1 DIAGRAMAS UML idProvider. 57
8.2 DIAGRAMES UML de geomètric-resources 63
8.3 DIAGRAMES DE SEQUENCIA 67
9. GLOSSARI 70

Índex de figures

Imatge 1: pom.xml 13
Imatge 2: start.spring.io 14
Imatge 3: esquelet projecte idprovider 15
Imatge 4: exemple de tipat en atributs 17
Imatge 5: exemple de tipat en funcions 17
Imatge 6: exemple d'expresio JSX 18
Imatge 7: exemple utilitzant createElement() 18
Imatge 8: exemple compilat amb Babel 19
Imatge 9: flux unidirecional de Redux 19
Imatge 10: Funcionament dels directoris locals de git 20
Imatge 11: header JWT 22
Imatge 12: payload JWT 22
Imatge 13: JWT codificat Base64 23
Imatge 14: esquema del sistema complet 24
Imatge 15: exemple MongoDB del idProvider 25
Imatge 16: bytes per un ObjectId 25
Imatge 17: application.properies del idProvider 27
Imatge 18: JwtConfiguration 27
Imatge 19: JwtProvider part 1 28
Imatge 20: JwtProvider part 2 29
Imatge 21: JwtProvider part 3 30
Imatge 22: WebSecurityConfiguration de idProvider 31
Imatge 23: WebSecurityConfiguration de idProvider 2n part 32
Imatge 24: JwtAuthenticationFilter de idProvider 33
Imatge 25: AuthController de idProvider 34
Imatge 26: esquelet del projecte geometric-app 37
Imatge 27: Arbre dels Components 40
Imatge 28: exemple de React.FunctionalComponent 41
Imatge 29: index.tsx de geometric-app 42
Imatge 30: funcio dispatch de Redux 44
Imatge 31: ThunkAcction de Redux 45
Imatge 32: Acction de Redux 46
Imatge 33: exemple de reducer de Redux 47
Imatge 34: store de Redux 48
Imatge 37: exemple de la MongoDB de geometric-resources 51

1. INTRODUCCIÓ

Tenia clar que volia fer un projecte de software, i la meva idea era fer-lo a través de l'empresa on estava fent pràctiques, però no ens vam entendre i no va ser possible. Un dia vaig anar a parlar amb el Marcel i em va proposar fer un estudi sobre l'autenticació a través de tokens. Més que un estudi, he acabat fent una implementació d'un sistema d'autenticació amb JSON WEB TOKEN, basat en una arquitectura de microserveis i buscant les tecnologies més modernes possibles.
A mi personalment m'agrada programar i m'ha agradat molt fer aquest projecte, amb la llibertat que m'ha deixat el Marcel, que tot i no estar d'acord amb tot el que he fet, sempre m'ha deixat fer, i no m'ha posat cap inconvenient. Vam començar parlant de la programació funcional i vaig estar fent un tutorial de Haskell i anant a unes classes del professor Jordi Forga on també explicava aquest llenguatge, que és un dels pocs llenguatges funcionals purs. He intentat aplicar sempre tot el que vaig aprendre de Haskell i la programació funcional a Java que evidentment no es un llenguatge funcional pur, però amb bones pràctiques pots fer que es comporti de manera semblant, i també en tota la part de TypeScript que és molt millor que Java per fer un codi funcional.

Mai havia treballat amb bases de dades No-SQL i ara hi he pogut treballar per primera vegada.
També he desenvolupat des de zero un front-end, i això m'ha permès aprendre TypeScript, React i Redux.

L'autenticació basada en tokens ha guanyat importància en els últims anys degut a l'augment de les Single Page Applications (SPA), les web API, l'estructura de microserveis i l'internet de les coses (IoT).

Quan parlem d'autenticació amb tokens normalment es parla de JSON Web Tokens (JWT), que és el que s'ha convertit en l'estàndard. JWT té diferents formats: JWS (JSON WEB SIGNED) que és en el quin he basat el meu TFG o el JWE JSON WEB ENCRIPTED que és quan totes les dades van encriptades perquè només el destinatari final sigui capaç de llegir el token. Tan JWS com JWE són especificacions de JWT. A partir d'aquí utilitzaré el terme JWT o token per referir-me a JWS utilitzat en el projecte.

El sistema sencer està format per tres servidors: dos back-end -cada un gestionant una base de dades diferent- i un front-end. Aquest sistema es pot dir que és un sistema de microserveis, ja que cada part està encarregada de resoldre un servei del sistema. Tot el sistema hauria d'utilitzar el protocol HTTPS per encriptar les comunicacions, però no he desenvolupat aquesta part; ja que és necessari tenir una Autoritat Certificadora. Per fer les proves en local no té importància, però sí que seria imprescindible si això fos un sistema desplegat en producció. Aquesta Autoritat Certificadora també seria necessària per compartir la clau pública per validar els tokens i no haver de configurar-la de forma manual.

La part més important és el idProvider: un back-end que gestiona una base de dades amb la informació d'usuaris, i és l'encarregat de crear i firmar els tokens.

Després tenim un altre back-end que l'he anomenat geomètric-resources que rebrà aquests tokens en les sol·licituds HTTP a través d'un servei REST que és capaç de validar-los. Si és així, dona per autenticat l'usuari i li permet l'accés als recursos de la seva base de dades. Entre aquests dos back-end hi ha una aplicació, que és la que utilitza un usuari; podria ser una aplicació mòbil, un programa o una pagina web. En aquest cas he creat una pàgina web SPA, que l'he anomenat gometric-app.
És un exemple per demostrar que el sistema funciona i que ens podem autenticar a diferents servidors amb un JWT. Aquesta SPA permet crear figures geomètriques que es guarden en una base de dades, que és gestionada pel segon back-end, el geomètric-resources.

Un sistema així desenvolupat en microserveis té avantatges a la hora d'evolucionar-lo i fer els deploys, ja que no és un sol servidor enorme (normalment anomenat monòlit) on qualsevol canvi afecta a tot el sistema i a l'hora de fer una actualització -per petita que sigui- s'ha de fer un deploy de tot i això comportarà més temps, ja que ocuparà molt més. La base de dades serà més complexa perquè guardarà informació de diversos serveis.
Amb l'arquitectura de microserveis tenim servidors petits i més simples i de la mateixa manera cada un d'ells té una base de dades més petita i simple que si ho tinguéssim tot junt.

Altres avantatges que té aquesta arquitectura és l'escalabilitat del sistema: poder balancejar els servidors de manera diferent donant prioritat als serveis que tenen més us, i poder desplegar en contenidors utilitzant una tecnologia com pot ser Docker.

Un possible inconvenient podria ser un alt us de memòria (tot i que cada cop és menys important) o bé més consum elèctric. Una altra dificultat és una complexitat a l'hora de gestionar molts serveis, com per exemple poder fer proves de tot el sistema sencer, ja que normalment es desenvolupen els serveis per separat

En aquest cas és un sistema molt petit, és nomes un exemple; però podria ser l'inici d'un gran sistema, ja que si tenim un servidor que controla els usuaris, a partir d'aquí podríem anar donant serveis diversos a aquests usuaris. Si fos una botiga, per exemple, podríem fer un servidor que mostrés els catàlegs, un altre perquè controles les comandes, un altre que controlés els pagaments, un altre que enviés els e-mail, etc. D'aquesta manera no seria necessari tornar a desplegar tot el sistema quan canviéssim els catàlegs o quan afegíssim un nou sistema de pagament, o canviéssim el contingut d'un e-mail.

2. ESTAT DE L'ART DE LES TECNOLOGIES APLICADES EN AQUEST PROJECTE

2.1 MAVEN

Maven és una eina open-source creada el 2001 amb l'objectiu de simplificar els processos de build (compilar i generar executables a partir del codi font).

Maven és un intent d'aplicar patrons a la infraestructura de construcció d'un projecte i proporciona una interfície comú per fer els builds del software, s'encarrega d'afegir i compilar les llibreries necessàries i les dependències del projecte.

Maven és una eina capaç de gestionar un projecte de software complet, des de l'etapa en la que es comprova que el codi és correcte, fins que es desplega l'aplicació, passant per l'execució de proves i generació d'informes i documentació.

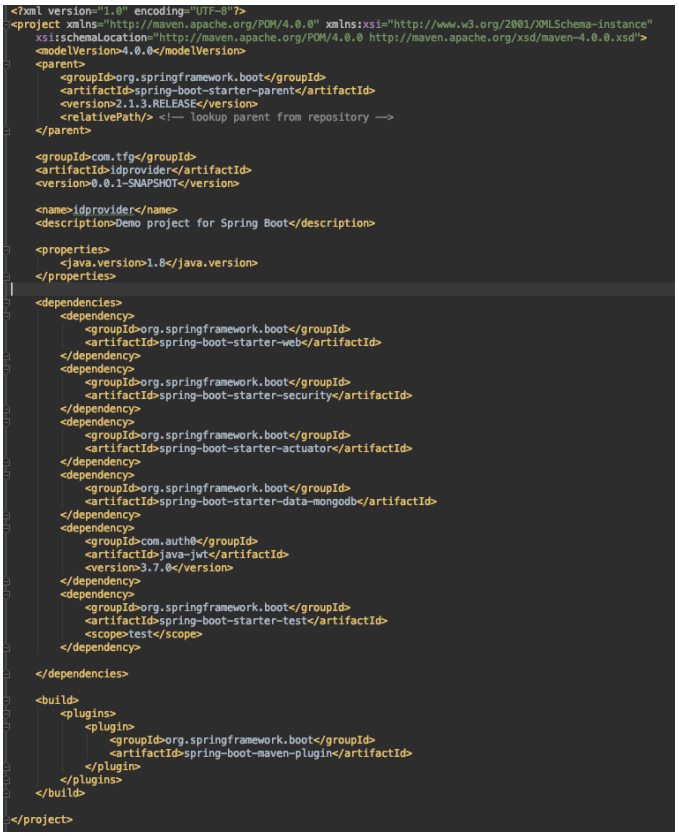
Maven defineix les següents fases pel cicle de vida del projecte:

- **validate**: Validació que el projecte és correcte, i que té tota la informació necessària.
- **compile**: Compilació del codi font del projecte.
- **test**: Prova el codi font compilat utilitzant un framework de proves unitàries com pot ser JUnit.
- **package**: empaquetar el codi compilat i transformar-lo en un format del tipo .war o .jar
- **verify**: executar qualsevol comprovació dels resultats de les proves d'integració per garantir que es compleixen els criteris de qualitat.
- **install**: instal·lar el codi empaquetat en el repositori de Maven local, per poder utilitzar-lo com una dependència d'altres projectes.
- **deploy**: còpia el paquet final al repositori remot per compartir-lo amb altres desenvolupadors i projectes.

Poder realitzar una d'aquestes fases en el nostre codi, és tan simple com executar el comando $mvn *nom_fase*. A més a més van en cadena, per exemple $mvn verify executarà des de validate fins a verify.

Per una altra banda Maven gestiona les dependències entre mòduls i les versions de llibreries. Només hem d'indicar els mòduls que composen el projecte o les llibreries que utilitza el software que estem desenvolupant en un fitxer que es diu pom.xml. A més a més, en el cas de les llibreries no fa falta descarregar-les; ja que Maven té un repositori central on estan la majoria de les llibreries necessàries i el propi Maven s'encarrega de descarregar-les i adjuntar-les al projecte.

Aquest es el pom.xml del idProvider, POM (Project Object Model)



```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.3.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <groupId>com.tfg</groupId>
    <artifactId>idprovider</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>idprovider</name>
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb</artifactId>
        </dependency>
        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>java-jwt</artifactId>
            <version>3.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Imatge 1: pom.xml

Podem veure que té la informació amb el nom del projecte, la versió del SDK de Java que es la 1.8 i que té 5 dependències:

- **spring-boot-starter-web** (per fer anar un servei rest)
- **spring-boot-starter-security** (per gestionar el tema del login i filtrar els end points que necessiten token)
- **spring-boot-starter-actuator** (per poder obtenir informació de la aplicació quan està desplegada)
- **spring-boot-starter-data-mongodb** (per poder gestionar una base de dades MongoDB)
- **java-jwt** (per poder llegir, crear i firmar els tokens)
- **spring-boot-starter-test** (per poder testejar el projecte amb test unitaris)

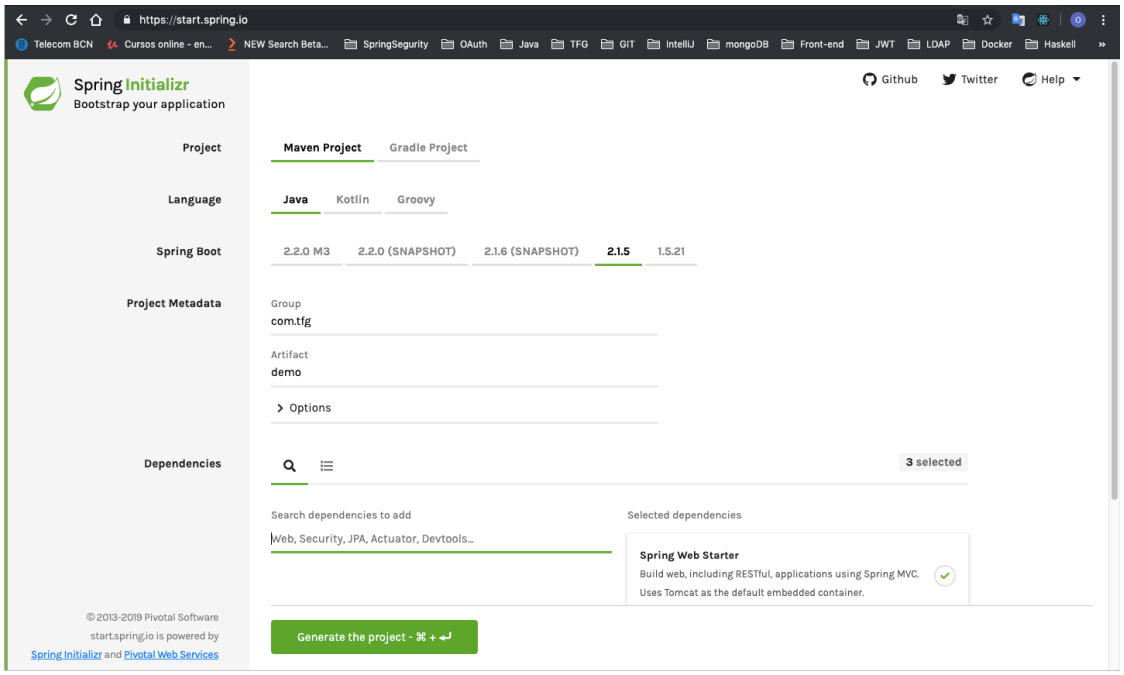
Moltes d'aquestes dependències són llibreries del framework Spring Boot; un framework pensat per desenvolupar una arquitectura basada en microserveis, al no indicar quina versió de la dependència estem utilitzant de manera automàtica descarregarà la latest (ultima versió estable)

2.2 SPRING BOOT

Spring Boot és una eina que neix amb la finalitat de simplificar el desenvolupament d'aplicacions basades en el framework Spring. Spring Boot busca que el desenvolupador es centri en el desenvolupament de la solució i no en la configuració, que és la part mes complexa de Spring, per tal

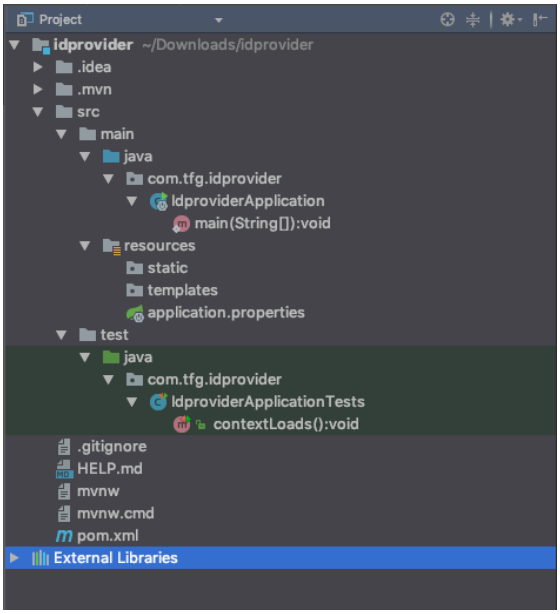
d'aconseguir això, té un mòdul d'autoconfiguració que aconsegueix simplificar molt aquesta tasca.

A més a més, per crear un projecte basat en el framework Spring Boot podem iniciar-lo des de la pàgina web https://start.spring.io/



Imatge 2: start.spring.io

Haurem d'introduir la informació bàsica del projecte, com ara el llenguatge que utilitzarem, Java en aquest cas, i la versió tant de Java (1.8) com del framework (2.1.5) a més de les dependències.
Això generarà una estructura de carpetes i l'arxiu pom.xml; és a dir, crearà un projecte (Maven) buit, però configurat per poder arrencar en el port per defecte 8080 amb un servidor Tomcat. I a partir d'aquí poder començar a desenvolupar la solució i canviar les configuracions que no volem que hi estiguin per defecte.



Imatge 3: esquelet projecte idprovider

Una aplicació de Spring Boot arrenca a la classe que té l'anotació @SpringBootApplication. Aquesta anotació és equivalent a utilitzar les tres anotacions: @EnableAutoConfiguration, @Configuration i @ComponentScan. Això fa que un cop arrenca l'aplicació, fa una autoconfiguració i busca els components per crear els Beans, fent la injecció de dependències necessària per arrencar l'aplicació.

Els Beans a Spring conformen el nucli de la injecció de dependències. Un Bean a Spring és qualsevol objecte que està emmagatzemat i gestionat pel contenidor d'Spring i pot ser injectat en qualsevol altre component (IoC). Aquests beans no s'han de confondre amb els Enterprise Java Beans (EJB).

Spring té un contenidor de Beans on crearà una instància de cada un dels Beans necessàris pel funcionament de l'aplicació.
Spring Boot recomana la configuració a través de classes de Java, tot i que encara es pot utilitzar els fitxers XML com es feia abans amb Spring.
Sempre podem crear classes de configuració addicional amb l'anotació @Configuration.

Si mantenim l'estructura de codi amb la classe anotada amb @SpringBootApplication sola en el package arrel, i el nostre codi està repartit en diferents packages amb classes anotades amb @Component, @Controller, @Service i @Repository, a l'arrencar l'aplicació Spring Boot crearà un Bean per a cada una d'aquestes classes i les guardarà en el IoC de forma automàtica.
Si no hauríem de definir el nom de les classes que volem registrar com a Bean en el parèntesis del @ComponentScan.

Si els nostres Beans tenen definit un constructor Spring Boot, ja s'encarrega de fer la injecció de dependències necessària per poder crear l'objecte i afegir-lo com un Bean amb tots els atributs necessaris. Si no tenim constructor, podríem utilitzar l'anotació @Autowired per crear els atributs necessaris.

2.3 MONGODB

MongoDB és una base de dades No-SQL orientada a documents; per tant no tenim ni taules ni files ni columnes. Tot i que si busquem una similitud amb una SQL, podem dir que tenim col·leccions (taules), documents (files) i atributs (columnes).

Aquests documents es guarden en BSON, que és una representació binaria de JSON. Cada document té un límit de 16MB.

Una de les diferències més importants amb una base de dades SQL és que els documents dins una col·lecció no han de tenir tots la mateixa forma i un document no només pot tenir atributs, sinó que pot tenir documents o llistes a dins seu.

Un cop instal·lada amb el comando **$mongod,** arrenquem el servei de MongoDB que per defecte estarà escoltant el port 27017.
Una altra de les diferències més importants és que no tenim JOINTS ni el llenguatge SQL, però es poden fer consultes des de la consola de comandos. Per accedir a la consola de MongoDB, primer hem d'iniciar el servei i després en un altre terminal **$mongo** aquest comando ens donarà accés a la consola de MongoDB.

2.4 TYPESCRIPT

Tota la part del front-end servidor i SPA està programada amb el llenguatge Typescript, que es compila a codi JavaScript, això permet tenir un tipat fort i detectar molts errors en temps de compilació i no en execució.

En TypeScript pots escriure ECMAScript (ES), que és l'estàndard de JavaScript.

En HTML5, els navegadors d'avui en dia són capaços d'executar la versió ES5 que és de finals del 2009.

Però el llenguatge de ECMAScript és realment potent a partir de la versió ES6 de l'any 2015, que és on ha obtingut les funcions anònimes (també anomenades funcions de fletxa o lambdas), classes, variables let i const, iteradors, mòduls, promises ...

En la versió ES8 es van introduir les funcions async await molt útils per controlar les promises, al fer crides REST, per exemple.

A l'escriure el codi en TypeScript podem utilitzar la versió més moderna de ECMAScript. TypeScript farà una comprovació de tipus i Babel compilarà el codi a ES5 per que qualsevol navegador el pugui interpretar.
A més TypeScript et permet tenir interfícies, herències i sobrecàrrega.

La forma d'indicar el tipus en TypeScript és la següent:

nom_atribut: tipus

```
export interface IProfile {
    username: string,
    email: string,
    roles: string[]
    personalData: IPersonalData
}
```

Imatge 4: exemple de tipat en atributs

nom_funció (paràmetres:tipus_parametres): tipus_retorn

```
export function setGeometricList(state = initialGeometricList, action: GeometricListActions): ListFiguresState {
    switch (action.type) {
```

Imatge 5: exemple de tipat en funcions

2.5 NODE

El front-end és un servidor Express que és el framework web més popular de Node.js.

Aquest servidor està desenvolupat utilitzant npm (Node Package Manager) que és un gestor de paquets per JavaScript. Gràcies a npm podem tenir qualsevol llibreria disponible en el nostre projecte amb una sola línia de codi, utilitzant el comando

- **$npm install nom_paquet**
- **$npm install @types/nom_paquet**

Aquest segon comando és per instal·lar els tipus utilitzats per TypeScript.
npm ens ajuda a administrra els nostres mòduls de node, distribuir paquets i agregar dependències d'una manera senzilla.
Podríem dir que npm fa una funció semblant a la fa que mvn (Maven) en Java.
En un projecte Node.js tenim un document que es diu package.json i té la informació del propi projecte, com pot ser el nom, la versió, les dependències, els scripts ...
Aquest document s'anirà modificant a l'utilitzar el comando npm install i s'aniran afegint aquí totes les dependències necessàries.
També tenim un altre document que es diu package-lock.json que també té informació del projecte, es modifica de forma automàtica cada cop que es modifica el package.json o s'afegeix un nou mòdul de Node utilitzant npm, descriu l'arbre exacte de com ha estat generat el projecte.
Això serveix pels següents temes:

- Descriu una representació única de l'arbre de les dependències. De manera que tan companys d'equip, implementacions o integració continua tenint exactament les mateixes dependències.
- Proporciona una facilitat per viatjar en el temps cap a estats anteriors de node_modules.
- Optimitza el procés d'instal·lació de npm, ja que pot ignorar els paquets repetits instal·lats prèviament.

Com que he creat una SPA basada en React, el servidor és molt simple, ja que no te accés a cap base de dades, simplement gestiona les vistes i fa peticions HTTP als altres servidors.

2.6 REACT

React fa que sigui fàcil crear interfícies d'usuari interactives. Dissenya vistes simples per cada estat de l'aplicació en components. Un component és una part petita independent i reutilitzable que representa una part de la UI
React utilitza fitxers .jsx (en JavaScript) o .tsx (en TypeScript) pels components, això fa que el codi de les vistes sigui molt més declaratiu.
Aquests fitxers accepten expressions JSX, però després de la compilació es converteixen a funcions regulars de JavaScript i s'avaluen els objectes de JavaScript.

Per defecte, React DOM escapa dels valors incrustats a JSX abans de representar-los. Per tant, garanteix que mai no s'injectarà res que no estigui escrit explícitament a l'aplicació. Tot es converteix en una cadena abans de ser processat. Això ajuda a evitar atacs XSS.

JSX representa objectes, i Babel compila JSX fins a React.createElement.
Aquests dos exemples son iguals:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

Imatge 6: exemple d'expresio JSX

Un cop compilat per Babel tenim això

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

Imatge 7: exemple utilitzant createElement()

Finalment el navegador llegirà un objecte JavaScript semblant a aquest.

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

Imatge 8: exemple compilat amb Babel

Si el nom de l'expressió comença amb majúscula, això indica que és un component de React. Si no, és un element HTML.

2.7 REDUX

Redux és un contenidor d'estat predictible per a aplicacions de JavaScript.

Ajuda a escriure aplicacions que es comporten de forma coherent i són fàcils de provar.
Redux es pot utilitzar amb React o amb qualsevol altra biblioteca de vistes. És petit (2Kb, incloses les dependències), però té un gran ecosistema de complements disponibles.

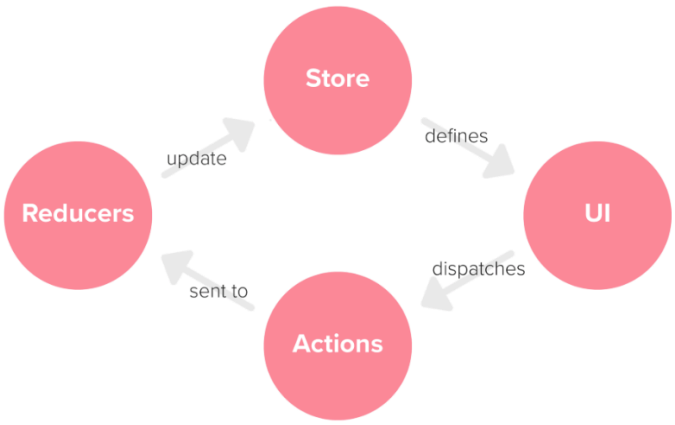
Redux està disponible com un paquet a npm per al seu ús amb un agrupador de mòduls o en una aplicació Node:

$npm install --save redux

D'aquesta manera tenim que la part visual està formada per components funcionals de React, i tot el funcionament de l'aplicació està controlat per Redux. així aconseguim separar la part visual de la funcionalitat. Tot el codi té una orientació funcional; és a dir, són petites funcions on el resultat depèn sempre de l'entrada. Només les funcions que realitzen la connexió amb un dels altres servidors no segueixen aquest principi. Tampoc hi ha cap bucle, ja que sempre que opero amb llistes utilitzo les funcions map(), filter() etc

La part de Redux també està escrita amb TypeScript i està dividida en tres parts: accions, reducers i store.

Redux sempre segueix el mateix flux unidireccional.



Imatge 9: flux unidirecional de Redux

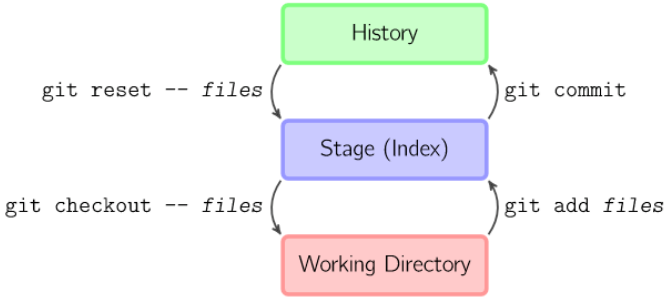
Els components de React a través de la funció dispatch de Redux poden llençar accions.
Aquestes accions les recullen els reducers, que son els encarregats d'actualitzar l'estat de l'store.
L'store es qui manté l'estat de l'aplicació. Es diu que és la única font de la veritat. Cada cop que l'estat és actualitzat, aquest es torna a passar cap als components i això provoca que es tornin a "repintar" tots els components que tenen algun canvi en els seus valors.

2.8 GITHUB

GitHub és un servei de hosting de repositoris Git, que ofereix les funcionalitats de Git, com poden ser un control de versions i poder compartir i revisar el codi en equip. A diferència de Git, el qual és estrictament una eina de línia d'ordres, GitHub proporciona una interfície gràfica basada en web i escriptori.

Jo l'he utilitzat bàsicament per tenir una copia de seguretat del codi i poder-lo compartir amb el tutor del projecte. També per poder fer alguna branca de prova amb funcionalitats que al final no he afegit al projecte.

Un repositori local de git està compost per tres arbres: el directori local on estan els arxius que modifiques, l'Index que actua com a zona intermèdia i el Head que apunta al últim commit realitzat.



Imatge 10: Funcionament dels directoris locals de git

Els comandos bàsics per poder gestionar un repositori git són:

$git init per iniciar un repositori de git en una carpeta local
$git clone username@host:/path/to/repository per clonar un repositori remot
$git add <filename> per registrar un fitxer amb canvis a l'Index
$git add . per registrar tots els fitxers amb canvis a l'Index
$ git commit -m "Commit message" per passar tots els fitxers de l'Index al Head
$git push origin master per enviar els canvis del Head al repositori remot
$git checkout –b feature_x per crear una nova branca amb el nom feature_x
$git push origin feature_x per pujar la nova branca al repositori remot
$git checkout master per canviar a la branca master

Tot el codi es pot veure en els tres repositoris següents:

https://github.com/lru89/idprovider

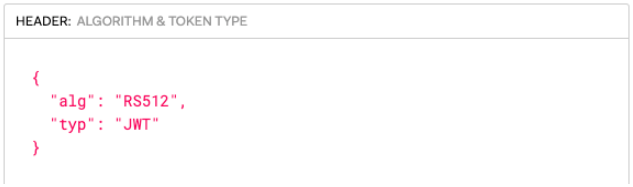
https://github.com/lru89/geometric-resources

https://github.com/lru89/geometric-app

3. METODOLOGIA / DESENVOLUPAMENT DEL PROJECTE

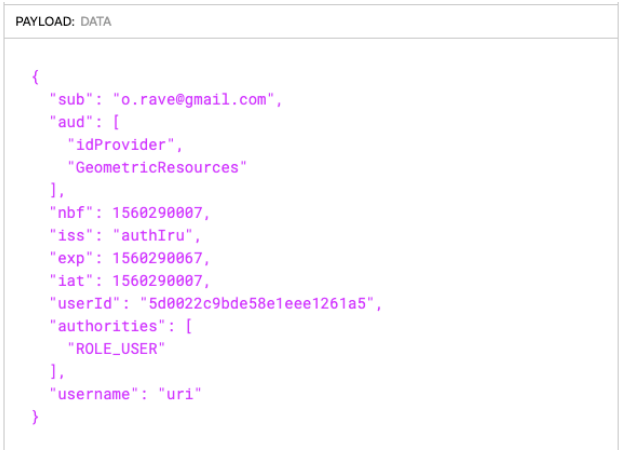
3.1 FORMAT DEL JWS

Un JSON Web Token defineix el format del token. Com el nom indica, té forma de JSON, i està dividit en tres parts: header, payload i signature.
En la meva implementació, el header té la informació mínima , que és utilitzada per indicar quin tipus de token és i quin tipus d'algoritme s'ha utilitzat per la firma.



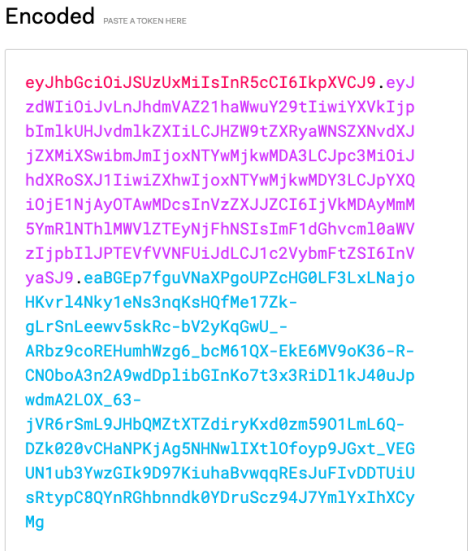
Imatge 11: header JWT

El payload té la informació de qui ha creat el token, el temps que és vàlid, els servers on pot ser utilitzat i la informació de l'usuari.



Imatge 12: payload JWT

Aquestes dues parts estaran codificades amb base64 i separades per un punt, i tota aquesta informació estarà encriptada amb la clau privada d'algoritme RSA512 i codificat en base64 en la última part, que es la signature. Per tant, si algú modifica el token, no serà validat després, ja que la firma no coincidirà amb el header i el payload.



Imatge 13: JWT codificat Base64

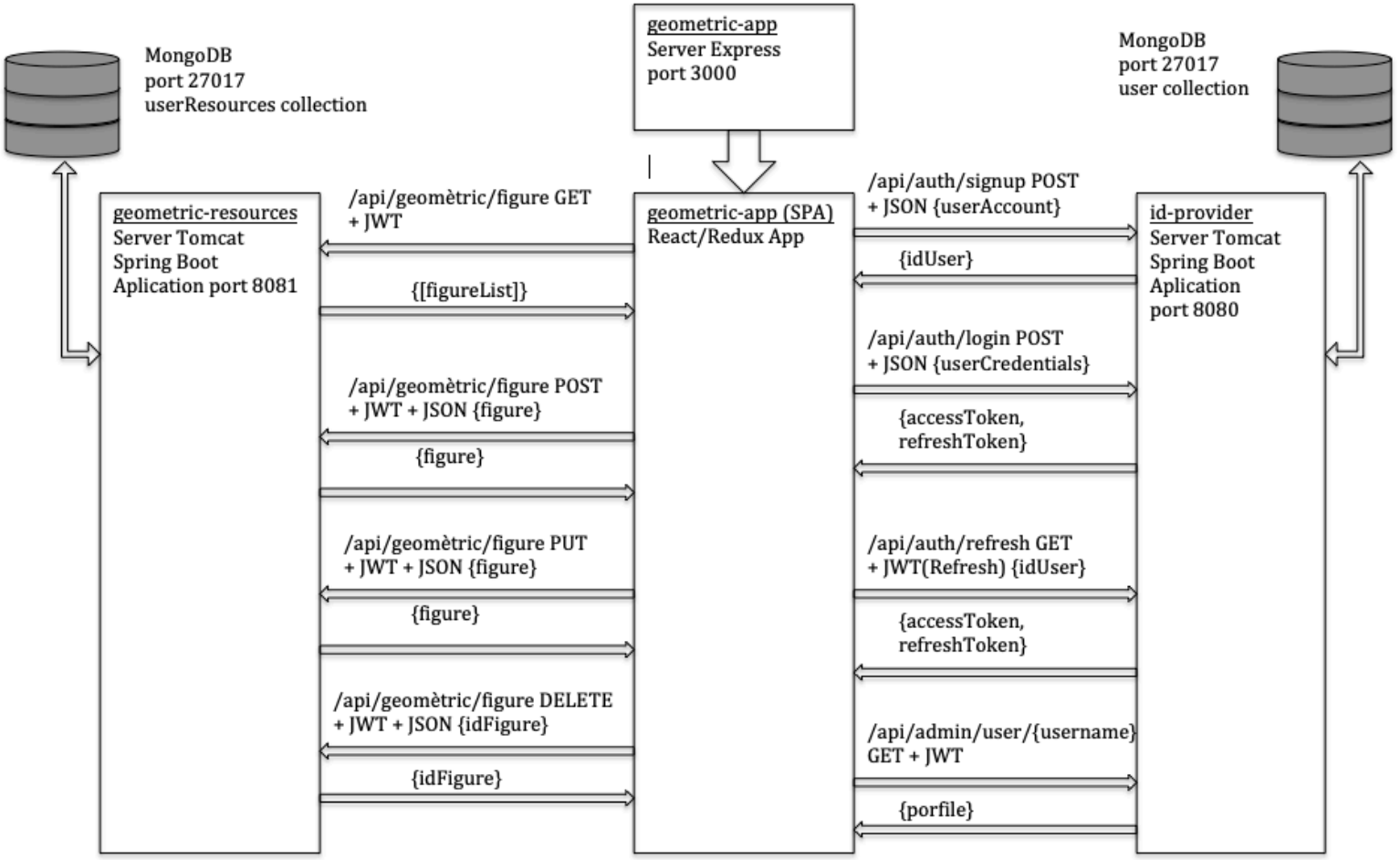
Tenint en compte això, hem de ser conscients que en un JWT no hi hauria d'haver informació privada de l'usuari, com ara contrasenyes, ja que tant el header com el payload només estan codificats amb base64. Si algú fos capaç d'interceptar el token podria accedir a la informació. També és important dir que no necessitem posar aquesta informació dins el token. I que si fos imprescindible afegir informació d'aquest tipus, hauríem d'utilitzar un altre tipus de token com JWE.

Quan verifiquem un JWT, a part de comprovar que la signature coincideix amb el payload, també es comproven els temps:

- iat moment en el que el token ha estat firmat (iat < temps actual)
- exp moment en el qual caduca el token (exp > temps actual)
- nbf moment a partir del qual el token pot ser utilitzat (nbf < temps actual)

També es fan comprovacions del valor de audience (aud) que conté els noms o url dels dominis on pot ser utilitzat el token.
En el cas del token de refresh, comprovo també que el valor jti (JWT ID) coincideixi amb el idRefresh que tinc guardat a la base de dades. Cada cop que un usuari fa un login o utilitza un token de refresh, canvio aquest id i li entrego el token de refresh amb el nou id. D'aquesta manera, tot i que el token de refresh té un temps de caducitat més alt, només és vàlid l'últim que ha estat firmat.

3.2 SISTEMA



Imatge 14: esquema del sistema complet

3.3 IDPROVIDER

En el sistema descrit comença un servidor que és l'encarregat de generar tokens, el idProvider. Aquest servidor és un projecte Maven fet amb el framework SpringBoot i programat amb Java 8.

Aquest servidor gestiona una base de dades MongoDB amb la informació dels usuaris. És una API RESTful on es poden crear nous usuaris, fer un login per obtenir un token, obtenir la informació del usuari o refrescar els tokens sense necessitat d'utilitzar les credencials.
Tant per crear un usuari nou com per fer un login no cal estar autenticats i qualsevol aplicació, ja sigui una pagina web o una aplicació mòbil, podrien utilitzar-los.

3.3.1 Base de dades

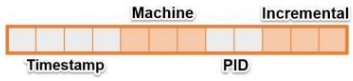
La base de dades és una MongoDB que té una col·lecció (taula) de documents (files) com aquest per cada usuari:


```
{
    "_id" : ObjectId("5d0022c9bde58e1eee1261a5"),
    "jwtRefreshId" : ObjectId("5d0022d7bde58e1eee1261a7"),
    "username" : "uri",
    "password" : "$2a$10$/ZBnETSg9Hfqq3zDuLPP8.SfiOOpRbFe2sM6jr6p8hYZN0JEfuqx.",
    "email" : "o.rave@gmail.com",
    "roles" : [
        "ROLE_USER"
    ],
    "personalData" : {
        "firstName" : "oriol",
        "lastName" : "raventos"
    },
    "_class" : "com.tfg.idprovider.model.User"
}
```


Imatge 15: exemple MongoDB del idProvider

Com podem veure, té forma de JSON i conté la informació de perfil agregada al crear l'usuari més els roles i dos ids. El primer és el quin representa a l'usuari i el segon es el quin indica quin id ha de tenir el tokenRefresh per ser vàlid. Amb aquest segon id, que fa referencia al token de refresh, aconseguim que només l'últim token de refresh generat sigui vàlid; d'aquesta manera evitem que si un token de refresh es robat per algú que no es el propietari, en el moment que l'usuari faci un login queda anul·lat l'anterior.

El camp id de l'exemple anterior és del tipus ObjectId. Aquest objecte està especialment pensat per garantir unicitat en entorns distribuïts com MongoDB. El camp està compost per 12 bytes. Els quatre primers bytes són un timestamp amb els segons des de l'epoch d'Unix, els tres següents bytes representen l'identificador únic de la màquina, els dos següents l'identificador del procés i - per finalitzar- els últims tres bytes, són un camp incremental.



Imatge 16: bytes per un ObjectId

En definitiva, els nou primers bytes ens garanteixen un identificador únic per segon, màquina i procés. Els tres últims bytes ens garanteixen que cada segon

podem inserir 2 ^ 24 = 16.777.216 documents amb un identificador diferent. Encara que tècnicament un _id podria repetir-se, en la pràctica és un nombre tan alt que és molt difícil que això passi.

3.3.2 Contrasenyes dels usuaris

No podem guardar la contrasenya en text pla; ja que si la base de dades quedés compromesa, això seria un desastre. A més a més, per temes de protecció de dades, ni els propis gestors del sistema no han de poder veure la contrasenya.
Per això utilitzo un sistema que es diu BCrypt perquè és el mecanisme més modern que he trobat i està acceptat per Spring Security. Altres mecanismes com MD5PasswordEncoder o ShaPasswordEncoder ara mateix es consideren dèbils i no és aconsellable utilitzar-los.

BCrypt genera i guarda a la base de dades un hash del text pla del password introduït per l'usuari. Primer de tot genera un salt (un string aleatori) per codificar cada contrasenya de manera diferent, d'aquesta manera dues contrasenyes iguals es guarden de manera diferent.

En totes les contrasenyes podem veure tres prefixos $:

- $2a que indica la versió de BCrypt. Aquesta en concret és de l'any 2014 i el password ha de estar codificada en UTF-8 i
- $10 indica que utilitza 2^{10} rondes d'expansió de clau, això vol dir que ha fet 1024 rondes d'afegir el salt al password i calcular el hash, tornar a afegir els salt al resultat i tornar a calcular el hash resultant, aproximadament es tarda 0,1 segons a fer aquest càlcul.
- Els 22 caràcters que segueixen al tercer $ és el salt aleatori utilitzat per crear el hash. I els següents 31 caràcters són el hash final.

Amb aquesta tècnica, el password real de l'usuari no queda guardat, sinó que només guardem hash resultant d'aplicar a l'operació BCrypt, com una cadena de 60 caràcters. Aquests caràcters estan codificats en Radix-64 que utilitza l'alfabet unix/ crypt i que no és l'estandard Base-64.

3.3.3 Magatzem de claus

Aquest servidor té un magatzem de claus que és generat per un petit programa java anomenat keytool. És una eina de gestió de claus i certificats, que en permet generar un magatzem i parells de clau rsa i guardar-los de forma segura i encriptada en un document .pkcs12

```
$ keytool -genkeypair -alias idprovider -storetype pkcs12 -keyalg RSA -keysize 2048 -keystore keystore.p12 -validity 3650 -dname "CN=localhost, OU=idProvider, O=UPC, L=Barcelona, S=Catalunya, C=CA" -storepass secret
```

Amb aquest comando generem un magatzem que té el nom de keystore.pkcs12. El crearem dins la carpeta resources/keystore del projecte; d'aquesta manera podrem accedir a ell des del codi i llegir el parell de claus rsa

de 2048 bytes, amb una validesa de 3650 dies (10 anys), aquest magatzem té una contrasenya per accedir a ell, que es *secret.*

En el document application.properties definirem els valors necessaris per accedir al magatzem i per configurar la base de dades.

```
##port por defecte de idprovider
server.port=8080
##informacion para conectar la mongodb
spring.data.mongodb.host=localhost
spring.data.mongodb.port=27017
spring.data.mongodb.database=id_provider_db
spring.data.mongodb.authentication-database=admin
spring.data.mongodb.username=root_provider
spring.data.mongodb.password=secret_provider
##informacion del keystore
idp.keystore.source = keystore/keystore.pkcs12
idp.keystore.password = secret
idp.keystore.signing-key-alias = idprovider
```

Imatge 17: application.properies del idProvider

3.3.4 Creació del JWT

Per crear i firmar un token utilitzo dues classes:

- JwtConfiguration: Aquesta classe -com el nom indica- és una configuració, i és l'encarregada de llegir el magatzem de claus i crear un Bean que és un objecte KeyPair que conté les claus. La privada per poder crear la firma i la pública per validar-la.

```
package com.tfg.idprovider.config;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.ClassPathResource;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.Certificate;

@Configuration
public class JwtConfiguration {

    @Value("${idp.keystore.source}")
    private String keyStoreSource;

    @Value("${idp.keystore.password}")
    private String keyStorePassword;

    @Value("${idp.keystore.signing-key-alias}")
    private String signingKeyAlias;

    @Bean
    public KeyPair keyPair() throws Exception {
        final KeyStore keyStore = KeyStore.getInstance("PKCS12");
        keyStore.load(new ClassPathResource(keyStoreSource).inputStream, keyStorePassword.toCharArray());
        final Key key = keyStore.getKey(signingKeyAlias, keyStorePassword.toCharArray());
        final Certificate certificate = keyStore.getCertificate(signingKeyAlias);
        return new KeyPair(certificate.publicKey, key);
    }

}
```

Imatge 18: JwtConfiguration

Com podem veure, aquesta classe està anotada com a @Configuration, té tres atributs que els llegeix del fitxer application.properties fent us de l'anotació @Value i posant el nom del paràmetre que volem llegir dins "${nom_parametre}"

Aquest objecte està anotat amb @Bean; d'aquesta manera indiquem a Spring que ha guardar aquest objecte en el seu Contenidor, per poder-lo utilitzar després en IoC.

Per crear aquest objecte utilitzo els tres paràmetres per llegir el keystore que conté la clau privada i la clau pública (certificat).

- JwtProvider: Aquesta classe fa us del Bean KeyPair i és l'encarregada de crear, firmar i validar els tokens.

Veurem aquesta classe per parts.

```java
package com.tfg.idprovider.security;

import ...


@Component
public class JwtProvider {

    private static final String ISSUER = "authIru";
    private static final String ID_PROVIDER = "idProvider";
    private static final String GEOMETRIC_RESOURCES = "GeometricResources";

    private KeyPair keyPair;

    public JwtProvider(KeyPair keyPair) { this.keyPair = keyPair; }

    public JwtAuthenticationDto generateTokens(MyUserDetails myUserDetails) {

        String accessToken = generateAccessToken(myUserDetails);
        String refreshToken = generateRefreshToken(myUserDetails);

        JwtAuthenticationDto jwt = JwtAuthenticationDto.JwtAuthenticationDtoBuilder
                .builder()
                .withAccessToken(accessToken)
                .withRefreshToken(refreshToken)
                .build();

        return jwt;
    }
```

Imatge 19: JwtProvider part 1

Aquesta classe està anotada com a @Component perquè és un component de l'aplicació que també haurà d'estar en el contenidor d'Spring. Veiem que el constructor té el paràmetre KeyPair, que és la Bean creada en la classe anterior. El primer mètode rep les dades d'un usuari i crearà els dos tokens (access i refresh) per encapsular-los dins un DTO per retornar-los com un JSON.

```
    private String generateAccessToken(MyUserDetails myUserDetails) throws JWTCreationException {

        Algorithm algorithm = getAlgorithm(keyPair);
        Map<String, Object> headers = headerClaims;

        return JWT.create()
                .withHeader(headers)
                .withIssuer(ISSUER)
                .withSubject(myUserDetails.getEmail())
                .withAudience(ID_PROVIDER, GEOMETRIC_RESOURCES)
                .withExpiresAt(dateExpiresAccessToken())
                .withNotBefore(dateNotBefore())
                .withIssuedAt(dateIssuedAt())
                .withClaim( name: "username", myUserDetails.getUsername())
                .withClaim( name: "userId", myUserDetails.getId().toString())
                .withArrayClaim( name: "authorities", getRoles(myUserDetails))
                .sign(algorithm);

    }

    private String generateRefreshToken(MyUserDetails myUserDetails){

        val algorithm = getAlgorithm(keyPair);
        val headers = headerClaims;

        return JWT.create()
                .withHeader(headers)
                .withIssuer(ISSUER)
                .withJWTId(myUserDetails.jwtRefreshId.toHexString())
                .withAudience(ID_PROVIDER)
                .withExpiresAt(dateExpiresRefreshToken())
                .withNotBefore(dateExpiresAccessToken())
                .withIssuedAt(dateIssuedAt())
                .withClaim( name: "userId", myUserDetails.id.toString())
                .withArrayClaim( name: "authorities", getRoles(myUserDetails))
                .sign(algorithm);

    }
    private Date dateExpiresAccessToken() { return Date.from(Instant.now().plus( amountToAdd: 1, ChronoUnit.MINUTES)); }

    private Date dateExpiresRefreshToken() { return Date.from(Instant.now().plus( amountToAdd: 1, ChronoUnit.HOURS)); }

    private Date dateNotBefore() { return Date.from(Instant.now()); }

    private Date dateIssuedAt() { return Date.from(Instant.now()); }

    private Algorithm getAlgorithm(KeyPair keyPair) {
        val publicKey = keyPair.public;
        val privateKey = keyPair.private;
        return Algorithm.RSA512(publicKey, privateKey);
    }

    private Map<String, Object> getHeaderClaims() {
        val headerClaims = new HashMap();
        headerClaims["alg"] = "RS512";
        headerClaims["typ"] = "JWT";
        return headerClaims;
    }

    private String[] getRoles(MyUserDetails myUserDetails) {
        return myUserDetails.authorities.map(a->a.authority).toArray(String[]::new);
    }
```

Imatge 20: JwtProvider part 2

Com es pot veure, per crear tant l'accessToken com el refreshToken utilitzem uns petits mètodes definits més avall. Aquests dos tokens són quasi iguals; els canvis més importants són:

- El temps utilitzat per definir l'expedició sempre serà més gran en el refreshToken.

- El refreshToken té definit el temps nbf com el temps en que caduca l'accessToken, d'aquesta manera no pot ser utilitzat abans que caduqui el accessToken.

- I les audiences el accessToken pot ser utilitzat en el idProvider i en el geomètric-resources

També te uns mètodes que són utilitzats per fer la validació.

```
public ObjectId getUserIdFromJWT(String token) {

    try{
        DecodedJWT decodedJWT = JWT.decode(token);

        Claim claimUserId = decodedJWT.getClaim( s: "userId");
        if(!claimUserId.null){
            return new ObjectId(claimUserId.asString());
        }
        return null;
    }catch (JWTDecodeException e) {
        e.printStackTrace();
        return null;
    }
}

public boolean validateToken(String authToken) {
    try{
        Algorithm algorithm = getAlgorithm(keyPair);
        DecodedJWT decodedJWT = JWT.require(algorithm)
                .withIssuer(ISSUER)
                .withAudience(ID_PROVIDER)
                .acceptLeeway(5)            //Aceptem 5 seg de marge en exp nbf i iat
                .build()
                .verify(authToken);
        return true;
    }catch (JWTVerificationException e ){
        return false;
    }
}

public boolean validateTokenRefresh(String tokenRefresh, ObjectId jwtRefreshId) {
    try{
        Algorithm algorithm = getAlgorithm(keyPair);
        DecodedJWT decodedJWT = JWT.require(algorithm)
                .withIssuer(ISSUER)
                .withJWTId(jwtRefreshId.toHexString())
                .withAudience(ID_PROVIDER)
                .acceptLeeway(5)            //Aceptem 5 seg de marge en exp nbf i iat
                .build()
                .verify(tokenRefresh);
        return true;
    }catch (JWTVerificationException e ){
        return false;
    }
}
```

Imatge 21: JwtProvider part 3

El primer per extreure el id de l'usuari, això es pot fer sense validar el token i els altres dos per validar tant el accessToken com el refreshToken.

La diferència més important de com es valida un i l'altre, és que el refreshToken comprova un id que relaciona e'l usuari amb l'únic token que pot utilitzar. Això ho faig per donar un punt més de seguretat, ja que el tokenRefresh té un temps de caducitat més alt i si algú aliè a l'usuari es fes amb la possessió d'un, pràcticament podria anar generant un parell de tokens de forma indefinida. En canvi, si cada vegada que l'usuari fa un login aportant les credencials genero un id pel tokenRefresh. Si algú es fes amb un tokenRefresh tindria accés durant una estona, però en el moment que l'usuari fes un login, aquest tokenRefresh o qualsevol altre tokenRefresh generat més tard quedaria anul·lat.

3.3.5 Seguretat web

Gràcies a la dependència Spring Security ens es bastant fàcil fer un control de l'autenticació per JWT. És necessari canviar la configuració bàsica, ja que per defecte farà un control de sessió i no és el que volem, però no és molt difícil canviar això i utilitzar les pròpies classes que et proporciona per fer el control a través del JWT.

Primer de tot és necessari escriure una classe de configuració que és la següent:

```
package com.tfg.idprovider.config;

import ...

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(
        securedEnabled = true,
        jsr250Enabled = true,
        prePostEnabled = true
)
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {

    private MongoUserDetailsService mongoUserDetailsService;
    private JwtAuthenticationEntryPoint unauthorizedHandler;
    private JwtProvider jwtProvider;

    @Autowired
    public WebSecurityConfiguration(MongoUserDetailsService mongoUserDetailsService,
                                    JwtAuthenticationEntryPoint unauthorizedHandler,
                                    JwtProvider jwtProvider) {

        this.mongoUserDetailsService = mongoUserDetailsService;
        this.unauthorizedHandler = unauthorizedHandler;
        this.jwtProvider = jwtProvider;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .cors()
                .and()
                .csrf().disable()
                .exceptionHandling().authenticationEntryPoint(unauthorizedHandler)
                .and()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .authorizeRequests()
                .requestMatchers(PathRequest.toStaticResources().atCommonLocations()).permitAll()
                .requestMatchers(EndpointRequest.to(InfoEndpoint.class, HealthEndpoint.class)).permitAll()
                .antMatchers( ...antPatterns: "/api/auth/signup").permitAll()
                .antMatchers( ...antPatterns: "/api/auth/login").permitAll()
                .anyRequest().authenticated();

        // Añadimos nuestro filtro de seguridad personalizado JWT
        http.addFilterBefore(jwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class);
    }
```

Imatge 22: WebSecurityConfiguration de idProvider

Podem veure que amplia d'una classe que és de spring boot i observar les anotacions @Configuration, per indicar que es una classe de configuració. @EnableWebSecurity per indicar que canviarem la configuració automàtica de la seguretat web. @EnableGlobalMethosSecurity per activar uns quants mètodes per comprovar aspectes del token, com poden ser els Roles de l'usuari, l'id d'usuari o altres informacions que porta el JWT.
També observem que té un constructor amb la notació @Autowired on s'aplica la injecció de dependències, ja que tots els atributs són Beans que explicaré mes endavant.

El mètode que veiem aquí és l'encarregat de configurar com es comportarà la nostra aplicació amb els diferents tipus de connexió que pugui rebre.
Primer de tot habilitem el CORS i inhabilitem el CRSF (per motius ja explicats anteriorment).
Habilitem un handler per contestar amb una resposta predeterminada i un HTTP estatus 401 quan algú vulgui accedir a un punt al qual no esta autoritzat.
Permetem l'accés al end points /actuator/health i actuator/info que són útils per obtenir informació i l'estat de l'aplicació quan aquesta està funcionant, i també

als /api/auth/signup i api/auth/login. Tots els altres endpoints necessiten estar autenticats. Per autenticar els usuaris en les peticions HTTP activem un filtre que explicaré mes endavant.

La resta de la classe son els Beans necessaris per aquesta configuració

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

@Bean
public JwtAuthenticationFilter jwtAuthenticationFilter() {
    return new JwtAuthenticationFilter(jwtProvider, mongoUserDetailsService);
}


@Override
public void configure(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception {
    authenticationManagerBuilder
            .userDetailsService(mongoUserDetailsService)
            .passwordEncoder(passwordEncoder());
}

@Bean(BeanIds.AUTHENTICATION_MANAGER)
@Override
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}
}
```

Imatge 23: WebSecurityConfiguration de idProvider 2n part

La primera Bean és l'encarregada de codificar les contrasenyes utilitzant el mètode BCrypt explicat abans. La segona és el filtre que necessitarà el Proveidor de Tokens i el servei que dona accés a la base de dades que gestiona la informació dels usuaris.
Després tornem a "sobreescriure" el mètode configure, realment estem afegint informació de configuració ja que no anul·lem l'anterior. Aquesta vegada configurem el authenticationManager que és l'encarregat d'autenticar els usuaris, per això li passem una instància del servei que dona accés a la base de dades i el Bean del passwordEncoder. Per últim és necessari crear una Bean de 'l authenticationManager que acabem de configurar.

Ara anem a veure la classe AuthenticationFilter:

```java
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private JwtProvider jwtProvider;
    private MongoUserDetailsService mongoUserDetailsService;

    private static final Logger logger = LoggerFactory.getLogger(JwtAuthenticationFilter.class);

    public JwtAuthenticationFilter(JwtProvider jwtProvider, MongoUserDetailsService mongoUserDetailsService) {
        this.jwtProvider = jwtProvider;
        this.mongoUserDetailsService = mongoUserDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        try {
            String jwt = getJwtFromRequest(request);

            if (StringUtils.hasText(jwt)) {
                ObjectId userId = jwtProvider.getUserIdFromJWT(jwt);

                UserDetails userDetails = mongoUserDetailsService.loadUserById(userId);
                UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(userDetails,  credentials: null, userDetails.authorities);
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch Exception ex {
            logger.error("Could not set user authentication in security context", ex);
        }

        filterChain.doFilter(request, response);
    }

    private String getJwtFromRequest(HttpServletRequest request) {
        String bearerToken = request.getHeader( s: "Authorization");

        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
            String jwt = bearerToken.substring(7, bearerToken.length());
            if(jwtProvider.validateToken(jwt)){
                return jwt;
            }
        }else if(StringUtils.hasText(bearerToken) && bearerToken.startsWith("BearerRefresh ")){
            String jwtRefresh = bearerToken.substring(14, bearerToken.length());
            if(jwtProvider.validateToken(jwtRefresh)){
                ObjectId userId = jwtProvider.getUserIdFromJWT(jwtRefresh);
                MyUserDetails userDetails = (MyUserDetails) mongoUserDetailsService.loadUserById(userId);
                if(jwtProvider.validateTokenRefresh(jwtRefresh, userDetails.getJwtRefreshId())){
                    return jwtRefresh;
                }
            }
        }
        return null;
    }
}
```

Imatge 24: JwtAuthenticationFilter de idProvider

Primer de tot podem observar que no té cap anotació, però amplia una classe d'Spring Boot que és OncePerRequestFilter i sobreescrivim el mètode doFilterInternal. Gràcies a això ja tenim que aquest mètode s'executarà una vegada a cada request que rebi l'aplicació, seguint la configuració descrita anteriorment i depenent de l'endpoint al quin estigui connectant, el deixarà accedir o no després d'executar-se i comprovar si està autenticat.

Primer de tot recupera el JWT del request i comprova si és un accessToken o un refreshToken per validar-lo.
Després el que fa es extreure l'id d'usuari del JWT i buscar aquest usuari a la base de dades a traves del id que acaba d'obtenir del token. Un cop té aquest usuari autenticat, el carrega al context de seguretat de Spring i d'aquesta manera podem obtenir l'usuari en qualsevol punt de l'aplicació.

Després del filtre executarem la classe que fa de controlador del servei REST, aquesta classe és molt simple gràcies a Spring Boot.

```java
package com.tfg.idprovider.controller;

import ...

@RestController
@RequestMapping("/api/auth")
public class AuthController {

    private LogInService logInService;
    private SignUpService signUpService;

    public AuthController(LogInService logInService, SignUpService signUpService) {
        this.logInService = logInService;
        this.signUpService = signUpService;
    }

    @RequestMapping(value = "/login", method = RequestMethod.POST)
    public ResponseEntity login(@Valid @RequestBody UserLogInDto user){
        return logInService.login(user);
    }

    @RequestMapping(value = "/signup", method = RequestMethod.POST)
    public ResponseEntity createUser(@Valid @RequestBody UserSignUpDto user){
        return signUpService.registerNewUserAccount(user);
    }

    @RequestMapping(value = "/refresh", method = RequestMethod.GET)
    public ResponseEntity refreshTokens(){
        Authentication authentication = SecurityContextHolder.context.authentication;
        MyUserDetails myUserDetails = (MyUserDetails) authentication.getPrincipal();
        ObjectId userId = myUserDetails.getId();
        return logInService.refreshTokens(userId);
    }
}
```

Imatge 25: AuthController de idProvider

En aquesta classe les anotacions fan quasi tota la feina.
La primera @RestController és similar a l'explicada anteriorment @Controller però afegint-hi @ResponseBody que retornarà en el payload de la resposta l'objecte serialitzat de manera automàtica, utilitzant la llibreria Jackson que és la predeterminada d'Spring.
L'anotació @RequestMapping indica la url a la que respon cada mètode. Al posar aquesta anotació sobre la classe fa que tots els mètodes responguin a la url ip:port/api/auth, però cada mètode torna a tenir aquesta anotació indicant el final de la url i el mètode HTTP al que respon. El que fa es composar la url, per exemple, el primer mètode respon a la url ip:port/api/auth/login.

3.3.6 API RESTful

URL completa	Mètode REST	Mètode Java
ip:port/api/auth/signup	POST	login(@Valid @RequestBody UserLogInDto user)
ip:port/api/auth/login	POST	createUser(@Valid @RequestBody UserSignUpDto user)
ip:port/api/auth/refresh	GET	refreshTokens()
ip:port/api/admin/user/{username}	GET	getProfile(@PathVariable("username") String username)

L'anotació @Valid seguida de @RequestBody és per indicar que juntament amb la petició ha de venir un JSON en el payload del Request amb un objecte que tingui la forma del tipus indicat.

L'últim servei Rest està definit en una altra classe, ja que no té a veure amb el tema de l'autenticació.
L'anotació @PathVariable és per indicar que la variable username, en aquest cas, vindrà dins la url indicat amb les claus {}, a més a més estem comprovant que sigui el mateix username que està escrit en el JWT.

El controlador sempre utilitza un @Service que a la vegada fa us d'un @Repository per consultar guarda o actualitzar una base de dades. Tots els serveis REST de l'aplicació segueixen el mateix flux de dades utilitzant diferents controladors i serveis. De repository només en tinc un, ja que només utilitzo una base de dades amb una sola col·lecció.

Funcionament:

- Primer de tot és necessari crear un usuari des de l'endpoint /api/auth/singup amb un mètode POST enviant un JSON amb la informació necessària per crear un nou usuari: username, password, email, firstName, lastName.
- Un cop tenim un usuari, podem fer un login a l'endpoint /api/auth/login també utilitzant un mètode POST i enviant un JSON amb el usernameOrEmail i el password. Podem fer el login utilitzant tant el nom d'usuari com l'email.
- El servidor verifica les credencials, i si són correctes en respondrà amb dos tokens firmats, l'accessToken i el refreshToken.
- El token és guardat pel cantó del client, normalment en el localstorage(si parlem d'una SPA)
- A partir d'aquí el client inclourà l'accessToken en cada una de les peticions REST que realitzi als diversos servidors on es pugui autentificar amb aquest token. De moment només pot ser utilitzat en el idProvider i geometricResources.
- Amb l'accessToken es pot accedir a un altre endpoint admin/user/{username} amb un mètode GET, per obtenir la informació de perfil d'un usuari. {username} serà el nom de l 'suari que es vol obtenir.
- Amb el refreshToken podem utilitzar l'endpoint /auth/refresh amb un mètode POST i enviant un JSON amb el id d'usuari per obtenir dos tokens nous sense necessitat de tornar a utilitzar el login, de manera que l'usuari no en serà conscient. El refreshToken té un temps de vida més alt que l'accessToken, però un cop utilitzat ja no servirà més.

3.3.7 JSON

Per serialitzar o deserialitzar els JSON des del back-end he utilitzat la llibreria Jackson. Per deserialitzar en un objecte que sigui immutable, he utilitzat un patró de disseny en que encapsulo l'objecte POJO en un objecte que anomeno DTO. Aquest DTO està compost per dues classes.

La primera conté el POJO, amb el constructor i els atributs privats. També té els mètodes gets per accedir als atributs, aquesta classe porta l'anotació @JsonDeserialize(builder =*nom_classe.nom_classe_builder*.class).

La segona classe és una classe interna i és l'encarregada de construir l'objecte. Esàa feta seguint el patró de disseny Builder.

D'aquesta manera aconsegueixo deserialitzar els JSON d'una manera neta i amb objectes que són immutables.

```java
package com.tfg.idprovider.model.dto;

import ...

@JsonDeserialize(builder = UserLogInDto.UserLogInDtoBuilder.class)
public class UserLogInDto {

    private final String usernameOrEmail;
    private final String password;

    private UserLogInDto(String usernameOrEmail, String password) {
        this.usernameOrEmail = usernameOrEmail;
        this.password = password;
    }

    public String getUsernameOrEmail() { return usernameOrEmail; }

    public String getPassword() { return password; }

    @JsonPOJOBuilder
    public static class UserLogInDtoBuilder{

        private String usernameOrEmail;
        private String password;

        private UserLogInDtoBuilder() {
        }

        public static UserLogInDtoBuilder builder() { return new UserLogInDtoBuilder(); }

        public UserLogInDtoBuilder withUsernameOrEmail(String usernameOrEmail) {
            this.usernameOrEmail = usernameOrEmail;
            return this;
        }

        public UserLogInDtoBuilder withPassword(String password) {
            this.password = password;
            return this;
        }

        public UserLogInDto build() { return new UserLogInDto(usernameOrEmail, password); }
    }

}
```

3.4 GEOMETRIC-APP

Per desenvolupar aquesta aplicació he utilitzat npm com ja he explicat abans. Per començar el projecte des d'una plantilla, amb una estructura bàsica de carpetes i amb la configuració bàsica per utilitzar React i TypeScript, tenim els següents comandos: un per instal·lar l'aplicació capaç de muntar l'esquelet del projecte i l'altre comando per utilitzar aquesta aplicació:

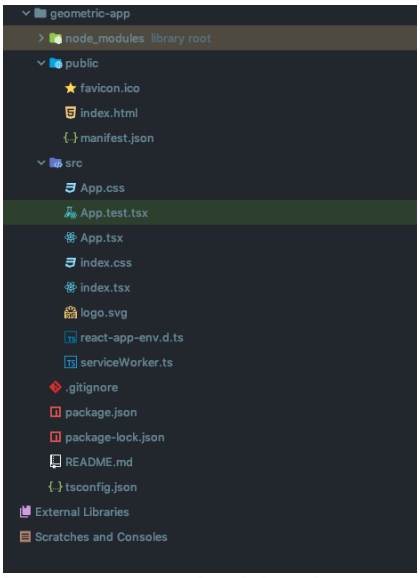
$sudo npm install –g create-react-app
$npx create -react-app geomètric-app –typescript

Si a l'utilitzar create-react-app no s'instal·len els tipus següents, ho hem de fer nosaltres utilitzant el següent comando:

$npm install typescript @types/node @types/react @types/react-dom @types/jest --save

A l'escriure el codi amb TypeScript, sempre necessitem els tipus pels objectes; per tant, quasi cada llibreria que instal·lem haurem d'instal·lar també els tipus corresponents.

Això crearà la següent estructura de carpetes i alguns fitxers bàsics per deixar el projecte llest per arrencar, tot i que de moment no fa res.



Imatge 26: esquelet del projecte geometric-app

Un dels fitxers és el serviceWorker.ts , un servidor de Node.js molt bàsic, però funcional al 100%, i com que en aquest cas no fa falta que connecti amb cap base de dades, ja em serveix tal i com està. Només ha de respondre a una ip i port, per donar tota la vista i arxius JavaScript al navegador.

Un cop tenim aquesta estructura, ens posem dins la carpeta geometric-app i podem afegir més mòduls per desenvolupar l'aplicació sencera.

Com ja he dit, l'estat de l'aplicació estarà controlat per Redux. Per instal·lar Redux utilitzem els següents comandos: un per instal·lar-lo i l'altre per instal·lar tipus que necessitem a l'escriure el codi en TypeScript:

$npm install redux react-redux --save
$npm install @types/react-redux --save

Redux per si sol sempre crearà accions pures, per tant no és capaç de fer connexions HTTP; ja que no pot executar funcions asíncrones que retornin altres funcions. Però Redux-Thunk sí, és el que s'anomena un middleware. És com una extensió de Redux, per tant també hem d'instal·lar redux-thunk, de la mateixa manera que abans amb els dos comandos següents:

$npm install redux-thunk --save
$npm install @types/redux-thunk --save

A més a més com que l'aplicació haurà de tractar amb JWT també afegirem una llibreria capaç de fer-ho.

$npm install jsonwebtoken --save
$npm install @types/jsonwebtoken –save

També he utilitzat una llibreria per fer el formulari del signup que permet mostrar missatge en el propi formulari (formik), després de fer unes comprovacions (yup)

$npm install formik --save
$npm install yup --save
$npm install @types/yup --save

Per dibuixar les figures he utilitzat konva-react i una paleta de colors de la llibreria react-color.

$npm install react konva react-konva --save
$npm install react-color –save
$npm install @types/react-color --save

Per fer els botons i els formularis amb una mica de disseny sense tocar el CSS:

$npm install bootstrap react-bootstrap --save
$npm install @types/react-bootstrap –save

No és necessari posar tots els comandos a l'inici; pots anar desenvolupant l'aplicació i en el moment que necessites una dependència, agregar-la amb el comando npm. Quan ho fas, la descarrega i agrega a les dependències del document package.json i Package-lock.json

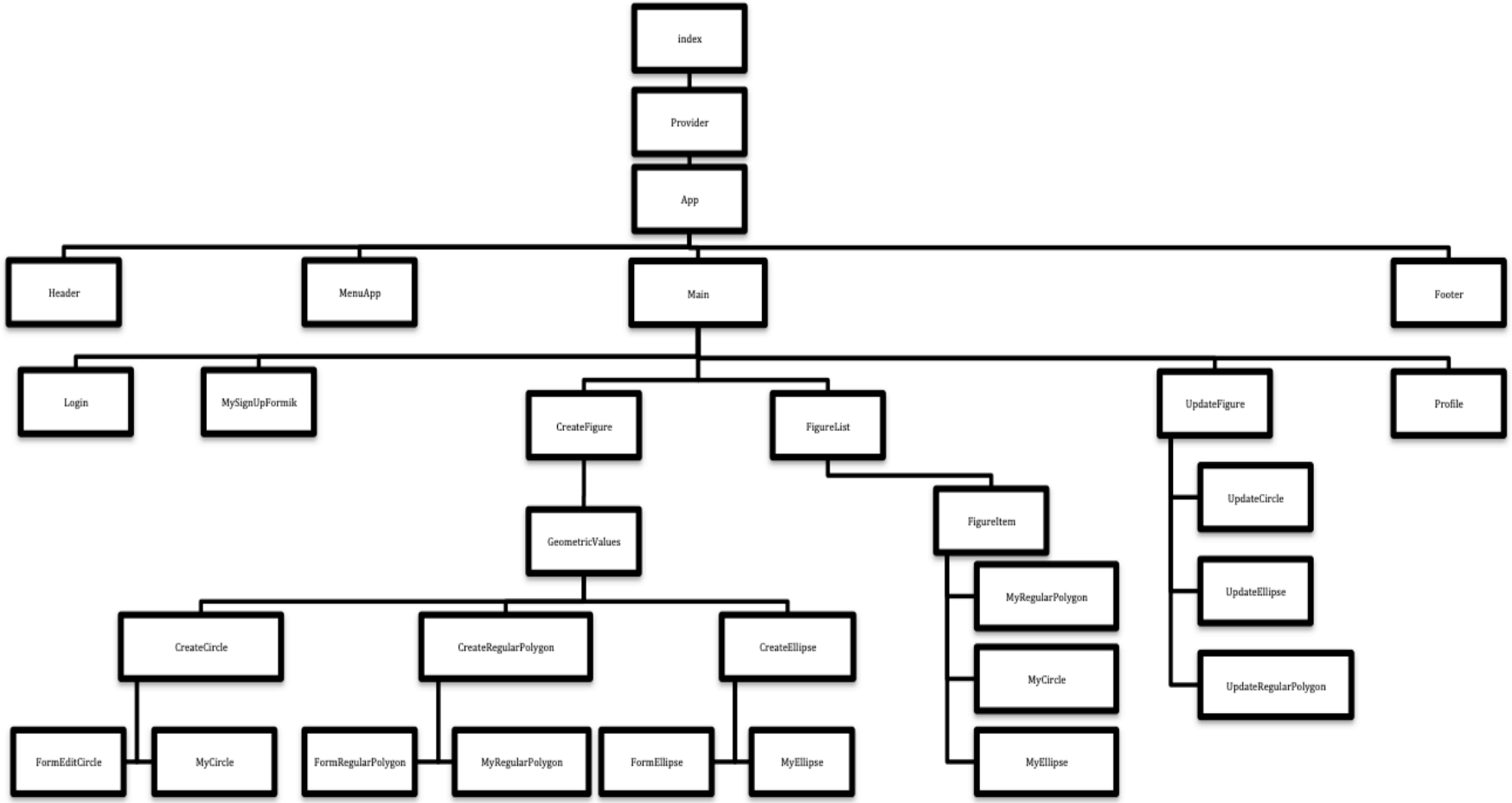
El front-end l'explicaré des de dos punts de vista:

- React, que són els components de la UI
- Redux, que fa el control de l'estat de l'aplicació

3.4.1 React

Des del punt de vista de React, tenim diferents components. Es pot dir que cada component és una part independent i reutilitzable de la UI de la web. Com que el control de l'estat el fa Redux, la majoria d'aquests components són molt simples, no son ni classes; ja que no són ells els encarregats d'actualitzar el seu propi estat.
Si no s'utilitza Redux , cada component es una classe (stateful) que "extends" de React.Component, però en aquest cas són variables del tipus const que aprofitant ECMAScript són una funció lambda del tipus React.FunctionComponent i no tenen estat propi (stateless)

Podem dibuixar un arbre de com estan posats els components per entendre millor com es fa la composició per aconseguir la SPA.



Imatge 27: Arbre dels Components

3.4.1.1 React.FuncionalComponent

Per exemple el component FigureList que és del tipus React.FunctionComponent. Aquest component rep una llista de figures de l'estat de Redux i passa un objecte d'aquesta llista cap al component inferior.

```
import ...

interface IProps {
    geometricListState: ListFiguresState,
    dispatch: ThunkDispatch<any,any,AnyAction>,
}

const FigureList: React.FunctionComponent<IProps> = (props: IProps) => {

    const {dispatch} = props;
    const {listFigures} = props.geometricListState;

    return(
        <div>
            <h3>Figures</h3>
            <ul>
                {listFigures.map( callbackfn: (figure: Figure) =>
                    <div key={figure.id}>
                        <FigureItem figure={figure}/>

                        <Button variant="primary" onClick={() => dispatch(fetchDeleteFigure(figure.id))}>
                            Delete
                        </Button>
                        <br/>
                        <br/>
                        <Button variant="primary" onClick={() => {
                            // console.log(figure.id);
                            dispatch(setType(figure.type));
                            dispatch(setTmpFigure(figure));
                            dispatch(showUpdateFigure())
                        }}>
                            Edit
                        </Button>
                        <br/>
                        <br/>
                    </div> )
                }
            </ul>
        </div>
    );
};

const mapStateToProps = (state: AppState) => ({
    geometricListState: state.getGeometricList
});

export default connect(mapStateToProps)(FigureList);
```

Imatge 28: exemple de React.FunctionalComponent

Primer de tot en cada component hem de definir com seran les seves props amb una interfície que sempre anomeno IProps. Aquestes props vindran a través de Redux amb les dues funcions del final mapStateToProps i connect que explicaré després.

Cada component definit com a FunctionComponent només depèn de les seves props que els i passa Redux des de l'estat global; per tant és una funció pura que canvia depenent de l'entrada.

Podem observar que no és una classe i que FigureList és una variable del tipus const que executa una funció lambda. Aquesta funció és molt simple: només depèn de props, de la qual obté la funció dispatch i la llista de figures. Un cop fet això, retorna una expressió JSX que es un amb un títol i una llista HTML. Aquesta llista es crea a partir del mètode map sobre la llista de figures obtinguda de props. Una vegada més utilitzo una funció lambda, i a partir de cada figura creo un element per la llista HTML amb un que inclou el component FigureItem, al qual li passem la figura que ha de renderitzar i els dos botons per poder eliminar o editar aquesta figura.

Aquests dos botons també estan fets utilitzant funcions lambda que criden la funció dispatch per passar accions a Redux.

3.4.2 Redux

Una aplicació React té un sol document HTML, l'index.html que està guardat en la carpeta públic. Aquest té un div amb el id=root que és on introduirem els nostres components, un dins l'altre per anar creant l'arbre de components que formen la SPA.
L únic component que hi a dins el div root es <Root /> que esta definit dins el index.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import './styles/index.css';
import App from './components/App';
import * as serviceWorker from './serviceWorker';
import { Provider } from 'react-redux'
import configureStore from "./redux/store/indexStore";

let store = configureStore();

let rootElement = document.getElementById( elementId: 'root');

const Root = () => (
    <Provider store={store}>
        <App />
    </Provider>
);

ReactDOM.render(<Root/>, rootElement);


// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Imatge 29: index.tsx de geometric-app

El component <Root /> té a dins dos components més: el <Provider /> i <App/>

Aquests components són molt importants:

- Provider defineix tots els elements que tenen accés a Redux; ja que és qui te l'Store, que és el "magatzem" amb l'estat global de l'aplicació. Per tant tots els components que estan dins del Provider, poden accedir a funcions que connecten amb Redux.

- App es el component principal de la UI

Podem veure el estat com un objecte molt gran que guarda tota la informació de l'aplicació. El podem representar com estaria definit en TypeScript si ajuntéssim totes les interfícies que el defineixen

Forma de l'estat de geomètric-app

```
AppState:{
  AuthState: {
    isFetching:boolean,
    isAuthenticated:boolean,
    loginMessage:string
  },
  SignupState: {
    isFetching:boolean,
    signupMessage:string,
  },
  FilterState: {
    visibilityFilter:string,
  },
  TmpFigureState: {
   selectType: string
   figure: Figure,
}
  ListFiguresState: {
    isFetching:boolean,
    listFigures:Figure [ ],
  },
  ProfileState: {
    isFetching:boolean,
    message:string,
    profile: {
      username:string,
      email:string,
      roles:string[],
      personalData: {
        firstName:string,
        lastName:string,
      }
    }
  }
}
```

Redux es basa en un flux de dades unidireccional, que ja hem vist abans (Imatge 9, pagina19) on podem veure que UI connecta amb Redux a través de una funció anomenada dispatch que llença accions; aquestes són recollides pels reducers que actualitzen l'estat global de l'aplicació a la store. Cada component que necessita dades de l'estat global utilitza dues funcions que el connecten a Redux:

- mapStateToProps: amb aquesta funció indiquem quina part de l'estat global passem a les props del component; sempre que utilitzem aquesta funció també li passarà la funció dispatch.

- connect: aquesta funció fa que cada cop que l'estat sigui modificat en el store executarà la funció mapStateToProps. De manera que si una part de l'estat que afecta el component ha canviat, actualitzarà les dades de les props i tornarà a renderitzar aquell component.

Com he comentat abans, si només utilitzem Redux, només es podria fer dispatch de creadors d'accions. Aquests creadors d'accions són funcions pures que retornen un objecte (acció) per actualitzar l'estat.

Les accions són objectes immutables i els creadors d'accions són funcions pures que retornen un objecte acció. L'estat global només pot ser actualitzat per accions, però si volem actualitzar l'estat a través d'una API REST necessitem poder fer dispatch de funcions asíncrones; funcions que criden altres funcions i que realitzin connexions a serveis REST. Depenent de la resposta, crearan una acció o una altra per actualitzar l'estat.

Això és possible gràcies a redux-thunk, que és el que s'anomena un middleware.
Redux-thunk és una llibreria molt petita, de fet és una sola funció, però permet que redux sigui molt més potent del que seria per ell sol.

Per actualitzar un estat que depèn d'una connexió a una API REST és necessari modelar els diferents estats de la connexió en diferents accions. Jo l'he modelat en tres parts: Request, Success i Error.

Anem a veure un exemple de codi complet des que fem dispatch per guardar una figura nova, fins que aquesta està guardada en l'estat de la store.

3.4.2.1 Accions

Primer de tot, en un punt del codi tenim un botó com aquest

```
<div>
    <MyCircle radius={this.state.radius} color={this.state.color}/>
    <Button variant="primary"
            onClick={() => {
                figure.color = this.state.color;
                figure.radius = this.state.radius;
                dispatch(fetchFigure(figure));
            }}>
        Save
    </Button>
</div>
```

Imatge 30: funcio dispatch de Redux

Quan fem clic en el boto, utilitzo una funció lambda que guarda les dades en la figura i crida la funció fetchFigure passant-li l'objecte que volem guardar.

La funció és aquesta:

```
export function fetchFigure(figure: Figure): ThunkAction<void, AppState, null, Action<string>> {

    return async (dispatch: any) => {

        await updateTokens(dispatch);
        const typeToken: any = localStorage.getItem( key: 'token_type') || null;
        const accessToken: any = localStorage.getItem( key: 'access_token') || null;
        const authHeader: string = typeToken + " " + accessToken;

        const json = {
            figure: figure,
        };

        let config: RequestInit = {
            method: 'POST',
            mode: 'cors',
            headers: {
                'Authorization': authHeader,
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(json),
        };

        dispatch(requestSaveFigure());

        const response = await fetch( input: 'http://localhost:8081/api/geometric/figure', config);

        if (!response.ok) {
             const text = await response.text();
             dispatch(errorSaveFigure(text));
        } else {
            const figureWithId: any = await response.json();
            dispatch(successSaveFigure(figureWithId));
            dispatch(showList());
        }
    }
}
```

Imatge 31: ThunkAcction de Redux

Aquesta funció retorna una altra funció que és asíncrona, i que crida altres funcions. Com que és una funció asíncrona utilitzo el await per fer que en algun moment esperi la resposta per seguir executant codi. És la manera de controlar l'execució asíncrona quan necessites els valors per continuar; això es una característica de ECMAScript7 que puc utilitzar perquè el codi està escrit amb TypeScript, això fa que el codi sigui molt més clar i llegible.
Primer de tot crido una funció per actualitzar els token, aquesta funció comprova que el temps d'expedició de l'accésToken no estigui caducat; si ho està, utilitzarà el refreshToken per obtenir uns nous tokens i si aquest tampoc

es vàlid, farà un logout, obligant a l'usuari a tornar a fer login utilitzant les credencials.
Després d'això obtinc el token del localstorage, creo un JSON amb la figura i preparo la petició REST creant un objecte amb tota la informació necessària per fer aquesta comunicació.
A continuació faig un dispatch d'una acció pura per actualitzar l'estat, això simplement és per indicar en quin punt de la comunicació passem, en aquest moment comença el request. Utilitzo l'API fetch per fer la petició REST, indicant amb un await que no segueixi fins a tenir la resposta. Després comprovo el status HTTP de la resposta i depenent de si és un 200 (ok tot correcte) o no, faig un dispatch de l'acció pura que indica error o succés. En el cas que sigui correcte també faig un dispatch per actualitzar la vista.

Ara veurem les accions pures utilitzades abans:

```
function requestSaveFigure(): GeometricListActions {
    return {
        type: SAVE_FIGURE_REQUEST,
        isFetching: true,
    }
}

function successSaveFigure(figure: Figure): GeometricListActions {
    return {
        type: SAVE_FIGURE_SUCCESS,
        isFetching: false,
        figure,
    }
}

function errorSaveFigure(message: string): GeometricListActions {
    return {
        type: SAVE_FIGURE_ERROR,
        isFetching: false,
        message,
    }
}
```

Imatge 32: Acction de Redux

Són funcions pures i realment molt simples. Podríem crear l'objecte en cada punt de codi, però això té el perill d'introduir errors; per tan, utilitzem aquestes funcions que s'anomenen creadors d'accions.

Quan fem un dispatch d'una acció d'aquestes, és enviada cap al reducer, i l'objecte que crea és utilitzat per actualitzar l'estat de l'aplicació de la següent manera:

3.4.2.2 Reducers

Un reducer també és una funció pura la qual a partir d'un estat i una acció, retorna un nou estat. És important entendre que retorna un nou estat no actualitza l'estat anterior; ja que si fes això, ja no seria pura.

```
const initialGeometricList: ListFiguresState = {
    isFetching: false,
    listFigures: []
};

export function setGeometricList(state = initialGeometricList, action: GeometricListActions): ListFiguresState {
    switch (action.type) {
        // DURANT EL REQUEST PODRIEM POSSSAR UNA BARRA DE CARREGA
        case LIST_REQUEST:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                listFigures: []
            });
        case LIST_SUCCESS:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                listFigures: action.listFigures
            });
        case LIST_ERROR:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                listFigures: []
            });
        case SAVE_FIGURE_REQUEST:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
            });
        case SAVE_FIGURE_SUCCESS:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                listFigures: [
                    ...state.listFigures,
                    action.figure
                ]
            });
        case SAVE_FIGURE_ERROR:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                message: action.message
            });
        case DELETE_FIGURE_SUCCESS:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                listFigures: state.listFigures.filter(
                    callbackfn: (figure: Figure) => {return figure.id !== action.idFigure }
                    )
            });
        case DELETE_FIGURE_ERROR:
            return Object.assign( target: {}, state, source2: {
                isFetching: action.isFetching,
                message: action.message
            });
        default:
            return state;
    }
}
```

Imatge 33: exemple de reducer de Redux

Aquest reducer té la funció d'actualitzar la part de l'estat que manté la llista de figures (ListFiguresState).

Primer de tot creo un estat inicial per si és el primer cop que executem el reducer, en el qual la variable isFetching és falsa i la llista està buida. És l'estat per defecte.

En l'exemple que estem veient, pot rebre tres accions que tenen a veure amb quan guardem una figura nova, pero també actualitzarà aquesta part de l'estat quan eliminem una figura o quan obtenim la llista del servidor.

Com que es una acció que depèn d'una connexió amb el servidor, sempre actualitzarem la variable isFetching per indicar si està a mitja connexió o no. Si el resultat es error, guardarem el missatge d'error que hem obtingut. Si el resultat és succés, actualitzarem la llista.

Per fer això utilitzo el mètode Object.assign passant en el primer paràmetre un objecte buit per indicar que vull un objecte nou. El que fa aquest mètode és copiar els valors de l'objecte que passem com a segon paràmetre a un nou objecte, menys els paràmetres que indiquem en l'objecte que passem com a tercer paràmetre.

En el cas de succés, per crear un nou array amb un element més, utilitzo el mètode spread (que es representa amb 3 punts davant del array al qual vols que operi) que agafa una array i l'expandeix. Un cop expandit introdueixo el nou element, que ve com a part de l'acció al final d'aquest nou array.

En una aplicació tenim molts tipus d'accions i varis reducers que es cuiden d'actualitzar diferents parts de l'estat, ja que si no seria un caos de codi.

3.4.2.3 Store

Al "final" del Flux tenim la store que és qui guarda l'estat global d'l'aplicació i combina el resultat de tots els reducers.

```
import {applyMiddleware, combineReducers, createStore, Store} from "redux";
import thunkMiddleware from "redux-thunk";
import {setAuth} from "../reducers/authReducer";
import {setSignup} from "../reducers/signupReducer";
import {setVisibilityFilter} from "../reducers/visibilityFilterReducer";
import {setGeometricList} from "../reducers/geometricListReducer";
import {setProfile} from "../reducers/profileReducer";

const rootReducer = combineReducers( reducers: {
    getAuth: setAuth,
    getSignup: setSignup,
    getVisibilityFilter: setVisibilityFilter,
    getGeometricList: setGeometricList,
    getProfile: setProfile,
});

export type AppState = ReturnType<typeof rootReducer>;

export default function configureStore(): Store {

    const createStoreWithMiddleware = applyMiddleware(thunkMiddleware)(createStore);

    return createStoreWithMiddleware(rootReducer);
}
```

Imatge 34: store de Redux

Aquí tenim definit el rootReducer que és la combinació de tots els reducers que té l'aplicació. També definim l'estat global de l'aplicació com AppState que és del tipus del rootReducer. I tenim la funció que configura la Store, amb el middleware de redux-thunk i el rootReducer. Aquesta última funció és la que passem al Provider a l'inici de l'aplicació per connectar tots els components amb l'estat global.

3.4.3 Polimorfisme en TypeScript

Ara explicaré com es poden definir els tipus d'objecte propis amb TypeScript i com pot fer el que seria polimorfisme, ja que ho he necessitat en varis punts de l'aplicació

Per exemple, l'objecte que defineix la figura no sempre és igual; però sempre és un objecte del tipus Figure:

```
export const CIRCLE = 'CIRCLE';
export const REGULARPOLYGON = 'REGULARPOLYGON';
export const ELLIPSE = 'ELLIPSE';


export interface Circle {
    id?: string,
    type: typeof CIRCLE,
    color: string,
    radius: number,
}

export interface RegularPolygon {
    id?: string,
    type: typeof REGULARPOLYGON,
    color: string,
    sides: number,
    radius: number,
}

export interface Ellipse {
    id?: string,
    type: typeof ELLIPSE,
    color: string,
    radiusX: number,
    radiusY: number,
}

export type Figure = Circle | RegularPolygon | Ellipse;
```

Imatge 35: exemple 1 de polimorfisme en TypeScript

Per fer el polimorfisme de la Figure, primer és necessari crear cada interfície que defineix cada tipus que el formaran, en aquest cas: Circle, RegularPolygon i Ellipse. Entre ells comparteixen que tots tenen id, type i color, però no tenen la mateixa forma geomètrica; per tant els altres atributs no els comparteixen.
Per definir el type de cada interfície utilitzem el mètode de typeof de TypeScript i assignem un nom que és un String definit en una const. Finalment crearem un nou type Figure dient que és igual a una llista amb les tres interfícies definides.
Dins el codi, si fem una comprovació de figure.type, ja sigui amb un if o un switch a dins d'aquell scope, el propi IDE ja detectarà la figure com al tipus que és realment.

De la mateixa manera és necessari crear tipus polimòrfics per les accions de Redux

```
export const SAVE_FIGURE_REQUEST = 'SAVE_FIGURE_REQUEST';
export const SAVE_FIGURE_SUCCESS = 'SAVE_FIGURE_SUCCESS';
export const SAVE_FIGURE_ERROR = 'SAVE_FIGURE_ERROR';

export interface SaveFigureRequest extends Action{
    type: typeof SAVE_FIGURE_REQUEST,
    isFetching: boolean,
}


export interface SaveFigureSuccess extends Action{
    type: typeof SAVE_FIGURE_SUCCESS,
    isFetching: boolean,
    figure: Figure
}

export interface SaveFigureError extends Action{
    type: typeof SAVE_FIGURE_ERROR,
    isFetching: boolean,
    message: string,
}
export type GeometricListActions =
    IListRequest | IListSuccess | IListError |
    SaveFigureRequest | SaveFigureSuccess | SaveFigureError |
    DeleteFigureRequest | DeleteFigureSuccess | DeleteFigureError;
```

Imatge 36: exemple 2 de polimorfisme en TypeScript

He definit els tipus d'acció utilitzats en Redux per guardar una figura, però aquí la diferència està en que cada una d'elles extends de l'interfície Action i que el tipus final engloba a molts més tipus d'acció.

3.5 GEOMETRIC-RESOURCES

Aquest és el tercer servidor. Com l'idProvider és un projecte Maven fet amb el framework SpringBoot i programat amb Java 8. També gestiona una base de dades MongoDB on guarda la informació de les figures que ha creat cada usuari.

És molt més simple que el primer, ja que nomes té la clau pública del idProvider per validar els tokens i donar per autenticat l'usuari. De la mateixa manera que ho fa el idProvaider, té un Filter per on passen totes les sol·licituds i llegint el JWT carrega un usuari amb la informació que porta el JWT. La informació més important per aquest servidor és l'id de l'usuari per buscar les seves figures en la base de dades.

3.5.1 Base de dades

Les dues bases de dades són independents una de l'altra, però evidentment un usuari que no està registrat en el idProvider no pot tenir figures; per tant els id d'usuari de la base de dades del geomètric-resources, abans han d'estar registrats al idProvider.


```
{
    "_id" : ObjectId("5cf6b58abde58ee064c657f0"),
    "figures" : [
        {
            "_id" : ObjectId("5d02a895bde58e821685b43f"),
            "type" : "CIRCLE",
            "radius" : 100,
            "color" : "#339d2f",
            "_class" : "com.tfg.geometricresources.model.Circle"
        },
        {
            "_id" : ObjectId("5d02a8abbde58e821685b440"),
            "type" : "REGULARPOLYGON",
            "sides" : 7,
            "radius" : 120,
            "color" : "#8a7a7a",
            "_class" : "com.tfg.geometricresources.model.RegularPolygon"
        },
        {
            "_id" : ObjectId("5d02a8bbbde58e821685b441"),
            "type" : "ELLIPSE",
            "radiusX" : 120,
            "radiusY" : 60,
            "color" : "#147982",
            "_class" : "com.tfg.geometricresources.model.Ellipse"
        }
    ],
    "_class" : "com.tfg.geometricresources.model.UserResources"
}
```


Imatge 37: exemple de la MongoDB de geometric-resources

Podem observar que els objectes que es guarden en aquesta base de dades no són tots iguals, però això no és un problema en una MongoDB.

3.5.2 API RESTful

Aquest servidor exposa una API RESTful, aquí utilitzant el mateix endpoint però amb mètodes HTTP diferents, podem fer totes les accions de CRUD sobre la base de dades.

URL completa	Mètode REST	Mètode Java
ip:port/api/geometric/figure	GET	getListFigures()
ip:port/api/geometric/figure	POST	createFigure(@Valid @RequestBody FigureDto figureDto)
ip:port/api/geometric/figure	PUT	updateFigure(@Valid @RequestBody FigureDto figureDto)
ip:port/api/geometric/figure	DELETE	deleteFigure(@Valid @RequestBody IdDto idFigure)

4. RESULTATS

Avantatges i inconvenients de l'autenticació basada en JWT: Sense estat, escalable i desacoblat.

El servidor ja no guarda informació de quins usuaris estan connectats o quins tokens ha firmat. Cada sol·licitud que fa l'usuari a un servidor va acompanyada del token i el servidor verifica l'autenticitat de la sol·licitud basant-se únicament amb el token.
D'aquesta manera el servidor només ha de firmar tokens quan un usuari fa un login i validar si aquell token encara es vàlid quan se li proporciona un token.

Cross Domain i CORS
El Cross Domain és un mecanisme de seguretat que evita que una aplicació pugui accedir a un servidor diferent del quin està allotjat la pròpia aplicació, i és necessari que estigui desactivat. Això es fa per evitar el Cross Site Request Foregery (CSRF) i el Cross Site Scripting (XSS).

- CSRF és quan un lloc web maliciós suplanta a l'usuari que està navegant i accedeix a una altra aplicació en el seu nom.

- XSS és quan un lloc web maliciós roba informació i l'envia a tercers. Això implica que ha d'existir una injecció de JavaScript

De forma predeterminada, React DOM escapa tots els valors incrustats en JSX abans de representar-los, per tal de garantir que mai es pugui injectar res que no estigui explícitament escrit a l'aplicació. Tot es converteix a una cadena abans de ser processat. Això ajuda molt a la prevenció d'atacs XSS.

CSRF és realment un problema si parlem de cookies o guardem el token en una cookie, ja que aquestes són enviades per defecte a totes les peticions cap al servidor.
Per tant, aquest segon problema és més important en l'us de cookies que en els tokens, ja que el guardem en el localstorage del navegador.

Si utilitzes un sistema d'autenticació basat en tokens i habilites el CORS, pots accedir a uns quants dominis sense problema, mentre tinguis un token vàlid podràs rebre resposta de varis servidors. Per tant, pots muntar un sistema basat en microserveis on tens diferents servidors que s'encarreguen de diversos serveis per una mateixa aplicació. És una bona pràctica utilitzar un token per autenticar l'usuari en aquest tipus d'arquitectura.

El problema real és si un tercer arriba a fer-se amb el token, ja que pot autenticar-se amb el teu usuari i obtenir les teves dades. És difícil que passi si tenim en compte que ens hem previngut sobre els atacs XSS, però tot i això pot passar si et deixes un ordinador obert amb el teu usuari connectat . Si et roben l'accessToken, tenen accés durant una estona, però no molta. Per això és recomanable donar una vida curta a aquest tipus de token. En canvi els refreshToken tenen un temps de vida bastant més llarg ; per això sempre has de tenir una forma d'invalidar-los.

5. PRESSUPOST

He invertit unes 600hores entre investigació i desenvolupament, a un preu d'enginyer junior, aproximadament 10€ /hora.

$$600\ x\ \frac{10€}{hora} = 4800€$$

El software que he utilitzat és pràcticament tot gratuït menys els IDE utilitzats que tenen un cost per llicència anual de 649€ tot i que actualment estic utilitzant la llicència per estudiants que es pot obtenir al ser alumne de la UPC.

Per tant el total seria:

$$4800 + 649 = 5449€$$

6. CONCLUSIONS

Veient els resultats, es pot dir que l'autenticació basada en JWT és una bona opció, però per si sola no té suficient seguretat com per ser utilitzada en segons quines aplicacions on s'accedeix a dades més sensibles; ja que només es pot verificar si el JWT ha estat modificat, però no si ha estat robat. De la mateixa manera, no podem invalidar accessToken un cop ha estat firmat i entregat a l'usuari; només queden invalidats passat el temps d'expiració.

Per solucionar això, es podria donar un identificador únic a cada JWT i que sempre que fos utilitzat en un servidor, aquest preguntés a l'idProvider si encara és vàlid aquell token. D'aquesta manera podríem invalidar tokens emesos, però el sistema deixaria d'estar totalment desacoblat i afegiríem un petit retard, ja que sempre estaríem fent una petició a l'idProvider per comprovar el JWT.

Una altra opció podria ser posar dins el JWT la informació de quina aplicació -ja sigui mòbil o navegador web- i el nom de la màquina que l'està executant. Si aquesta no coincideix a l'utilitzar el JWT, no validar l'autenticació de l'usuari.

Tot i aquest problema, has de tenir confiança en la teva aplicació i en el teu idProvider, sobretot intentant fer una aplicació molt robusta i que no tingui vulnerabilitats per on et puguin injectar codi maliciós i ser víctima d'un atac XSS.
Configurant bé els temps d'expiració dels JWT, fent l'accessToken de 20 o 30 minuts i refreshToken de molt més -ja sigui 24 hores o una setmana- i has de tenir clar que un cop utilitzat queda invalidat.
És un bon sistema d'autenticació d'usuaris, molt ràpid i simple, amb poques connexions a la base de dades per conèixer la identitat de l'usuari i amb suficient seguretat per moltes aplicacions que no impliquen dades confidencials com poden ser dades bancàries o mèdiques.

7. BIBLIOGRAFIA

[1] Maven 3.6.1 https://maven.apache.org/guides/getting-started/index.html

[2] Spring Boot 2.1.4 https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[3] Spring Security https://docs.spring.io/spring-security/site/docs/5.1.4.RELEASE/reference/htmlsingle/#jc-method

[4] MongoDB 4.0 https://docs.mongodb.com/manual/tutorial/getting-started/

[5] keytool - Key and Certificate Management Tool https://docs.oracle.com/javase/6/docs/technotes/tools/windows/keytool.html

[6] Node & npm https://www.npmjs.com/get-npm

[7] TypeScript https://www.typescriptlang.org/docs/home.html

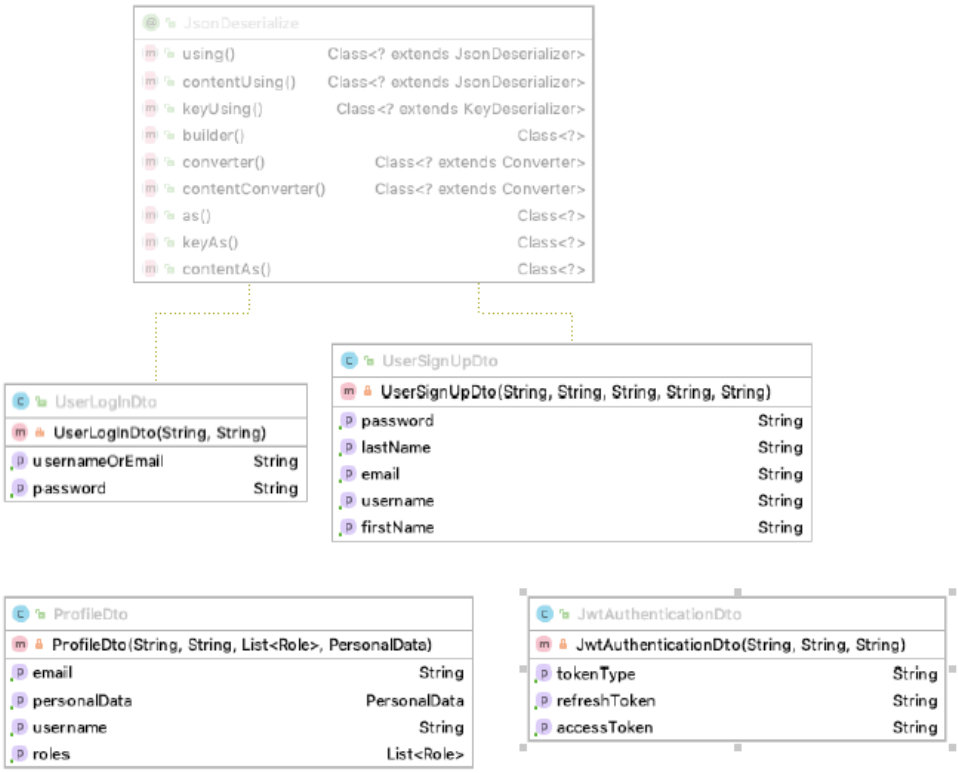
[8] React https://reactjs.org/docs/getting-started.html

[9] Redux https://redux.js.org/introduction/getting-started

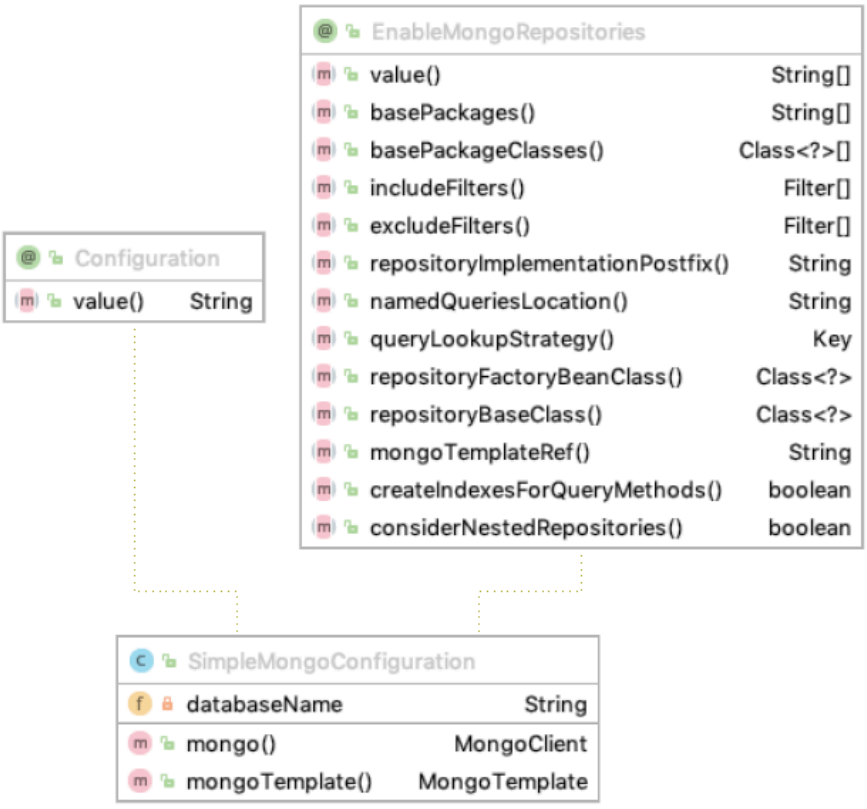
[10] JWT & JWT Java https://jwt.io/#debugger-io https://github.com/auth0/java-jwt

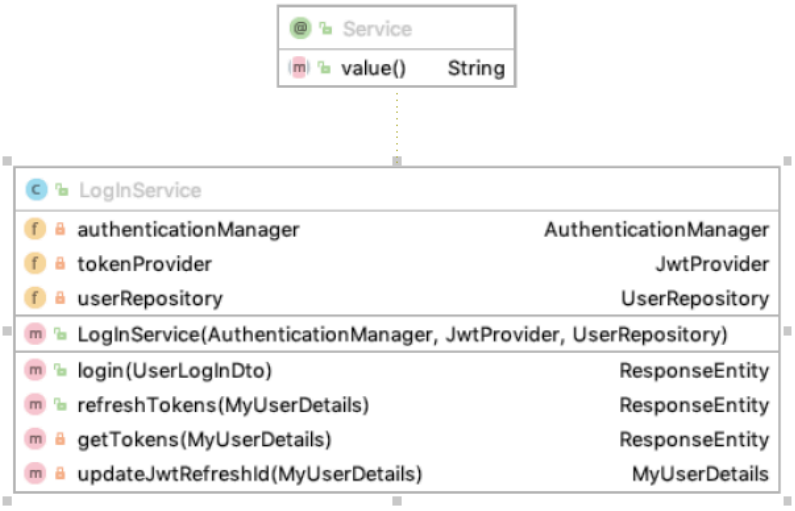
8. APÈNDIXS (Opcional)

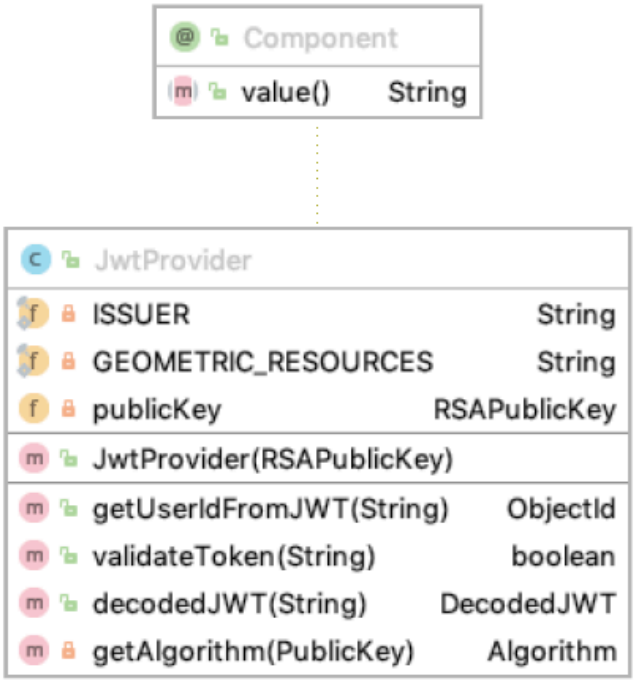
8.1 DIAGRAMAS UML idProvider.

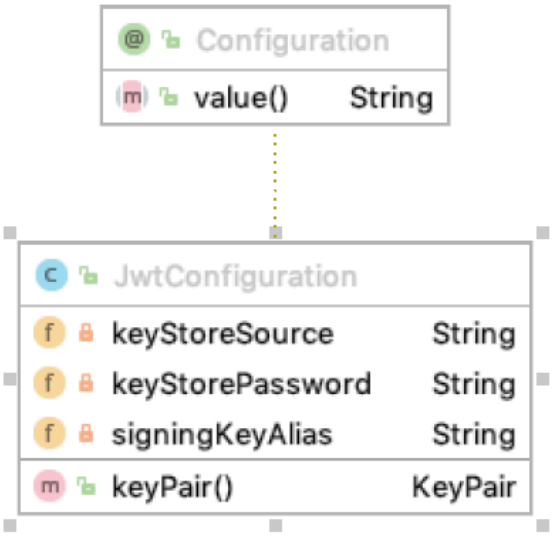

JsonDeserialize
using() Class<? extends JsonDeserializer>
contentUsing() Class<? extends JsonDeserializer>
keyUsing() Class<? extends KeyDeserializer>
builder() Class<?>
converter() Class<? extends Converter>
contentConverter() Class<? extends Converter>
as() Class<?>
keyAs() Class<?>
contentAs() Class<?>
UserLogInDto
UserLogInDto(String, String)
usernameOrEmail String
password String
UserSignUpDto
UserSignUpDto(String, String, String, String, String)
password String
lastName String
email String
username String
firstName String
ProfileDto
ProfileDto(String, String, List<Role>, PersonalData)
email String
personalData PersonalData
username String
roles List<Role>
JwtAuthenticationDto
JwtAuthenticationDto(String, String, String)
tokenType String
refreshToken String
accessToken String

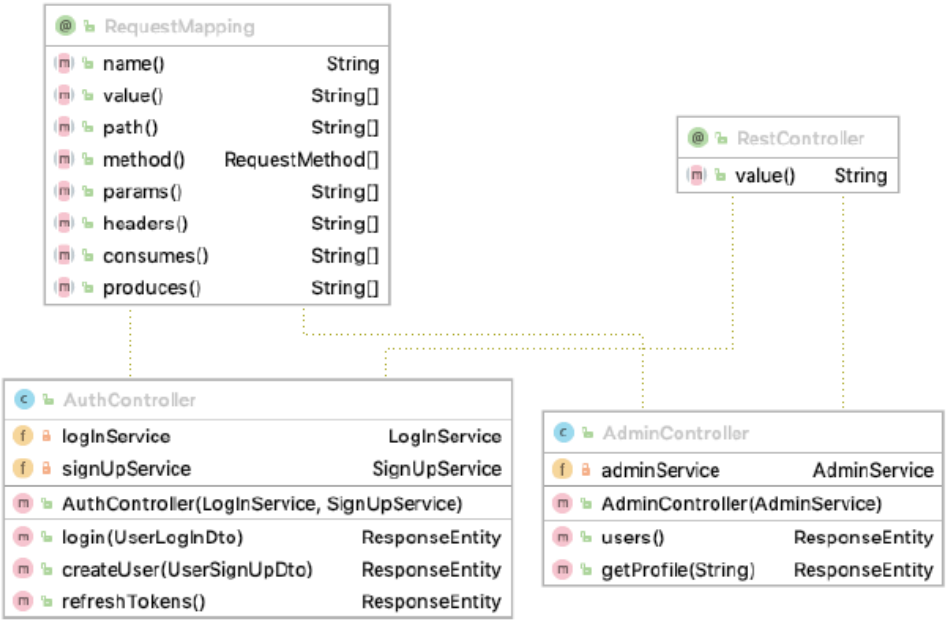

SecurityBuilder
build() O
SecurityConfigurer
init(B) void
configure(B) void
Order
value() int
WebSecurityConfigurer
Configuration
value() String
WebSecurityConfigurerAdapter
logger Log
context ApplicationContext
contentNegotiationStrategy ContentNegotiationStrategy
objectPostProcessor ObjectPostProcessor<Object>
authenticationConfiguration AuthenticationConfiguration
authenticationBuilder AuthenticationManagerBuilder
localConfigureAuthenticationBldr AuthenticationManagerBuilder
disableLocalConfigureAuthenticationBldr boolean
authenticationManagerInitialized boolean
authenticationManager AuthenticationManager
trustResolver AuthenticationTrustResolver
http HttpSecurity
disableDefaults boolean
WebSecurityConfigurerAdapter()
WebSecurityConfigurerAdapter(boolean)
configure(AuthenticationManagerBuilder) void
getHttp() HttpSecurity
authenticationManagerBean() AuthenticationManager
authenticationManager() AuthenticationManager
userDetailsServiceBean() UserDetailsService
userDetailsService() UserDetailsService
init(WebSecurity) void
configure(WebSecurity) void
configure(HttpSecurity) void
createSharedObjects() Map<Class<? extends Object>, Object>
trustResolver AuthenticationTrustResolver
contentNegotiationStrategy ContentNegotiationStrategy
objectPostProcessor ObjectPostProcessor<Object>
applicationContext ApplicationContext
authenticationConfiguration AuthenticationConfiguration
EnableWebSecurity
debug() boolean
EnableGlobalMethodSecurity
prePostEnabled() boolean
securedEnabled() boolean
jsr250Enabled() boolean
proxyTargetClass() boolean
mode() AdviceMode
order() int
WebSecurityConfiguration
mongoUserDetailsService MongoUserDetailsService
unauthorizedHandler JwtAuthenticationEntryPoint
jwtProvider JwtProvider
WebSecurityConfiguration(MongoUserDetailsService, JwtAuthenticationEntryPoint, JwtProvider)
configure(HttpSecurity) void
passwordEncoder() PasswordEncoder
jwtAuthenticationFilter() JwtAuthenticationFilter
configure(AuthenticationManagerBuilder) void
authenticationManagerBean() AuthenticationManager


EnableMongoRepositories
value() String[]
basePackages() String[]
basePackageClasses() Class<?>[]
includeFilters() Filter[]
excludeFilters() Filter[]
repositoryImplementationPostfix() String
namedQueriesLocation() String
queryLookupStrategy() Key
repositoryFactoryBeanClass() Class<?>
repositoryBaseClass() Class<?>
mongoTemplateRef() String
createIndexesForQueryMethods() boolean
considerNestedRepositories() boolean
Configuration
value() String
SimpleMongoConfiguration
databaseName String
mongo() MongoClient
mongoTemplate() MongoTemplate


Service
value() String
LogInService
authenticationManager AuthenticationManager
tokenProvider JwtProvider
userRepository UserRepository
LogInService(AuthenticationManager, JwtProvider, UserRepository)
login(UserLogInDto) ResponseEntity
refreshTokens(MyUserDetails) ResponseEntity
getTokens(MyUserDetails) ResponseEntity
updateJwtRefreshId(MyUserDetails) MyUserDetails

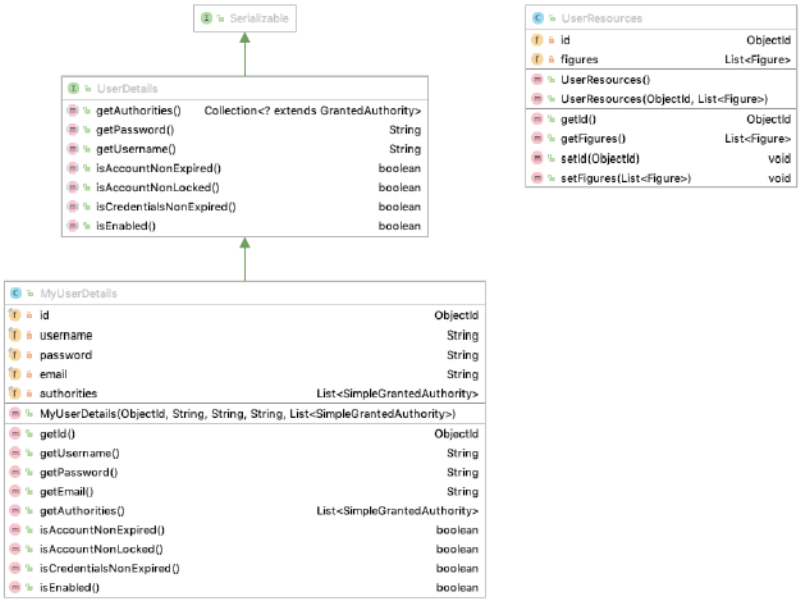

Component
value() String
JwtProvider
ISSUER String
GEOMETRIC_RESOURCES String
publicKey RSAPublicKey
JwtProvider(RSAPublicKey)
getUserIdFromJWT(String) ObjectId
validateToken(String) boolean
decodedJWT(String) DecodedJWT
getAlgorithm(PublicKey) Algorithm

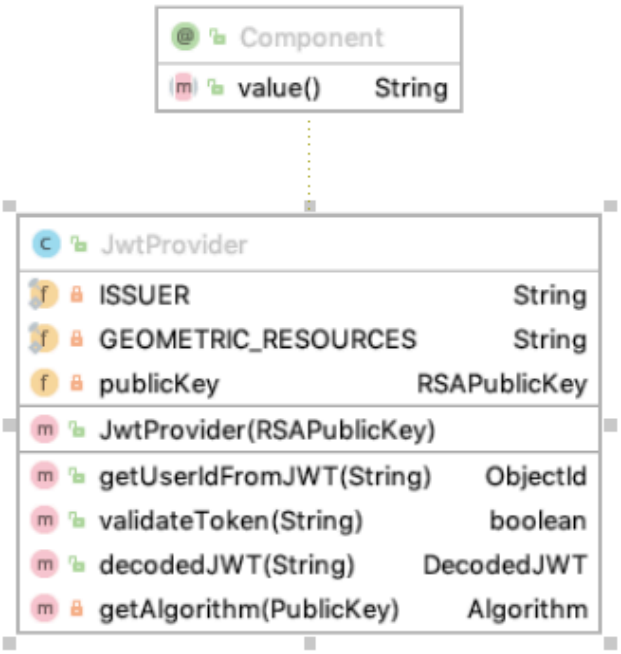

Configuration
value() String
JwtConfiguration
keyStoreSource String
keyStorePassword String
signingKeyAlias String
keyPair() KeyPair

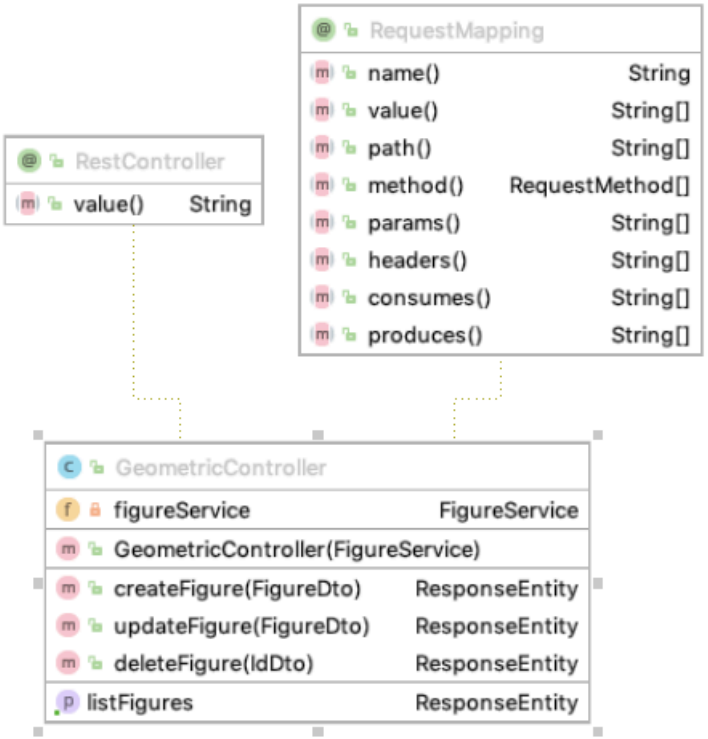

RequestMapping
name() String
value() String[]
path() String[]
method() RequestMethod[]
params() String[]
headers() String[]
consumes() String[]
produces() String[]
RestController
value() String
AuthController
logInService LogInService
signUpService SignUpService
AuthController(LogInService, SignUpService)
login(UserLogInDto) ResponseEntity
createUser(UserSignUpDto) ResponseEntity
refreshTokens() ResponseEntity
AdminController
adminService AdminService
AdminController(AdminService)
users() ResponseEntity
getProfile(String) ResponseEntity

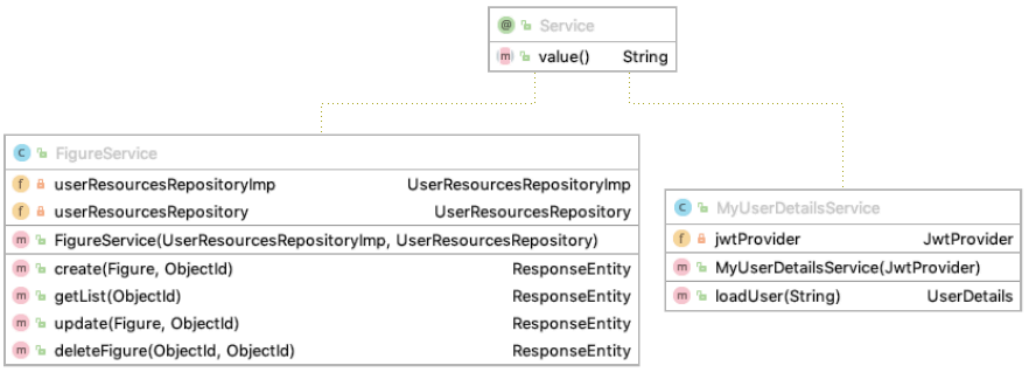

GenericFilterBean
OncePerRequestFilter
JwtAuthenticationFilter

8.2 DIAGRAMES UML de geomètric-resources





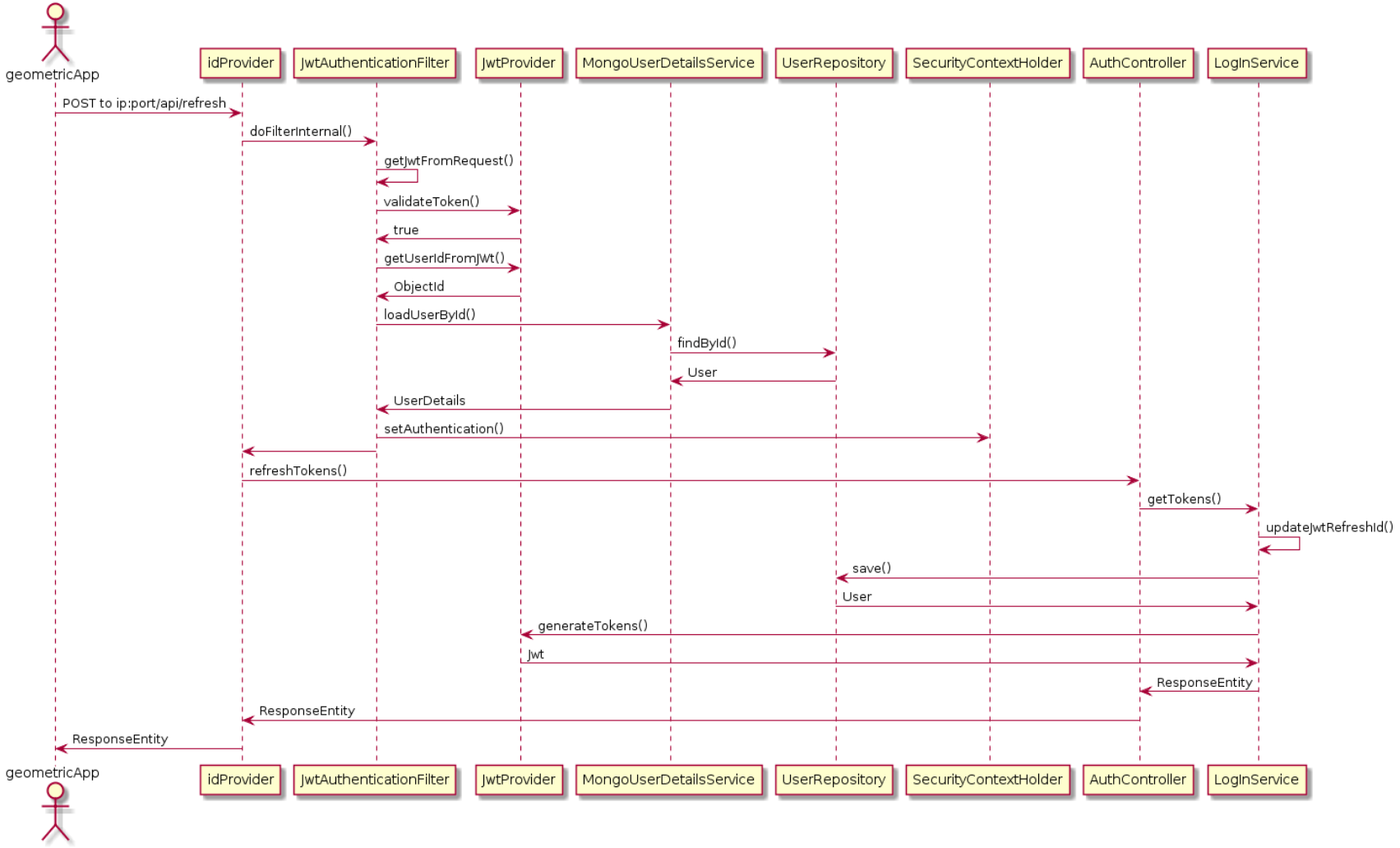

RequestMapping
name() String
value() String[]
path() String[]
method() RequestMethod[]
params() String[]
headers() String[]
consumes() String[]
produces() String[]
RestController
value() String
GeometricController
figureService FigureService
GeometricController(FigureService)
createFigure(FigureDto) ResponseEntity
updateFigure(FigureDto) ResponseEntity
deleteFigure(IdDto) ResponseEntity
listFigures ResponseEntity

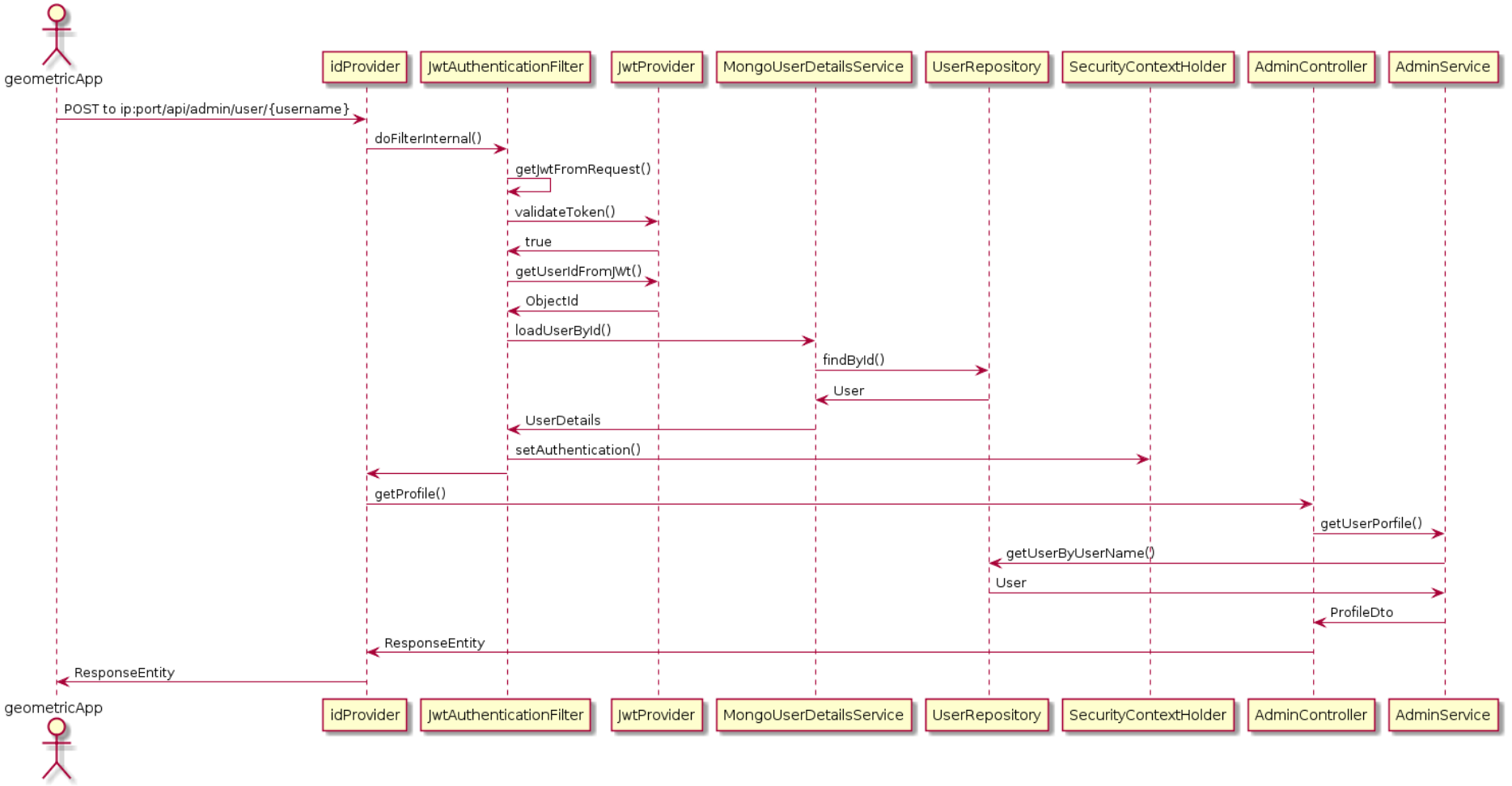

Service
value() String
FigureService
userResourcesRepositoryImp UserResourcesRepositoryImp
userResourcesRepository UserResourcesRepository
FigureService(UserResourcesRepositoryImp, UserResourcesRepository)
create(Figure, ObjectId) ResponseEntity
getList(ObjectId) ResponseEntity
update(Figure, ObjectId) ResponseEntity
deleteFigure(ObjectId, ObjectId) ResponseEntity
MyUserDetailsService
jwtProvider JwtProvider
MyUserDetailsService(JwtProvider)
loadUser(String) UserDetails

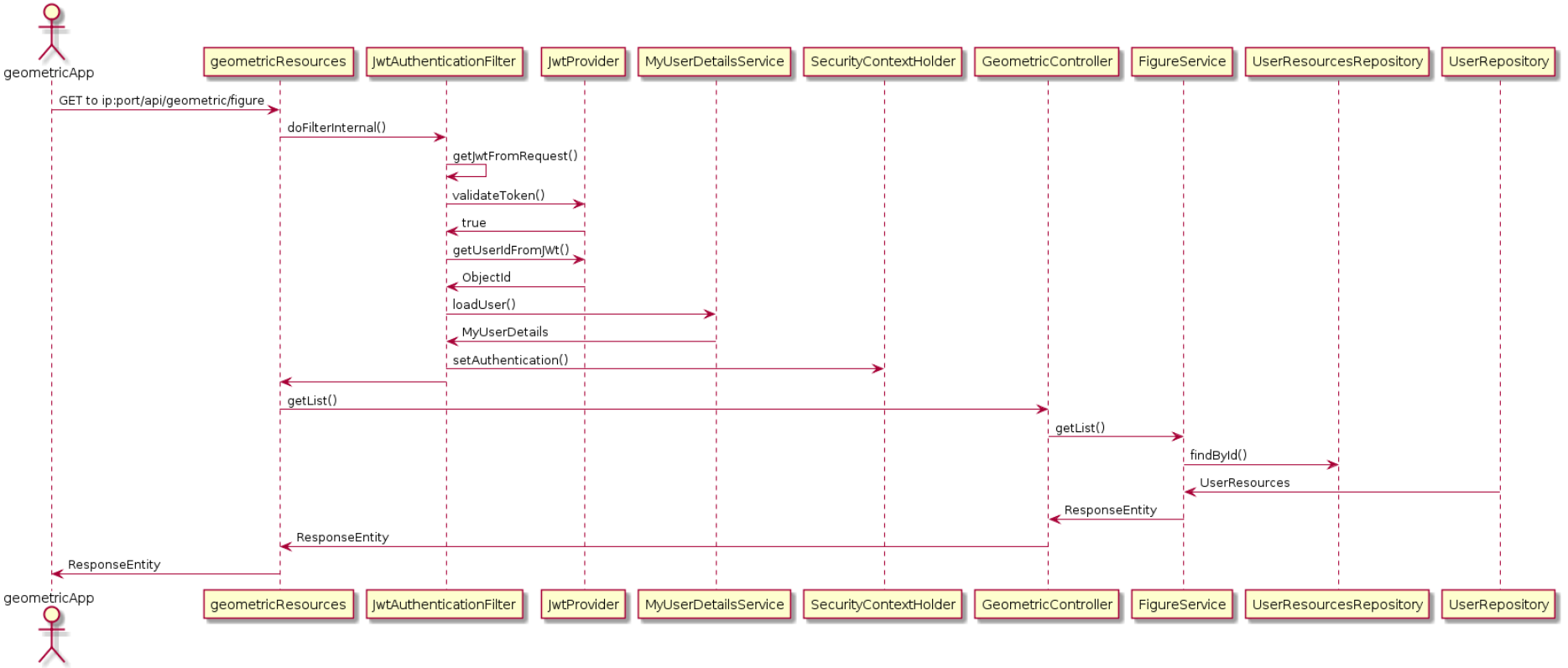

JsonTypeInfo
use() Id
include() As
property() String
defaultImpl() Class<?>
visible() boolean
JsonSubTypes
value() Type[]
Figure
id ObjectId
color String
Figure()
Figure(ObjectId, String)
getId() String
getColor() String
setId(ObjectId) void
setColor(String) void
Circle
type String
radius Integer
Circle()
Circle(ObjectId, String)
Circle(ObjectId, String, Integer)
getType() String
getRadius() Integer
setRadius(Integer) void
RegularPolygon
type String
sides Integer
radius Integer
RegularPolygon()
RegularPolygon(ObjectId, String)
RegularPolygon(ObjectId, String, Integer, Integer)
getType() String
getSides() Integer
getRadius() Integer
setSides(Integer) void
setRadius(Integer) void
Ellipse
type String
radiusX Integer
radiusY Integer
Ellipse()
Ellipse(ObjectId, String)
Ellipse(ObjectId, String, Integer, Integer)
getType() String
getRadiusX() Integer
getRadiusY() Integer
setRadiusX(Integer) void
setRadiusY(Integer) void


JsonDeserialize
using() Class<? extends JsonDeserializer>
contentUsing() Class<? extends JsonDeserializer>
keyUsing() Class<? extends KeyDeserializer>
builder() Class<?>
converter() Class<? extends Converter>
contentConverter() Class<? extends Converter>
as() Class<?>
keyAs() Class<?>
contentAs() Class<?>
IdDto
id ObjectId
IdDto(ObjectId)
getId() String
FigureDto
figure Figure
FigureDto(Figure)
getFigure() Figure
IdDtoBuilder
id ObjectId
IdDtoBuilder()
builder() IdDtoBuilder
withId (ObjectId) IdDtoBuilder
build() IdDto
FigureDtoBuilder
figure Figure
FigureDtoBuilder()
builder() FigureDtoBuilder
withFigure(Figure) FigureDtoBuilder
build() FigureDto

8.3 DIAGRAMES DE SEQUENCIA




geometricApp
idProvider
JwtAuthenticationFilter
JwtProvider
MongoUserDetailsService
UserRepository
SecurityContextHolder
AdminController
AdminService
POST to ip:port/api/admin/user/{username}
doFilterInternal()
getJwtFromRequest()
validateToken()
true
getUserIdFromJWt()
ObjectId
loadUserById()
findById()
User
UserDetails
setAuthentication()
getProfile()
getUserPorfile()
getUserByUserName()
User
ProfileDto
ResponseEntity
ResponseEntity
geometricApp
idProvider
JwtAuthenticationFilter
JwtProvider
MongoUserDetailsService
UserRepository
SecurityContextHolder
AdminController
AdminService


geometricApp
geometricResources
JwtAuthenticationFilter
JwtProvider
MyUserDetailsService
SecurityContextHolder
GeometricController
FigureService
UserResourcesRepository
UserRepository
GET to ip:port/api/geometric/figure
doFilterInternal()
getJwtFromRequest()
validateToken()
true
getUserIdFromJWt()
ObjectId
loadUser()
MyUserDetails
setAuthentication()
getList()
getList()
findById()
UserResources
ResponseEntity
ResponseEntity
ResponseEntity
geometricApp
geometricResources
JwtAuthenticationFilter
JwtProvider
MyUserDetailsService
SecurityContextHolder
GeometricController
FigureService
UserResourcesRepository
UserRepository

9. GLOSSARI

Front-end -> Sistema que actua amb el usuari i es el encarregat de mostra la presentació.

Back-end -> Sistema que gestiona una base de dades i respon a traves de un API REST.

Endpoint -> una URL on hi ha exposat un servei REST.

SPA -> Single Page Application.

SDK -> Software Development Kit.

IoC -> Acrònim de Inversió de Control: terme per referir-se a la injecció de dependències que utilitza el framework Spring.

UI -> User Interface: terme utilitzat per referir-se ala part visual amb la que interactua el usuari.

API -> Application Programming Interface: és un conjunt de subrutines, funcions i procediments que ofereix certa aplicació per ser utilitzat per un altre programari com una capa d'abstracció.

REST -> Representational State Transfer: protocol client/servidor sense estat, basat en peticions HTTP.

API RESTful -> Es una API que utilitza un protocol REST.

JSON -> JavaScript Object Notation: format estandarditzat per representar objectes en text pla sense comprovacions.

DTO -> Acrònim de Data Transfer Object.

POJO -> Plain Old Java Object

CRUD -> Create Read Update Delete