

# DOSSIER PROJET

*Nom*

► RISPAL

*Prénom*

► Gwenaëlle

*Adresse*

► 4 square Daumesnil  
94300 Vincennes

## Titre professionnel visé

Développeur·se web et web mobile – Niveau III

## Sommaire

<b>I. Introduction</b>	<b>p.</b>	<b>3</b>
<b>II. Compétences mises en œuvre</b>	<b>p.</b>	<b>4</b>
<b>III. A propos</b>	<b>p.</b>	<b>6</b>
▶ Français .....	p.	7
▶ Anglais .....	p.	8
<b>IV. Conception</b>		<b>9</b>
▶ Cahier des charges .....	p.	10
▶ User Stories et Use Case .....	p.	11
▶ Maquette .....	p.	13
▶ Diagrammes .....	p.	15
<b>V. Réalisation</b>	<b>p.</b>	<b>18</b>
▶ Outils utilisés .....	p.	19
▶ Back-end.....	p.	23
▶ Front-end .....	p.	38
▶ Déploiement.....	p.	52
<b>VI. Veille technologique</b>	<b>p.</b>	<b>54</b>
▶ Sécurité .....	p.	55
▶ Situation de travail ayant nécessité une recherche.....	p.	57
<b>VII. Conclusion</b>	<b>p.</b>	<b>61</b>
<b>VIII. Remerciements</b>	<b>p.</b>	<b>62</b>
<b>IX. Lexique</b>	<b>p.</b>	<b>63</b>

## INTRODUCTION

---

N'ayant pas suivi un cursus informatique dans mes études, j'ai découvert mon intérêt pour le développement totalement par hasard au sein de mon premier emploi. Mon travail consistait à effectuer des tâches répétitives sur des fichiers Excel, que j'ai finalement réussi à automatiser à l'aide de macros Excel. J'ai par la suite eu l'occasion d'assister à une formation Visual Basic for Application dans le cadre du DIF de mon entreprise.

Cette introduction au développement m'a fait me questionner sur mon orientation professionnelle, j'ai ainsi décidé de me reconvertir dans ce domaine. J'ai commencé par me former en autonomie sur des MOOC en ligne, mais ce que je cherchais, c'était surtout une formation qui soit diplômante.

Après plusieurs mois de recherches pour trouver une formation qui correspondait à mes moyens, je suis tombée sur l'école Simplon.co. Après avoir passé un entretien, j'ai été admise en tant qu'apprenante pour passer le Certificat de Qualification Professionnelle Développeur Nouvelles Technologies en alternance. Malheureusement cette formation a été annulée en raison du manque d'apprenants ayant trouvé une entreprise d'accueil. Simplon.co m'a ensuite redirigée vers la formation pour le Titre Professionnel Développeur web et web mobile. Cette formation débutant 10 mois plus tard, PayinTech, l'entreprise que j'avais trouvée pendant ma recherche d'alternance, m'a proposé un contrat en CDD en attendant le début de la formation. J'ai ainsi eu l'occasion d'avoir ma première expérience professionnelle dans le monde du développement informatique.

## COMPETENCES MISES EN ŒUVRE

### I. DEVELOPPER LA PARTIE FRONT-END D'UNE APPLICATION WEB OU WEB MOBILE EN INTEGRANT LES RECOMMANDATIONS DE SECURITE

#### 1. Maquetter une application

Le maquettage de l'application se fait généralement pendant la phase de conception de l'application à partir de cas d'utilisation. Elle peut se faire en plusieurs itérations que l'utilisateur final valide quand il estime qu'il y retrouve toutes les fonctionnalités nécessaires.

#### 2. Réaliser une interface utilisateur web statique et adaptable

L'interface utilisateur se réalise à partir de la maquette de l'application. Son rôle est de proposer à l'utilisateur une navigation simple et fluide pour réaliser ses tâches.

#### 3. Développer une interface utilisateur web dynamique

Contrairement à l'interface web statique, le contenu de l'interface dynamique change en fonction d'informations qui ne sont connues qu'au moment où l'utilisateur la consulte.

### II. DEVELOPPER LA PARTIE BACK-END D'UNE APPLICATION WEB OU WEB MOBILE EN INTEGRANT LES RECOMMANDATIONS DE SECURITE

#### 5. Créer une base de données

La base de données va permettre de stocker les informations en rapport avec l'application. On a besoin d'un système de gestion de base de données pour manipuler les données ainsi stockées.

#### 6. Développer les composants d'accès aux données

Des composants sont nécessaires afin que l'application puisse accéder aux informations de la base de données. On utilise un langage en back-end afin de faire communiquer l'application avec la base de données.

#### 7. Développer la partie back-end d'une application web ou web mobile

La partie back-end de l'application ayant la capacité de communiquer avec la base de données, c'est elle qui va fournir indirectement les informations aux services de la partie front-end de l'application.

## III. COMPETENCES TRANSVERSALES DE L'EMPLOI

Utiliser l'anglais dans son activité professionnelle en développement web et web mobile

Il est important de maîtriser l'anglais dans le développement web car cela permet de comprendre les documentations techniques qui sont rarement en français. Il est aussi important de développer en anglais pour pouvoir partager son code et qu'il soit compréhensible pour un maximum de personnes, quel que soit leur langue maternelle.

Actualiser et partager ses compétences en développement web et web mobile

L'informatique étant un domaine en constante évolution, le développeur doit être capable de se former par lui-même et de partager le résultat de ses recherches avec ses pairs.

# A PROPOS

Avant de parler de Twitch-Tags, mon application, je dois d'abord présenter Twitch. Il s'agit d'un site permettant de regarder ou de diffuser des vidéos live. C'est un site très similaire à Youtube, à la différence que l'accent est mis sur les diffusions en live et sur l'interaction entre les spectateurs et les diffuseurs. La thématique du contenu que l'on trouve sur Twitch est surtout le jeu-vidéo, car le site n'autorisait par le passé que ce genre de thème, mais depuis récemment, plus de sujets sont désormais autorisés. Twitch est donc composé de chaînes, et chaque chaîne possède une chat-room où les spectateurs peuvent discuter entre eux ou parler directement au diffuseur.

J'ai décidé de faire de mon application un catalogue de chaînes Twitch, sur lesquelles il est possible d'ajouter des tags afin de mieux définir ces chaînes, leur contenu, leur ambiance. On peut sélectionner des tags pour effectuer une recherche qui retournera ainsi un résultat spécifique aux attentes de l'utilisateur. S'il manque une chaîne à l'application, l'utilisateur peut la rajouter. Il est aussi possible de créer un compte pour avoir accès à des fonctionnalités supplémentaires. Un utilisateur enregistré peut donc rajouter un nouveau tag à une chaîne, ou approuver un tag d'un simple clic.

L'objectif de mon application est de permettre à ses utilisateurs de découvrir facilement de nouvelles chaînes correspondant à leurs goûts.

Pour la réaliser, j'ai choisi de faire une API REST pour le back-end, développée en Java avec Spring Boot. Le front-end est une application mono-page (Single Page Application) développée en Angular 6.

Before talking about Twitch-Tags, my application, I must introduce Twitch. It is a website for watching and streaming digital video broadcasts. It is very similar to Youtube, except the focus is on live broadcast and on the interaction between viewers and streamers. Twitch originally focused entirely on video games but has since expanded to include more themes for its content. Twitch is composed of channels, and each of these channels has a chat room where viewers can talk among themselves or speak directly to the streamer.

I have decided that my application would be a catalog of Twitch channels, on which it is possible to add tags to better characterize them, their content, and their atmosphere. It is possible to select tags to perform a search that will return a result custom-made to the expectations of the user. If a user cannot find the channel he was looking for, he can add it to the application. It is also possible to create an account, or also approve of a tag with a simple clic.

In conclusion, the goal of my application is to help users to easily discover new channels matching their needs.

To create it, I made a REST API for the back end, developped in Java with Spring Boot. The front-end is a Single Page Application developped in Angular 6.



# CONCEPTION

## CAHIER DES CHARGES

### CONTEXTE ET DEFINITION DU PROBLEME

L'idée pour mon application m'est venue d'une problématique personnelle. J'utilise Twitch autant en tant que spectatrice qu'en tant que diffuseuse et cela m'a permis de faire émerger des problèmes de chaque côté qui peuvent se résoudre avec une seule et même solution.

Quand je suis diffuseuse, ma préoccupation première est de réussir à attirer des spectateurs. Pour ça, tout ce que je peux faire, c'est lancer ma diffusion, et espérer qu'un spectateur clique sur la vignette de ma chaîne dans le répertoire de ma catégorie de jeu. Seulement ce répertoire est classé de façon que les chaînes populaires soient tout en haut de la page, et il n'y a aucun moyen d'inverser ce tri. Donc pour trouver ma chaîne, le spectateur devra passer plein d'autres chaînes avant de tomber sur la mienne. Sachant qu'il y a plusieurs centaines de chaînes dans cette catégorie et que les chaînes les plus populaires ont entre 2000 et 5000 spectateurs, le spectateur potentiel devra scroller pendant de longues secondes pour arriver jusqu'à la ligne où se situe ma chaîne. Tout cela n'est pas très pratique pour les diffuseurs qui débutent, puisqu'évidemment, les spectateurs choisissent généralement de regarder les premiers channels visibles en haut de la page.

Quand je suis spectatrice et que je veux découvrir une nouvelle chaîne sur Twitch, je suis obligée de choisir un jeu ou une catégorie en particulier pour pouvoir ensuite en afficher la liste des chaînes. Cette liste m'affiche uniquement le nom de la chaîne, le titre de la diffusion, une capture d'écran et le nombre de personne regardant cette diffusion. Et c'est tout. Est-ce que le diffuseur est très bon dans ce jeu ? Est-il drôle ? Est-ce qu'il a plutôt tendance à s'énervé ? Est-ce que cette chaîne est regardable par toute la famille ? Autant de questions auxquelles on ne peut répondre qu'en prenant du temps pour voir ce que propose le diffuseur. En tant que spectatrice, j'ai plusieurs fois passé une heure à cliquer sur des chaînes au hasard pour voir si elles me convenaient, pour finalement abandonner faute d'avoir trouvé ce que je cherchais.

### OBJECTIF DU PROJET

L'objectif de mon application est donc d'aider les spectateurs de Twitch à découvrir de nouvelles chaînes correspondant le mieux possible à leurs attentes grâce à un système de tags. Un diffuseur sur Twitch pourra par exemple ajouter sa chaîne à l'application et y rentrer les tags qu'il pense lui correspondent. Ainsi les utilisateurs venant sur l'application pour faire une recherche pourront sélectionner les tags qu'ils désirent et afficher toutes les chaînes correspondant à ces tags. S'il trouve qu'un tag convient particulièrement bien à une chaîne, ils pourront même cliquer dessus et ainsi incrémenter un compteur qui montrera aux autres utilisateurs que ce tag est tout à fait pertinent sur cette chaîne. De cette manière un diffuseur possédant une chaîne ayant peu de spectateurs pourra plus facilement être trouvé s'il s'adresse à une niche en particulier par exemple.

## USER STORIES ET USE CASE

### USER STORIES

Les user stories ou récit utilisateur sont de courtes et simples descriptions d'une fonctionnalité décrite par la personne à laquelle elle est destinée :

« En tant que *<rôle de l'utilisateur>*, je veux *<but de la fonctionnalité>*. »

J'ai choisi de débiter la phase de conception de mon application en écrivant des User Stories. Cela m'a permis de clarifier les rôles utilisateurs, de bien définir les fonctionnalités dont j'aurai besoin et d'aider à l'organisation et à la répartition du temps de travail.

Voici la liste des User Stories principales de mon projet :

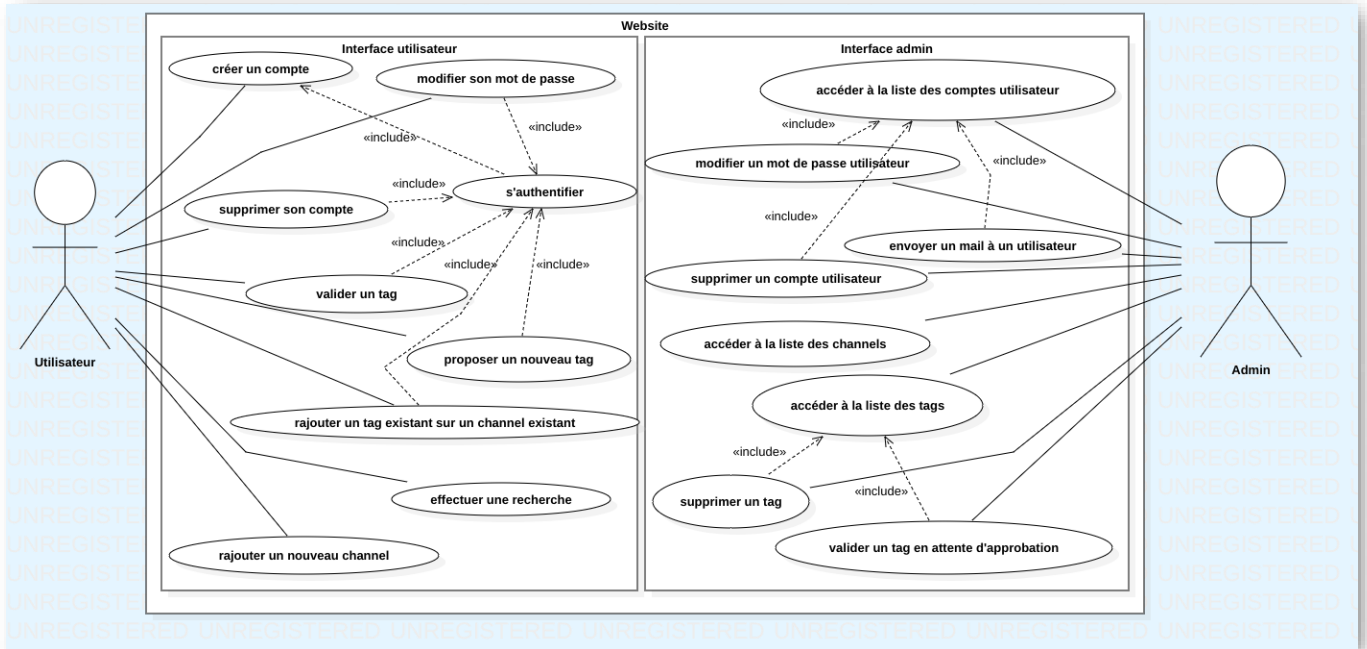
- > En tant qu'utilisateur, je veux pouvoir taper le nom d'une chaîne pour la rechercher.
- > En tant qu'utilisateur, je veux pouvoir sélectionner des tags dans une liste pour chercher des chaînes.
- > En tant qu'utilisateur, je veux pouvoir ajouter une nouvelle chaîne à l'application.
- > En tant qu'utilisateur, je veux pouvoir créer un compte pour accéder aux fonctionnalités supplémentaires.
- > En tant qu'utilisateur, je veux pouvoir accéder aux détails d'une chaîne.
- > En tant qu'utilisateur, je veux pouvoir ajouter un tag à une chaîne.
- > En tant qu'utilisateur, je veux pouvoir supprimer mon compte.
- > En tant qu'administrateur, je veux pouvoir supprimer un tag.
- > En tant qu'administrateur, je veux pouvoir modifier ou supprimer une chaîne.

### DIAGRAMME USE CASE

Le Use Case ou Cas d'utilisation est un diagramme qui permet d'identifier les interactions entre le système et ses utilisateurs. Le Use Case diffère des User Stories dans le sens où il permet de mieux identifier les liaisons et apporte une vision globale facilitée.

# DOSSIER PROJET

Voici mon diagramme Use Case :

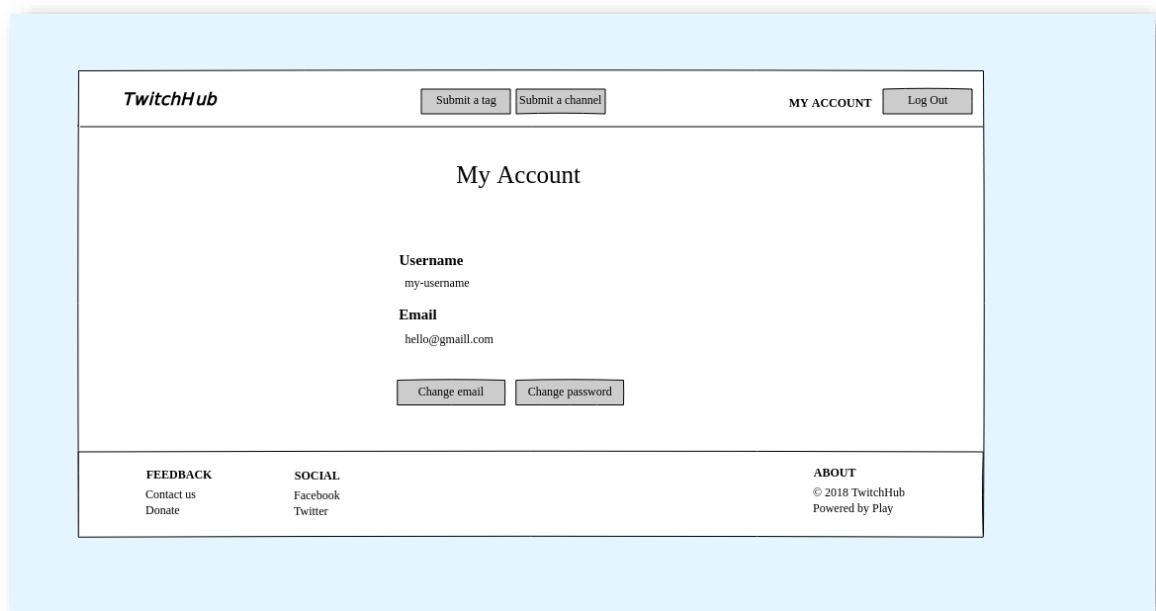


## MAQUETTE

Après avoir fait les User Story et le diagramme Use Case, j'ai décidé de passer à la création de la maquette. J'ai commencé par faire du **zoning** sur papier afin de schématiser grossièrement l'interface de mon application et d'en identifier les principales zones :



Je suis ensuite passé à la création du **wireframe**, aussi appelé maquette fil de fer, sur le logiciel **Pencil**. J'ai ainsi pu définir plus précisément l'organisation et la structure des éléments de mon application.



*Page de compte utilisateur*

# DOSSIER PROJET

**TwitchHub**

Submit a tagSubmit a channel

MY ACCOUNTLog Out

169 x 164

ChannelName<sup>FR</sup>

Followers : 166 786

<https://twitch.tv/channelname>

Partner : YES

TAGS

Click on a tag to endorse it!

positivity +10k

kid-friendly +5k

PUBG +2k

creative +500

[See more](#)

Want to add a tag to this channel ?

pop music

submit

[Don't find the tag you are looking for ? Submit a new tag here!](#)

Page de profil d'une chaine

**TwitchHub**

Submit a tagSubmit a channel

MY ACCOUNTLog Out

Tag Tag Tag

Search

RESULTS

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

84 x 77

Channel name

Tag Tag Tag

Page de résultat d'une recherche

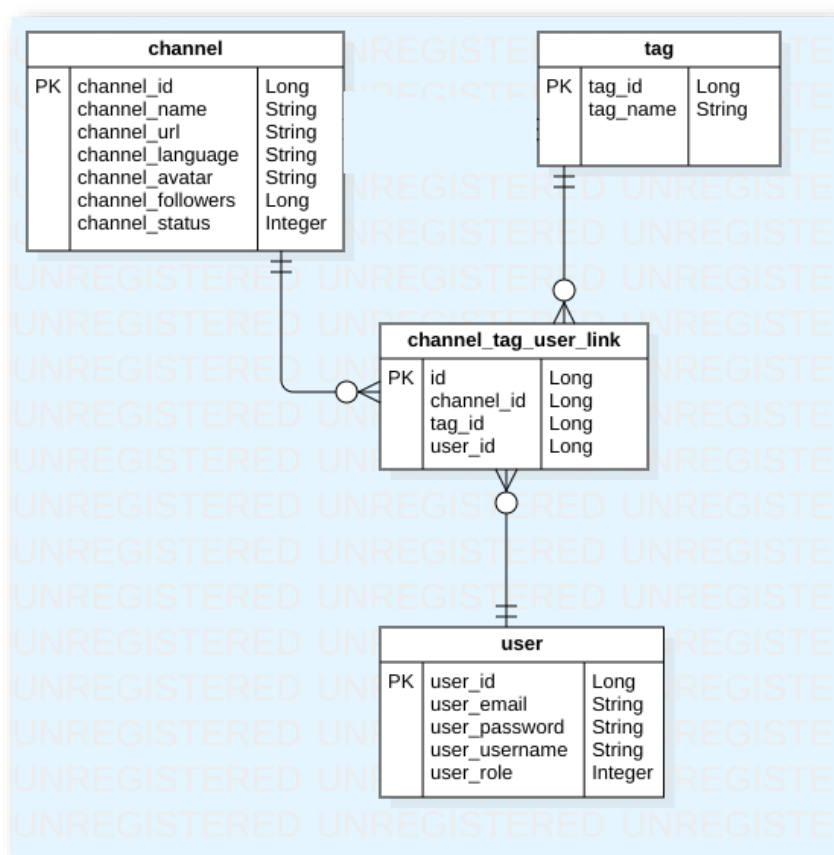
Page 14

DOSSIER PROJET-Gwenaëlle RISPAL

## DIAGRAMMES

### DIAGRAMME ENTITE-RELATIONS

J'ai me suis ensuite attelée au diagramme Entité-Relations, celui-ci montre les différentes entités de mon application et quel type de relation les lie entre elles. Ce diagramme est crucial pour bien organiser la base de données.

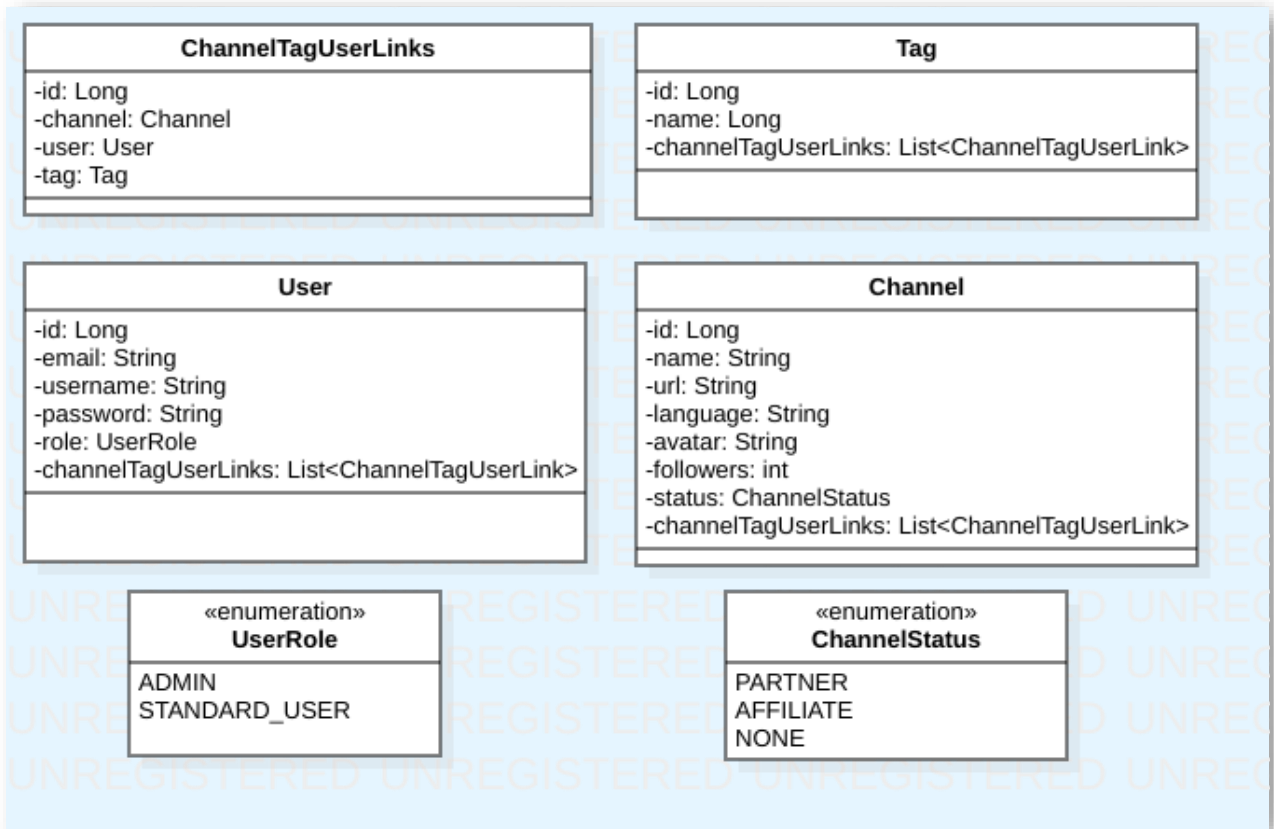


Nous avons donc les entités **channel**, **tag** et **user**, qui correspondent aux chaînes, tags et comptes utilisateurs de l'application. Concernant les relations, un channel peut avoir plusieurs tags et un tag peut être attribué à plusieurs channels, mais il s'agit de l'utilisateur qui attribue les tags, et un utilisateur ne peut pas attribuer plusieurs fois le même tag au même channel. J'ai donc choisi de matérialiser ces relations grâce à la table **channel\_tag\_user\_link**, ainsi un channel, un tag ou un user peuvent avoir plusieurs **channel\_tag\_user\_link**, mais un **channel\_tag\_user\_link** ne peut avoir qu'un seul channel, un seul tag et un seul user. **Channel\_tag\_user\_link** a donc une relation **OneToMany** avec chacune des autres tables.

# DOSSIER PROJET

## DIAGRAMME DE CLASSE UML

Enfin, la dernière étape dans la conception de mon application était la création du diagramme de classe UML. Il va décrire clairement la structure des éléments qui composeront mon back-end en modélisant ses classes, leurs attributs et opérations, et les relations les caractérisant.



On retrouve les quatre tables du diagramme ER, sous forme d'objet Java cette fois.

L'objet **Channel** possède les attributs suivants :

- **id** : il s'agit de l'identifiant de l'objet, c'est donc un Long
- **name** : le nom de la chaîne
- **url** : l'adresse à laquelle on peut retrouver la chaîne
- **language** : la langue principale utilisée sur la chaîne
- **avatar** : il s'agit de l'image principale utilisée par la chaîne
- **followers** : le nombre de personnes suivant la chaîne sur Twitch
- **status** : c'est le statut de la chaîne sur Twitch, ici j'ai choisi d'en faire une énumération puisqu'il n'y a que trois possibilités : **PARTNER**, **AFFILIATE** et **NONE** (ces statuts sont attribués par l'entreprise derrière Twitch, ils indiquent le niveau de monétisation de la chaîne)
- **channelTagUserLinks** : une liste des relations entre les tables

L'objet **User** possède les attributs suivants :

- **id** : il s'agit de l'identifiant de l'objet, c'est donc un Long



---

# DOSSIER PROJET

---

- **email**: l'adresse email de l'utilisateur
- **username** : le pseudo choisi par l'utilisateur
- **password** : le mot de passe de l'utilisateur
- **role** : une énumération indiquant les droits de l'utilisateur : **ADMIN** ou **STANDARD\_USER**
- **channelTagUserLinks** : une liste des relations entre les tables

L'objet **Tag** possède les attributs suivants :

- **id** : il s'agit de l'identifiant de l'objet, c'est donc un Long
- **name** : le nom du tag
- **channelTagUserLinks** : une liste des relations entre les tables

L'objet **ChannelTagUserLink** possède les attributs suivants :

- **id** : il s'agit de l'identifiant de l'objet, c'est donc un Long
- **channel** : la chaîne auquel le lien est lié
- **tag** : le tag auquel le lien est lié
- **user** : l'utilisateur auquel le lien est lié

# REALISATION

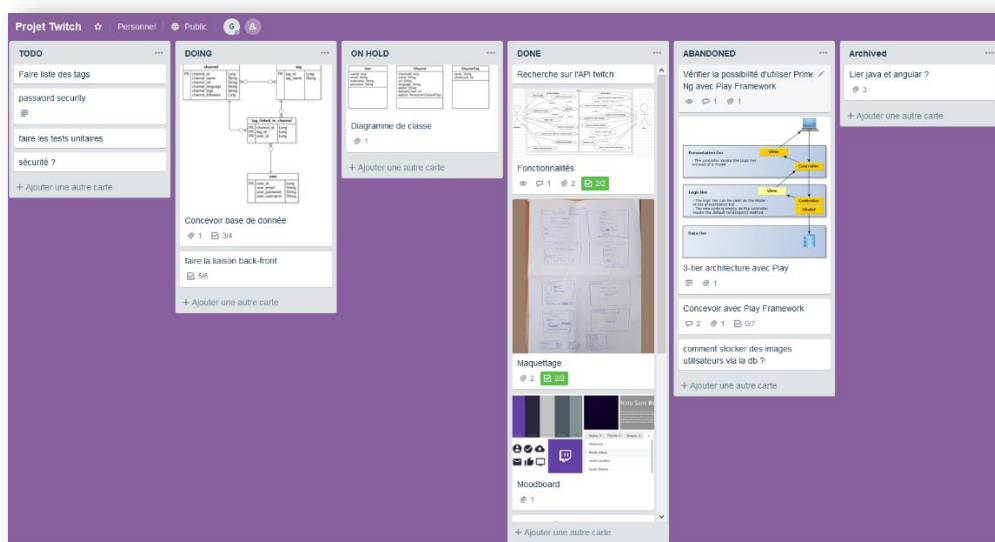
# DOSSIER PROJET

## OUTILS UTILISES

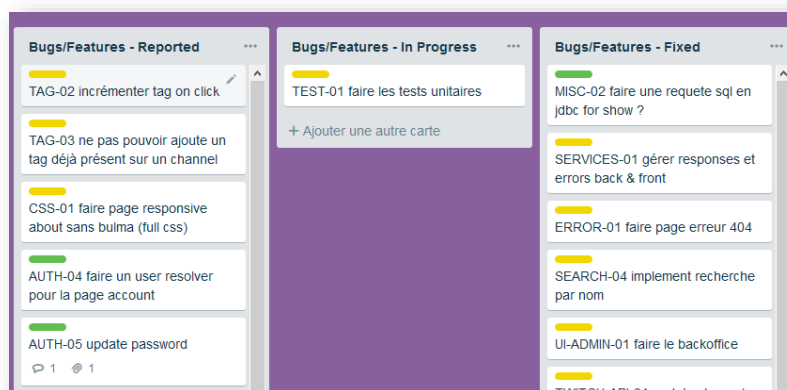
### ORGANISATION

Une fois la phase de conception finie, je me suis attelée à la réalisation de mon application. Pour cela un certain nombre d'outils m'ont été utiles.

Pour m'organiser dans la réalisation du travail et la gestion du temps, j'ai utilisé l'application **Trello**. J'ai ainsi listé grossièrement les tâches que j'avais à réaliser et le détail de leur réalisation.

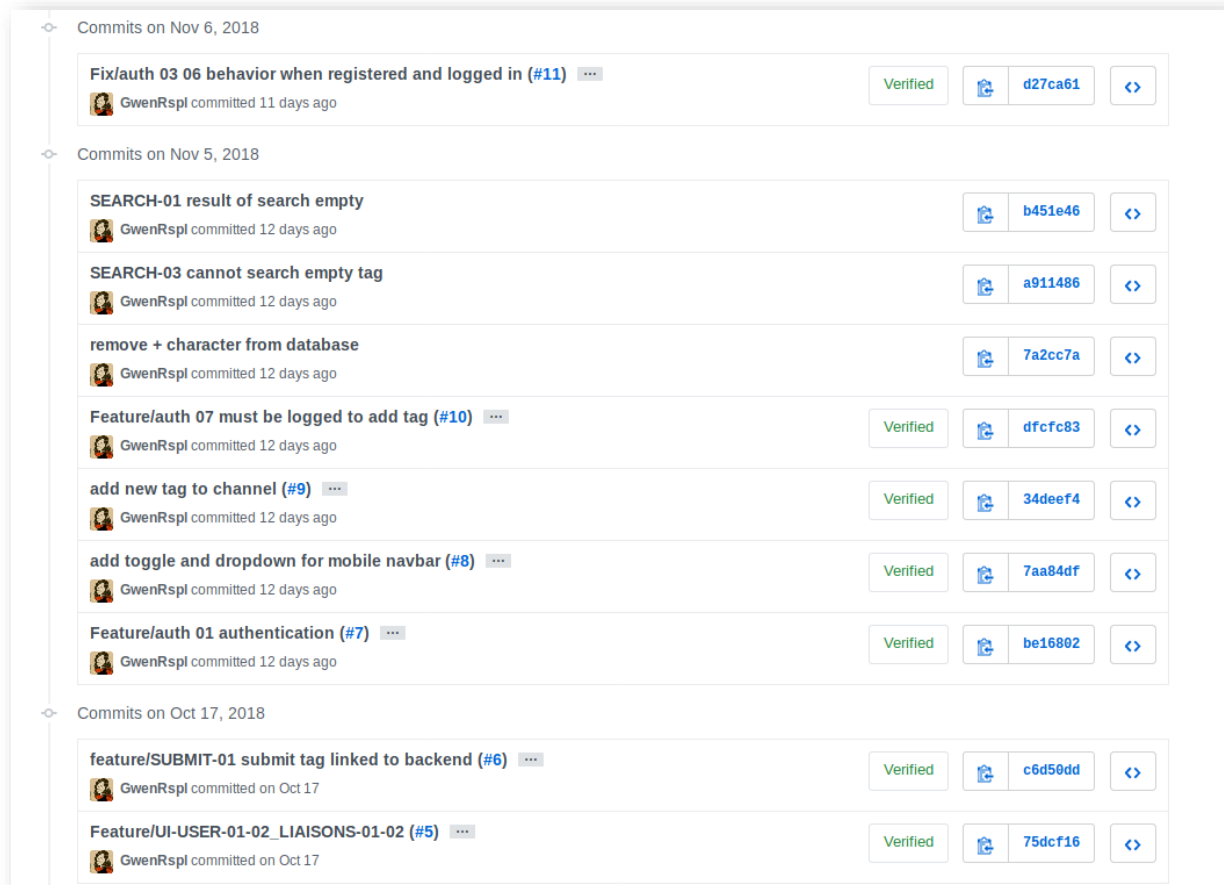


Une fois le développement de mon application bien commencé, j'ai un peu changé ma façon d'organiser mon Trello pour plutôt fonctionner avec un système de **tickets** comme je le faisais déjà dans mon entreprise. Les tickets ont chacun une tâche bien précise à réaliser et une priorité allant de "trivial" à "critique", il était ainsi plus facile de repérer en coup d'œil les tickets les plus urgents.



# DOSSIER PROJET

Pour les sauvegardes de mon travail, j'ai utilisé **Git** et **GitHub**. Quand je développais une nouvelle fonctionnalité, je le faisais sur une branche distincte de master, puis je faisais une pull-request que je mergeais une fois le développement de la fonctionnalité terminé. Ainsi la branche master était toujours protégée d'éventuelles erreurs ou problèmes de manipulation.



J'ai utilisé l'IDE **IntelliJ IDEA**. C'est le logiciel que j'utilise en entreprise, je le maîtrise donc bien et suis plus efficace dessus que sur Eclipse, son concurrent. Je m'en suis servi pour coder le back-end mais aussi le front-end puisque qu'il supporte plusieurs langages et différents frameworks.



# DOSSIER PROJET

## BACK-END

Concernant le système de gestion de base de données, j'ai utilisé **PostgreSQL** avec son outil d'administration graphique **pgAdmin**. Pour le développement de l'API REST, j'ai choisi d'utiliser **Java 8** dans un projet **SpringBoot**. SpringBoot est un micro framework qui a pour but de faciliter la configuration d'un projet Spring.

Pour gérer mon projet SpringBoot, j'ai utilisé **Maven**. C'est un outil de gestion de projets qui permet d'automatiser certaines tâches et de gérer les dépendances du projet. Concernant les dépendances, j'utilise l'ORM **Spring Data JPA**. Son rôle est de faire correspondre les objets de mon système aux tables correspondantes dans la base de données.

Enfin, j'ai utilisé le logiciel **Insomnia** pour tester les endpoints de mon API. Il s'agit d'une application permettant d'organiser, lancer et déboguer des requêtes HTTP.



## FRONT-END

Pour développer mon interface utilisateur, j'ai choisi d'utiliser **Angular 6**. Angular est un framework basé sur **TypeScript**, qui permet de construire facilement des applications web grâce à un système de conventions.

J'ai choisi d'utiliser Angular car TypeScript se base sur ES6, mon projet peut par conséquent bénéficier de toutes ses nouvelles fonctionnalités et le code est ainsi mieux structuré et plus lisible. Angular me permet aussi d'avoir une architecture **Single Page Application**, l'expérience utilisateur en est fluidifiée puisque cela évite le chargement d'une nouvelle page à chaque action demandée. Enfin Angular est régulièrement mis à jour et possède une grande communauté, il est donc très facile de trouver des informations ou des dépendances compatibles.

J'utilise aussi **NodeJS** et **NPM** pour faire fonctionner Angular. NodeJS permet de faire fonctionner le serveur dont aura besoin Angular, mais il est aussi nécessaire pour faire fonctionner **AngularCLI**, l'outil de ligne de commande d'Angular et NPM, qui lui s'occupe de la gestion des dépendances.

## DOSSIER PROJET

J'ai aussi choisi d'utiliser le framework CSS **Bulma**. Ce dernier est basé sur Flexbox, il est open-source, simple et surtout très lisible, contrairement à Bootstrap par exemple.

Enfin ayant fait la majeure partie de mon développement sous linux, j'ai surtout utilisé le navigateur **Firefox Developer Edition** mais j'ai aussi testé mon application sur les navigateurs **Chrome** et **Edge** les quelques fois où j'ai développé sous Windows.



## BACK-END

### API REST

Quand j'ai réfléchi à la façon dont j'allais faire le backend de mon application, mon choix s'est vite porté sur une API REST pour plusieurs raisons :

- Il s'agit d'une architecture standardisée et simple à mettre en œuvre, elle est aussi facilement compréhensible pour un développeur qui ne l'aurait pas développée lui-même.
- Elle supporte plusieurs formats de données différentes mais est surtout utilisé avec JSON ce qui se traduit par un meilleur support des navigateurs clients.
- REST utilise peu de bande passante

### BASE DE DONNEES

La première étape dans le démarrage de mon projet a été d'initialiser la base de données. J'ai donc créé une nouvelle base de données **twitch-tags-db** et son utilisateur.

Ensuite, j'ai écrit un script SQL pour créer mes tables en me référant à mon diagramme d'entité-relation. Ainsi j'ai créé les tables `channel`, `website_user`, `tag` et la table de jonction `channel_tag_user_link` avec les clés primaires et étrangères nécessaires.

```
DROP TABLE IF EXISTS channel CASCADE;
CREATE TABLE IF NOT EXISTS channel (
  id serial primary key,
  avatar character varying(255),
  status integer,
  followers bigint,
  language character varying(255),
  name character varying(255),
  url character varying(255)
);

DROP TABLE IF EXISTS tag CASCADE;
CREATE TABLE IF NOT EXISTS tag (
  id serial primary key,
  name character varying(255)
);

DROP TABLE IF EXISTS website_user CASCADE;
CREATE TABLE IF NOT EXISTS website_user (
  id serial primary key,
```

# DOSSIER PROJET

```
role integer,  
email character varying(255),  
password character varying(255),  
username character varying(255)  
);  
  
DROP TABLE IF EXISTS channel_tag_user_link;  
CREATE TABLE IF NOT EXISTS channel_tag_user_link (  
    id serial primary key,  
    channel_id bigint REFERENCES channel(id) ON DELETE CASCADE,  
    tag_id bigint REFERENCES tag(id) ON DELETE CASCADE,  
    user_id bigint REFERENCES website_user(id) ON DELETE CASCADE  
);
```

Puis j'ai écrit le script SQL chargé d'insérer des données dans mes tables :

```
INSERT INTO channel(status, avatar, followers, language, name, url)  
VALUES (1, 'https://static-cdn.jtvnw.net/jtv_user_pictures/e1d1c7bf-7e51-  
4e55-ba08-98451a3f9fc2-profile_image-300x300.png', 327, 'fr', 'Naevyah',  
'https://twitch.tv/naevyah'),  
      (2, 'https://static-cdn.jtvnw.net/jtv_user_pictures/redfanny_-  
profile_image-25848107b59dae96-300x300.jpeg', 4432, 'fr', 'RedFanny_',  
'https://twitch.tv/redfanny_'),  
      (0, 'https://static-cdn.jtvnw.net/jtv_user_pictures/zerator-  
profile_image-48eee9de24a47e53-300x300.png', 493724, 'fr', 'ZeratoR',  
'https://www.twitch.tv/zerator'),  
      (0, 'https://static-cdn.jtvnw.net/jtv_user_pictures/83f7ac2b8a6813e6-  
profile_image-300x300.png', 12116, 'FR', 'Nat_Ali',  
'https://www.twitch.tv/nat_ali');  
  
INSERT INTO website_user(role, email, password, username)  
VALUES (1, 'user1@user.fr', 'user1', 'user1'),  
      (1, 'user2@user.fr', 'user2', 'user2'),  
      (1, 'user5@user.fr', 'user5', 'user5');  
  
INSERT INTO tag(name)  
VALUES ('humor'),  
      ('positivity'),  
      ('MMORPG'),  
      ('kid-friendly'),  
      ('FPS'),  
      ('chill');  
  
INSERT INTO channel_tag_user_link(channel_id, tag_id, user_id)  
VALUES (1, 1, 1),  
      (2, 1, 2),  
      (3, 1, 2),  
      (3, 1, 3),  
      (3, 1, 4);
```



## SPRINGBOOT

L'étape suivante était d'initialiser mon projet SpringBoot avec ses dépendances. Pour ça j'utilisé **SpringCLI** avec la ligne de commande suivante :

```
λ spring init --dependencies=Web,JPA,PostgreSQL,Security,Lombok, DevTools twitch-tags
```

Par défaut, SpringBoot va chercher des entités dans les packages pour créer leurs tables respectives en base de données. Mais étant donné que je désirais les créer moi-même pour avoir un maximum de contrôle dessus, j'ai mis les scripts SQL précédemment écrits dans le dossier **resources** situé dans **src/main**. Dans ce même dossier, il y a un fichier **application.properties** qui sert à définir certains paramètres, dont ceux pour que SpringBoot puisse se connecter à la base de données. J'ai aussi rajouté deux paramètres pour que JPA affiche les logs de SQL et pour qu'il ne génère pas les tables lui-même. On trouve aussi dans ce fichier la clé secrète nécessaire à mon système d'authentification.

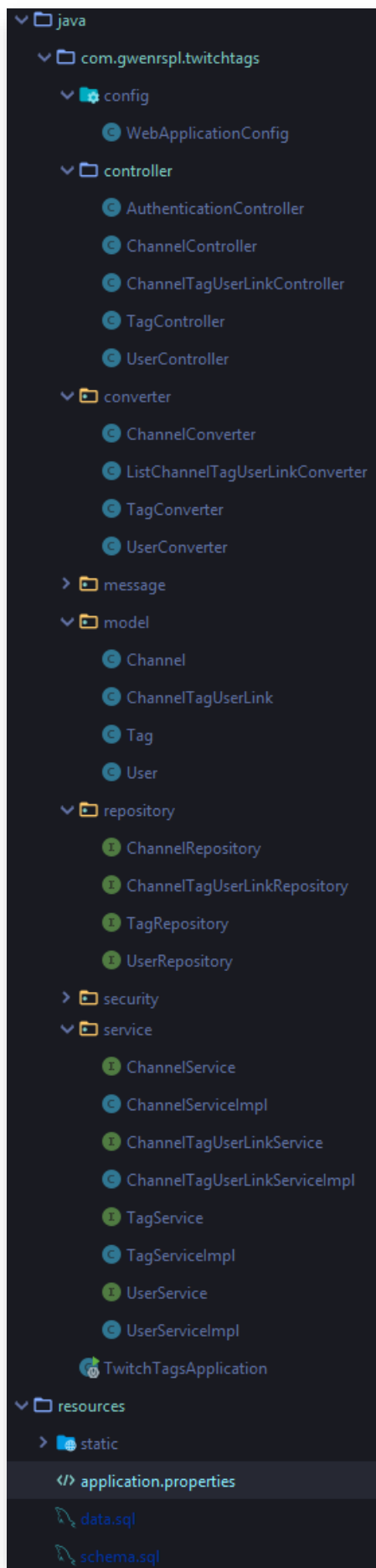
```
#Postgresql settings
spring.datasource.url=jdbc:postgresql://localhost:5432/twitch-tags-db
spring.datasource.username=gwenrspl
spring.datasource.password=toor
spring.datasource.initialization-mode=always

#JPA settings
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=none

gwenrspl.app.jwtSecret=rsplSecretKey
gwenrspl.app.jwtExpiration=86400
```

Ensuite en m'aidant de mon diagramme de classe, j'ai créé les classes et les packages donc j'avais besoin. Je me suis donc retrouvée avec cette structure finale :

# DOSSIER PROJET



# DOSSIER PROJET

J'ai commencé par créer mes entités dans le package model : **Channel**, **Tag**, **User** et l'entité de liaison **ChannelTagUserLink**.

Regardons de plus près la classe Channel :

```
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class Channel {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
    private String url;
    private String language;
    private Long followers;
    private String avatar;
    private ChannelStatus status;
    @OneToMany(mappedBy = "channel", cascade = CascadeType.ALL, fetch = FetchType.LAZY)
    @JsonSerialize(converter = ListChannelTagUserLinkConverter.class)
    private List<ChannelTagUserLink> channelTagUserLinks;

    public enum ChannelStatus {
        PARTNER(0),
        AFFILIATE(1),
        NONE(2);

        private int statusCode;

        ChannelStatus(int statusCode) {
            this.statusCode = statusCode;
        }

        public int getStatusCode() {
            return statusCode;
        }

        public void setStatusCode(int statusCode) {
            this.statusCode = statusCode;
        }
    }
}
```

J'ai ajouté l'annotation **@Entity** pour que JPA comprenne qu'il s'agit d'une entité. **@Data** et **@AllArgsConstructor** sont des annotations pour **Lombok** : **@Data** permet de générer tout le code standard qui accompagne généralement un objet Java : les **getters** et **setters** et les méthodes **toString**, **equals** et **hashCode**. Quant à **@AllArgsConstructor**, elle permet de générer un constructeur avec tous les

# DOSSIER PROJET

arguments correspondant aux attributs de la classe. Grâce à ces deux annotations, le code de ma classe est beaucoup plus concis.

J'ai ensuite défini les attributs de ma classe. Mon attribut **id** est annoté de **@Id** pour indiquer à JPA qu'il s'agit de l'attribut correspondant à la clé primaire de la table. **@GeneratedValue** indique que la clé primaire doit être incrémentée automatiquement et son paramètre **strategy** indique que la génération se fera à partir d'une identité propre au SGBD (PostgreSQL dans notre cas). L'attribut **status** étant de type Enum, j'ai créé **ChannelStatus** à l'intérieur de la classe Channel. J'ai annoté l'attribut **channelTagUserLinks** avec **@OneToMany** pour signaler à JPA qu'il existe une relation de cardinalité 0,n. En effet, un channel peut avoir plusieurs channelTagUserLinks, mais un channelTagUserLinks ne peut avoir qu'un seul channel.

Enfin pour éviter d'avoir une boucle infinie lors de l'affichage du JSON, j'ai ajouté l'annotation **@JsonSerialize** avec son paramètre **converter** pour indiquer que cet attribut doit être sérialisé à l'aide de la classe **ListChannelTagUserLinkConverter**.

```
public class ListChannelTagUserLinkConverter extends
StdConverter<List<ChannelTagUserLink>,
List<ListChannelTagUserLinkConverter.LinkSummary>> {

    @Override
    public List<ListChannelTagUserLinkConverter.LinkSummary> convert(final
List<ChannelTagUserLink> channelTagUserLinks) {
        final List<LinkSummary> summaryList = new ArrayList<>();
        for (final ChannelTagUserLink link : channelTagUserLinks) {
            final LinkSummary summary = new LinkSummary();
            summary.setId(link.getId());
            summary.setChannelName(link.getChannel().getName());
            summary.setTagName(link.getTag().getName());
            summary.setUsername(link.getUser().getUsername());
            summaryList.add(summary);
        }
        return summaryList;
    }

    public class LinkSummary {
        private Long id;
        private String channelName;
        private String tagName;
        private String username;

        public LinkSummary() {
        }
    }
}
```

Si on regarde la classe ListChannelTagUserLinkConverter, on peut voir qu'elle étend la classe **StdConverter** et qu'elle surcharge la méthode **convert()**. J'ai créé une classe interne **LinkSummary** qui représente un "résumé" de la classe ChannelTagUserLink, c'est elle qui sera sérialisée. Ses attributs sont

# DOSSIER PROJET

**id**, **channelName**, **tagName** et **username**, ce sont les seuls attributs qui m'intéressent quand je sérialise la classe Channel.

La méthode **convert()** prend donc en paramètre notre liste de ChannelTagUserLink et renvoie une liste de LinkSummary à la place. C'est à l'intérieur de cette méthode que la conversion a lieu : j'initialise ma liste de LinkSummary et je vais la remplir avec les données de la liste de ChannelTagUserLink.

Une fois les entités et les converters faits, j'ai créé les interfaces des **repository** pour chaque entité. Chaque interface étend **JpaRepository**, il s'agit d'une interface qu'offre **Spring Data Jpa**. **JpaRepository** étend **PagingAndSortingRepository**, qui étend **CrudRepository** qui elle-même étend l'interface **Repository**. Pour faire simple, **Repository** sert simplement à dire qu'une interface est un repository (qui vient du patron de conception repository, servant à abstraire l'accès aux données), le **CrudRepository** ajoute des méthodes **CRUD** pour la base de données, **PagingAndSortingRepository** rajoute un ensemble de méthodes pour la pagination et le tri. Enfin **JpaRepository** rajoute des méthodes comme la suppression en lots ou le vidage du cache.

```
public interface ChannelRepository extends JpaRepository<Channel, Long> {}
```

*Le repository de la classe Channel*

Ensuite, j'ai créé les **services** qui seront chargés de faire la communication entre le **repository** et le **controller**. J'ai commencé par créer une interface qui liste les méthodes disponibles :

```
public interface ChannelService {  
    List<Channel> listAll();  
    List<Channel> searchByName(String name);  
    List<Channel> searchByTag(String tag);  
    List<Channel> searchById(List<Long> id);  
    List<Channel> searchByTags(List<String> tags);  
    Channel getOne(Long id);  
    Channel create(Channel channel);  
    Channel update(Long id, Channel channel);  
    Boolean isPresent(String name);  
    void delete(Long id);  
}
```

# DOSSIER PROJET

Puis j'ai créé l'implémentation du service :

```
@Service
public class ChannelServiceImpl implements ChannelService {

    private final ChannelRepository repository;
    private ChannelTagUserLinkService linkService;
    private TagService tagService;

    public ChannelServiceImpl(ChannelRepository repository,
ChannelTagUserLinkService linkService, TagService tagService) {
        this.repository = repository;
        this.linkService = linkService;
        this.tagService = tagService;
    }

    @Override
    public List<Channel> listAll() {
        return this.repository.findAll();
    }

    @Override
    public List<Channel> searchByName(String name) {
        return this.repository.findAll().stream()
            .filter(channel ->
channel.getName().toUpperCase().contains(name.toUpperCase()))
            .collect(Collectors.toList());
    }

    @Override
    public Channel getOne(final Long id) {
        Optional<Channel> optChannel = this.repository.findById(id);
        return optChannel.orElse(null);
    }

    @Override
    public Channel create(final Channel channel) {
        return this.repository.save(channel);
    }

    @Override
    public Boolean isPresent(String name) {
        Optional<Channel> channelToFind = this.repository.findAll().stream()
            .filter(channel ->
channel.getName().toLowerCase().equals(name.toLowerCase()))
            .findAny();
        return channelToFind.isPresent();
    }

    @Override
    public void delete(final Long id) {
        this.repository.deleteById(id);
    }
}
```

# DOSSIER PROJET

L'annotation **@Service** permet de dire à Spring que cette classe est un service. C'est dans cette classe que je défini les méthodes de l'interface, elles sont en charge de faire les opérations nécessaires à la logique métier.

Enfin, j'ai créé les **controllers** de chaque entité. Voici le code du controller de Channel :

```
@RestController
@RequestMapping("/api/channels")
@CrossOrigin(origins= "http://localhost:4200")
public class ChannelController {

    private final ChannelService service;

    public ChannelController(final ChannelService channelService) {
        this.service = channelService;
    }

    @GetMapping(value = {"", "/"})
    public List<Channel> listAll() {
        return this.service.listAll();
    }

    @GetMapping("/{id}")
    public Channel getOne(@PathVariable final Long id) {
        return this.service.getOne(id);
    }

    @PostMapping("/create")
    public Channel create(@RequestBody final Channel channel) {
        return this.service.create(channel);
    }

    @GetMapping("/search")
    public List<Channel> searchByTags(@RequestParam("tag") List<String> tags) {
        return this.service.searchByTags(tags);
    }

    @PutMapping("/{id}")
    public ResponseEntity<?> update(@PathVariable final Long id, @RequestBody final
Channel channel) {
        this.service.update(id, channel);
        return new ResponseEntity<>(new ResponseMessage("Channel updated
successfully"), HttpStatus.OK);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<?> delete(@PathVariable final Long id) {
        this.service.delete(id);
        return new ResponseEntity<>(new ResponseMessage("Channel deleted
successfully"), HttpStatus.OK);
    }
}
```

L'annotation **@RestController** indique à Spring que la classe est un controller, **@RequestMapping** sert à indiquer l'url où seront accessibles par défaut les requêtes du controller, enfin **@CrossOrigin** permet d'autoriser les requêtes cross-origin provenant de l'url spécifiée. Cette dernière annotation est

# DOSSIER PROJET

nécessaire car les requêtes AJAX inter-domaine sont interdites par sécurité. La **W3C** a ainsi créé **CORS** qui offre donc la possibilité aux servers de pouvoir mieux contrôler leurs requêtes inter-domaine en ajoutant un header HTTP à leurs réponses qui permet au client de savoir quels sont les origines autorisées.

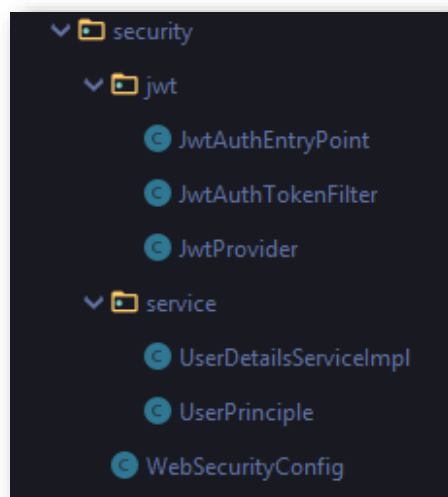
Chaque méthode du **controller** possède une annotation indiquant le type de requête qu'elle accepte et sur quel endpoint. En paramètre, j'ai défini si la méthode attendait un **@RequestBody**, **@RequestParam** ou encore un **@PathVariable**.

La méthode **create()** par exemple, n'accepte que les requêtes **POST** à l'adresse /api/channels/create et doit recevoir un objet de type Channel dans le corps de la requête, sous forme de **JSON** par exemple.

## SECURITE

Concernant la sécurité de mon application, la plupart des **endpoints** (points de terminaison) de mon API sont public puisqu'ils doivent pouvoir être accédés par n'importe quel visiteur du site, en revanche certains sont sécurisés car il faut que l'utilisateur soit enregistré pour y avoir accès, d'autres encore ne sont aussi accessibles que par les utilisateurs ayant le statut d'administrateur.

Pour implémenter la sécurité j'ai utilisé **Spring Security** avec un système d'authentification à l'aide de jetons **JWT**. J'ai créé les classes suivantes pour implémenter le système :



Spring Security va intercepter les requêtes HTTP, et les filtrer grâce à la classe **JwtAuthTokenFilter**. La classe **JwtProvider** est chargée de générer ou valider le token reçu. **JwtAuthEntryPoint** va s'occuper des erreurs d'authentification.

La classe **UserPrinciple** implémente **UserDetails**, c'est elle qui contient les informations nécessaires pour construire l'authentification.



# DOSSIER PROJET

La classe **UserDetailsServiceImpl** est là pour aider à créer un **UserPrinciple** à partir d'un string représentant le pseudonyme de l'utilisateur.

Enfin, la classe **WebSecurityConfig** sert de configuration pour Spring Security, c'est là qu'on lui indique quels sont les endpoints que l'on souhaite sécuriser ou non.

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Autowired
    private JwtAuthEntryPoint unauthorizedHandler;

    @Bean
    public JwtAuthTokenFilter authTokenFilter() {
        return new JwtAuthTokenFilter();
    }

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

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

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

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors().and().csrf().disable()
            .authorizeRequests()
                .antMatchers("/api/users/usernames",
                    "**").permitAll()
                .antMatchers("/api/users/**").denyAll()
                .anyRequest().authenticated()
                .and().exceptionHandling().authenticationEntryPoint(unauthorizedHandler)

        .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.addFilterBefore(authTokenFilter(), UsernamePasswordAuthenticationFilter.class);
    }
}
```

Ici on voit bien que la seule url sécurisée est /api/users/\*\*, sauf l'url spécifique /api/users.usernames.

# DOSSIER PROJET

Enfin j'ai créé un **controller** pour que les utilisateurs puissent s'enregistrer et s'identifier :

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

    private AuthenticationManager authenticationManager;
    private UserService userService;
    private PasswordEncoder encoder;
    private JwtProvider jwtProvider;

    public AuthenticationController(final UserService service, AuthenticationManager
authenticationManager, PasswordEncoder encoder, JwtProvider jwtProvider) {
        this.userService = service;
        this.authenticationManager = authenticationManager;
        this.encoder = encoder;
        this.jwtProvider = jwtProvider;
    }

    @PostMapping("/signin")
    public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginForm loginRequest) {
        Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginRequest.getUsername(),
loginRequest.getPassword()));
        SecurityContextHolder.getContext().setAuthentication(authentication);
        String jwt = jwtProvider.generateJwtToken(authentication);
        UserDetails userDetails = (UserDetails) authentication.getPrincipal();
        return ResponseEntity.ok(new JwtResponse(jwt, userDetails.getUsername(),
userDetails.getAuthorities()));
    }

    @PostMapping("/signup")
    public ResponseEntity<?> registerUser(@Valid @RequestBody SignupForm signupRequest) {
        if(userService.existsByUsername(signupRequest.getUsername())) {
            return new ResponseEntity<>(new ResponseMessage("Fail -> Username is already
taken"), HttpStatus.BAD_REQUEST);
        }

        if(userService.existsByEmail(signupRequest.getEmail())) {
            return new ResponseEntity<>(new ResponseMessage("Fail -> Email is already taken"),
HttpStatus.BAD_REQUEST);
        }

        User user = new User(signupRequest.getUsername(),signupRequest.getEmail(),
encoder.encode(signupRequest.getPassword()));
        String strRole = signupRequest.getRole();
        User.UserRole role;
        switch (strRole) {
            case "admin":
                role = User.UserRole.ADMIN;
                break;
            default:
                role = User.UserRole.STANDARD_USER;
        }
        user.setRole(role);
        userService.create(user);
        return new ResponseEntity<>(new ResponseMessage("User registered successfully"),
HttpStatus.OK);
    }
}
```

# DOSSIER PROJET

Quand l'utilisateur s'enregistre, on vérifie que le username qu'il a choisi n'est pas déjà pris, et si son adresse email n'existe pas déjà dans la base de données. Si tout est bon, on encrypte son mot de passe, on lui assigne un rôle et ses informations sont enregistrées dans la base de données.

C'est au moment où l'utilisateur s'identifie que le token JWT est généré et renvoyé dans la réponse.

## TESTS

### Tests unitaires

Après avoir créé mes classes, je me suis attaquée aux tests unitaires. Ces derniers sont destinés à tester une classe, les vérifications sont faites en exécutant une petite partie de code, pour vérifier que le résultat du test est bien le résultat attendu.

Pour effectuer mes tests, j'ai utilisé les frameworks **JUnit** et **Mockito**. **JUnit** permet de faire des tests unitaires spécialement conçu pour le langage de programmation Java, et **Mockito** permet de créer et de configurer des doubles d'objet (mock).

Prenons en exemple, la classe **TagControllerTest**, il s'agit de la classe de test de **TagController** :

```
@SpringBootTest
@RunWith(SpringRunner.class)
public class TagControllerTest {

    MockMvc mockMvc;

    @Autowired
    protected WebApplicationContext context;

    @Autowired
    TagController tagController;

    @MockBean
    TagService tagService;

    private List<Tag> tags;

    @Before
    public void setUp() throws Exception {
        this.mockMvc = MockMvcBuilders.webAppContextSetup(context).build();
        Tag tag1 = Tag.builder()
            .name("humor")
            .build();
        Tag tag2 = Tag.builder()
            .name("positivity")
            .build();

        tags = new ArrayList<>();
        tags.add(tag1);
        tags.add(tag2);
    }
}
```

# DOSSIER PROJET

```
@Test
public void listAll() throws Exception {
    when(tagService.listAll()).thenReturn(tags);

    mockMvc.perform(get("/api/tags").contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.name", is("humor")))
        .andExpect(jsonPath("$.name", is("positivity")));
}

@Test
public void getOne() throws Exception {
    when(tagService.getOne(1L)).thenReturn(tags.get(0));

    mockMvc.perform(get("/api/tags/1").contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("name", is("humor")));
}

@Test
public void create() throws Exception{
    Tag tag3 = Tag.builder()
        .name("chill")
        .build();
    when(tagService.create(tag3)).thenReturn(null);

    mockMvc.perform(post("/api/tags/create").contentType(MediaType.APPLICATION_JSON).content(
        "{\"name\": \"chill\"}"))
        .andExpect(status().isOk());
}

@Test
public void update() throws Exception{
    Tag tag3 = Tag.builder()
        .name("chill")
        .build();
    when(tagService.update(1L, tag3)).thenReturn(null);

    mockMvc.perform(put("/api/tags/1").contentType(MediaType.APPLICATION_JSON).content("{\"name\": \"chill\"}"))
        .andExpect(status().isOk());
}

@Test
public void deleteTest() throws Exception{
    doNothing().when(tagService).delete(0L);

    mockMvc.perform(delete("/api/tags/1"))
        .andExpect(status().isOk());
}
}
```

L'annotation **@SpringBootTest** indique que cette classe effectue des tests basés sur SpringBoot. L'annotation **@RunWith** vient de JUnit, elle indique que la classe de test doit utiliser la classe spécifiée en paramètre pour lancer les tests, et non la classe qu'utilise JUnit par défaut.

# DOSSIER PROJET

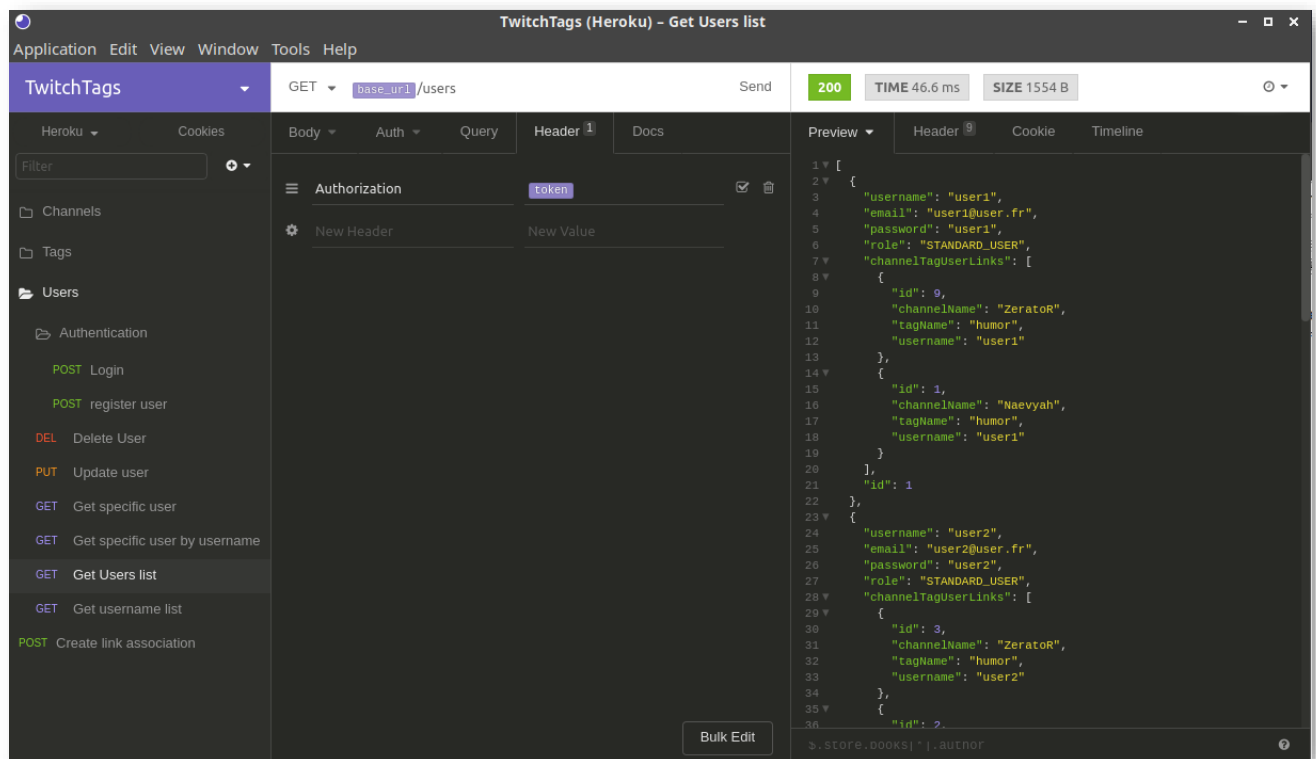
L'attribut **MockMvc** permet aux tests de se lancer sans avoir besoin de lancer le serveur, elle va de pair avec l'annotation **@MockBean** qui indique la classe à "mock", dans ce cas **TagService**, et pour pouvoir initialiser la classe **MockMvc**, il y a besoin de **WebApplicationContext**.

La création du mock est réalisée pendant l'exécution de la méthode **setUp()**, et cette dernière a lieu juste avant chaque test comme l'indique l'annotation **@Before**.

Le premier test, **listAll()** est chargé de tester la méthode **listAll()** de **TagController**. La première ligne : **when(tagService.listAll()).thenReturn(tags);** demande à **tagService** de renvoyer la liste de tag que nous avons initialisé pendant **setUp()**, à chaque fois que sa méthode **listAll()** est appelée. Ensuite avec **mockMvc**, on effectue une requête **GET** sur l'endpoint correspondant à la méthode **listAll()** du controller: **mockMvc.perform(get("/api/tags").contentType(MediaType.APPLICATION\_JSON))**. Les lignes suivantes vérifient que la réponse est bien un code 200, et qu'elle contient bien les données attendues.

## Test des endpoints de l'API

Enfin, j'ai utilisé le logiciel **Insomnia** pour faire des requêtes pour tester mes **controllers**. Dans un souci d'organisation et pour pouvoir les réutiliser facilement plus tard, j'ai mis mes requêtes dans des dossiers séparés selon qu'elles concernent les **channels**, les **users** ou les **tags**.

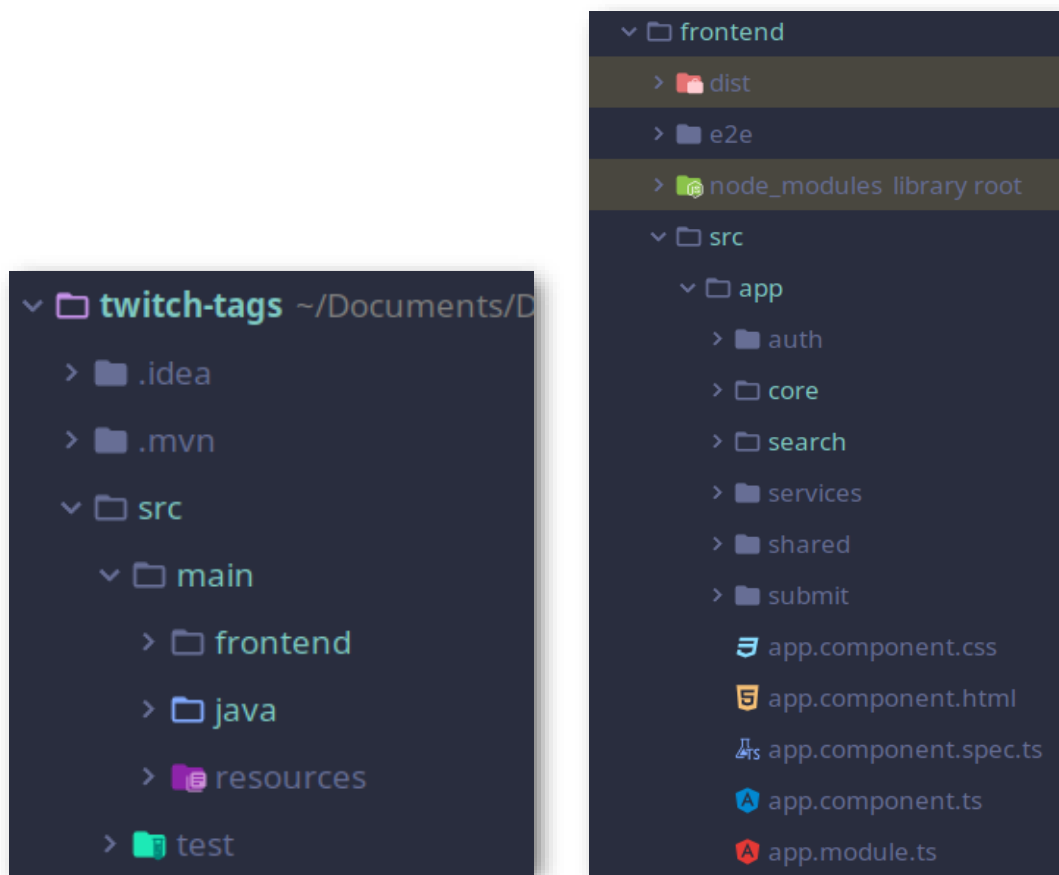


# DOSSIER PROJET

## FRONT-END

### ANGULAR

Pour la réalisation de mon front-end, j'ai choisi d'initialiser mon projet Angular dans mon projet SpringBoot, dans le dossier **main**. J'ai donc généré mon projet dans un dossier nommé **frontend** dont voici la structure :



*Emplacement et structure du projet Angular*

J'ai réparti mon projet en plusieurs modules pour plus de clarté. J'ai choisi de faire trois feature modules, **auth**, **search** et **submit**, un **shared** module et un **core** module.

- Un **feature module** est comme son nom l'indique un module contenant tous les composants et services nécessaires à une fonctionnalité en particulier.

# DOSSIER PROJET

- Le **core module** contient les composants nécessaires au fonctionnement basique de l'application, comme le **header**, le **footer** ou la **home** par exemple, mais aussi les services utilisés par plusieurs autres modules.
- Le **shared module** regroupe les composants nécessaires à plusieurs modules à la fois.

J'ai commencé par créer les **classes typescript** représentant mes **objets Java** (channel, tag, user et channel-tag-user-link) dans un dossier **models**, dans le **shared module**, puisque ces classes seront utilisées dans tout mon application.

Voici par exemple la classe Channel :

```
export class Channel {
  id: number;
  name: string;
  url: string;
  language: string;
  avatar: string;
  followers: number;
  partner: boolean;
  affiliate: boolean;
  status: string;
  channelTagUserLinks: ChannelTagUserLink[];

  constructor(name: string, url: string, language: string, avatar: string,
    followers: number, partner: boolean, status: string, channelTagUserLinks?:
    ChannelTagUserLink[]) {
    this.name = name;
    this.url = url;
    this.language = language;
    this.avatar = avatar;
    this.followers = followers;
    this.partner = partner;
    this.status = status;
    this.affiliate = false;
    if(channelTagUserLinks){
      this.channelTagUserLinks = channelTagUserLinks;
    }
    this.channelTagUserLinks = [];
  }
}
```

On y retrouve les mêmes attributs que dans la classe Java, ainsi qu'un constructeur.

Dans le composant home, j'affiche une liste de tous les channels existant dans la base de données. Pour réaliser cette fonctionnalité, j'ai créé le service suivant :

# DOSSIER PROJET

```
export class ChannelsService {

  private defaultPath = '/api/channels/';
  private createPath = this.defaultPath + 'create';
  private existencePath = this.defaultPath + 'is-present';
  private searchPath = this.defaultPath + 'search';
  private searchNamePath = this.defaultPath + 'search/name';
  private twitchApiPath = 'https://api.twitch.tv/kraken/channels/';
  private twitchApiHeaders = new HttpHeaders().set('Client-ID', 'wsdfgfvhbjnkvjwh18d39');

  constructor(private http: HttpClient) {
  }

  getChannels() {
    return this.http.get<Channel[]>(this.defaultPath);
  }

  getChannel(id: number) {
    const idPath = this.defaultPath + id;
    return this.http.get<Channel>(idPath);
  }

  updateChannel(channel: Channel) {
    const idPath = this.defaultPath + channel.id;
    return this.http.put<Channel[]>(idPath, channel);
  }

  saveChannel(channel: Channel) {
    return this.http.post<Channel>(this.createPath, channel);
  }

  deleteChannel(channel: Channel) {
    const idPath = this.defaultPath + channel.id;
    return this.http.delete(idPath);
  }

  getDataFromTwitchApi(channel: string) {
    const path = this.twitchApiPath + channel;
    return this.http.get<TwitchChannel>(path, {
      headers: this.twitchApiHeaders
    }).pipe(catchError( err => throwError(err)));
  }

  alreadyExist(channelName: string) {
    return this.http.post(this.existencePath, channelName);
  }

  search(tags: string){
    return this.http.get(this.searchPath + '?tag=' + tags);
  }

  searchName(name: string){
    return this.http.get(this.searchNamePath + '?channel=' + name);
  }
}
```



# DOSSIER PROJET

**ChannelsService** est chargé de communiquer avec l'API REST de mon backend pour toutes les opérations à effectuer en rapport avec les channels. Ainsi, pour récupérer la liste des channels, le service va faire une requête HTTP GET sur l'endpoint correspondant de l'API :

```
getChannels() {  
  return this.http.get<Channel[]>(this.defaultPath);  
}
```

Etant donné qu'il s'agit là d'une opération asynchrone, le composant voulant récupérer la réponse de cette requête va devoir s'y abonner avec la méthode **subscribe()**. C'est ce que fait la méthode **getChannels()** de notre composant, que l'on appelle dans le **ngOnInit()** pour que les channels soient récupéré à l'initialisation du composant :

```
export class PopularComponent implements OnInit {  
  channels: Channel[];  
  
  constructor(private channelService: ChannelsService) {  
    this.channels = [];  
  }  
  
  ngOnInit() {  
    this.getChannels();  
  }  
  
  getChannels() {  
    this.channelService.getChannels().subscribe(  
      (channels: Channel[]) => {  
        this.channels = channels;  
      },  
      error => console.error(error)  
    );  
  }  
}
```

Si on retourne sur **ChannelsService**, on peut voir que la méthode **getDataFromTwitchApi()** est différentes des autres. En effet, je voulais que lorsqu'un nouveau **channel** est rajouté à mon application, celle-ci aille chercher toutes les informations correspondantes directement sur l'API mise à disposition par Twitch, plutôt que d'obliger l'utilisateur à renseigner toutes les informations lui-même. Cela permet aussi d'éviter les erreurs et d'avoir des informations à jour.

```
getDataFromTwitchApi(channel: string) {  
  const path = this.twitchApiPath + channel;  
  return this.http.get<TwitchChannel>(path, {  
    headers: this.twitchApiHeaders  
  }).pipe(catchError( err => throwError(err)));  
}
```

# DOSSIER PROJET

Le composant **SubmitChannel** permet à l'utilisateur de rajouter un nouveau **channel** dans l'application :

```
export class SubmitChannelComponent implements OnInit {

  private _error = false;
  private _message = '';
  submittedChannel: string;

  constructor(private service: ChannelsService, private router: Router, private
title: Title) { }

  ngOnInit() {
    this.title.setTitle('TwitchTags - Submit channel');
  }

  submitChannel(){
    this.service.alreadyExist(this.submittedChannel).subscribe((data: boolean) => {
      if(data) {
        this._error = true;
        this._message = 'This channel already exist in database.';
      } else {
        this.service.getDataFromTwitchApi(this.submittedChannel).subscribe(data => {
          if(data.status == '404') {
            this._error = true;
            this._message = 'This channel does not exist.';
          } else {
            let status = 'NONE';
            if (data.partner) status = 'PARTNER';

            let channel: Channel = new Channel(data.display_name,
                                                data.url,
                                                data.broadcaster_language,
                                                data.logo,
                                                data.followers,
                                                data.partner,
                                                status );

            this.service.saveChannel(channel).subscribe(data => {
              this.router.navigate(['/app/profile/' + data.id]);
            })
          }
        }, error1 => {
          console.log("Channel not found " + error1);
          this._error = true;
          this._message = 'This channel does not exist.';
        })
      }
    });
  }
}
```

Pour se faire, il va vérifier que le channel n'existe pas déjà dans la base de données, et le cas échéant, il va essayer de récupérer les informations sur l'API Twitch via le service. Si le channel existe bien

# DOSSIER PROJET

sur Twitch, ses informations sont récupérées et utilisées pour créer un nouvel objet Channel que le service va ensuite sauver en base de données. L'utilisateur est enfin redirigé vers la page du channel nouvellement créé.

## AUTHENTIFICATION

Après avoir créé tous les composants et services nécessaires au fonctionnement basique de mon application, je suis passée à l'implémentation de l'authentification.

Pour ça j'ai créé le **service** suivant pour effectuer les requêtes sur l'API:

```
const httpOptions = {
  headers: new HttpHeaders({'Content-Type': 'application/json'})
};

export class AuthService {

  private loginUrl = '/api/auth/signin';

  private signupUrl = '/api/auth/signup';

  constructor(private http: HttpClient) {

  }

  attemptAuthentication(credentials: LoginInfo): Observable<JwtResponse> {
    return this.http.post<JwtResponse>(this.loginUrl,
                                      credentials,
                                      httpOptions);
  }

  signup(info: SignupInfo): Observable<string> {
    return this.http.post<string>(this.signupUrl, info, httpOptions);
  }
}
```

La méthode **attemptAuthentication()** prend en paramètre les informations d'identification de l'utilisateur et renvoi un objet de type **JwtResponse**. La class **JwtResponse** est un simple objet contenant le token JWT, le pseudo et le rôle de l'utilisateur.

J'ai aussi besoin d'un service qui va stocker le token dans la session du navigateur, il s'agit de **TokenStorageService** :

# DOSSIER PROJET

```
const TOKEN_KEY = 'AuthToken';
const USERNAME_KEY = 'AuthUsername';
const AUTHORITIES_KEY = 'AuthAuthorities';

export class TokenStorageService {
  private roles: Array<string> = [];

  constructor(){}

  signOut(){
    window.sessionStorage.clear();
  }

  public saveToken(token: string){
    window.sessionStorage.removeItem(TOKEN_KEY);
    window.sessionStorage.setItem(TOKEN_KEY, token);
  }

  public getToken(): string {
    return sessionStorage.getItem(TOKEN_KEY);
  }

  public saveUsername(username: string) {
    window.sessionStorage.removeItem(USERNAME_KEY);
    window.sessionStorage.setItem(USERNAME_KEY, username);
  }

  public getUsername(): string {
    return sessionStorage.getItem(USERNAME_KEY);
  }

  public saveAuthorities(authorities: string[]) {
    window.sessionStorage.removeItem(AUTHORITIES_KEY);
    window.sessionStorage.setItem(AUTHORITIES_KEY, JSON.stringify(authorities));
  }

  public getAuthorities(): string[] {
    this.roles = [];
    if(sessionStorage.getItem(TOKEN_KEY)){
      JSON.parse(sessionStorage.getItem(AUTHORITIES_KEY)).forEach(authority => {
        this.roles.push(authority.authority);
      });
    }
    return this.roles;
  }

  public isAuthenticated() {
    return !!this.getToken();
  }

  public isAdmin() {
    let authorities = this.getAuthorities();
    return authorities.includes('ROLE_ADMIN');
  }
}
```

Ce service va aussi stocker le pseudo et le rôle de l'utilisateur, et possède des méthodes pour déterminer si l'utilisateur est bien identifié et s'il a le rôle d'administrateur.

# DOSSIER PROJET

Le composant **SigninComponent** a la charge de récupérer les identifiants de l'utilisateur à partir d'un formulaire, puis de récupérer le token transmis par **AuthService** et de le passer à **TokenStorageService** pour qu'il le stocke.

```
<ng-template #loggedOut>
<form class="container" [formGroup]="loginForm" (ngSubmit)="login()" >
  <div class="columns">
    <div class="column"></div>
    <div class="column is-one-third">

      <div class="card has-text-centered">
        <div class="card-content">
          <h1 class="title">Sign In</h1>
          <div class="field">
            <p class="control has-icons-left has-icons-right">
              <input formControlName="username" name="username" class="input" type="text"
placeholder="Username" [ngClass]="{ 'is-danger': submitted && f.username.errors}">
              <span class="icon is-small is-left">
                <i class="fas fa-envelope"></i>
              </span>
            </p>
            <div *ngIf="submitted && f.username.errors">
              <p *ngIf="f.username.errors.required" class="help is-danger has-text-left">This
field is required</p>
            </div>
          </div>
          <div class="field">
            <p class="control has-icons-left">
              <input formControlName="password" name="password" class="input" type="password"
placeholder="Password" [ngClass]="{ 'is-danger': submitted && f.password.errors}">
              <span class="icon is-small is-left">
                <i class="fas fa-lock"></i>
              </span>
            </p>
            <div *ngIf="submitted && f.password.errors">
              <p *ngIf="f.password.errors.required" class="help is-danger has-text-left">This
field is required</p>
            </div>
          </div>
          <div class="field">
            <p class="control has-text-centered">
              <button class="button">
                Login
              </button>
            </p>
          </div>
          <div *ngIf="submitted && isLoginFailed" class="alert alert-danger">
            Login failed: username and/or password incorrect
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

*Page HTML de SigninComponent contenant le formulaire*

# DOSSIER PROJET

```
export class SigninComponent implements OnInit {
  loginForm: FormGroup;
  submitted = false;
  isLoggedIn = false;
  isLoginFailed = false;
  roles: string[];
  private loginInfo: LoginInfo;
  errorMessage = '';
  isAdmin = false;

  constructor(private route: Router, private formBuilder: FormBuilder, private
authService: AuthService, private tokenStorage: TokenStorageService, private
title: Title) { }

  ngOnInit() {
    this.title.setTitle('TwitchTags - Log in');
    this.loginForm = this.formBuilder.group({
      username: ['', [Validators.required]],
      password: ['', [Validators.required]]
    });
    if(this.tokenStorage.getToken()){
      this.isLoggedIn = true;
      this.roles = this.tokenStorage.getAuthorities();
      this.isAdmin = this.tokenStorage.isAdmin();
    }
  }
  login() {
    this.submitted = true;
    if(this.loginForm.invalid) {
      console.log('invalid form');
      return;
    }
    this.loginInfo = new LoginInfo(this.loginForm.value.username,
this.loginForm.value.password);
    this.authService.attemptAuthentication(this.loginInfo).subscribe(
      data => {
        this.tokenStorage.saveToken(data.token);
        this.tokenStorage.saveUsername(data.username);
        this.tokenStorage.saveAuthorities(data.authorities);
        this.isLoginFailed = false;
        this.isLoggedIn = true;
        this.roles = this.tokenStorage.getAuthorities();
        this.isAdmin = this.tokenStorage.isAdmin();
        window.location.reload();
      },
      error => {
        console.log(error);
        this.errorMessage = error.error.message;
        this.isLoginFailed = true;
      }
    );
  }
}
```

*Classe typescript de SigninComponent*

# DOSSIER PROJET

Enfin, j'ai créé un HTTP interceptor, nommé **AuthInterceptor** ici :

```
const TOKEN_HEADER_KEY = 'Authorization';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor(private tokenService: TokenStorageService){}
  intercept(req: HttpRequest<any>, next: HttpHandler):
  Observable<HttpEvent<any>> {
    let authRequest = req;
    const token = this.tokenService.getToken();
    if(token != null) {
      authRequest = req.clone({headers: req.headers.set(TOKEN_HEADER_KEY,
'Bearer ' + token)});
    }
    return next.handle(authRequest);
  }
}

export const HttpInterceptorProviders = [{provide: HTTP_INTERCEPTORS,
useClass: AuthInterceptor, multi: true}];
```

Il a la charge d'intercepter toutes les requêtes HTTP émises par l'application, pour y ajouter un header contenant le token de l'utilisateur. Ainsi les requêtes vers les endpoints sécurisés de l'API ne seront pas rejetées.

J'ai également créé une classe **AuthGuard**, implémentant **CanActivate**, qui consiste à filtrer l'accès de mes routes. Dans le cas présent, l'utilisateur ne pourra accéder aux pages sécurisées par l'AuthGuard que s'il est authentifié.

```
@Injectable()
export class AuthGuard implements CanActivate {

  constructor(private tokenService: TokenStorageService) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.tokenService.isAuthenticated();
  }
}
```

Pour appliquer l'**AuthGuard** à des composants, il faut l'ajouter au fichier de routing via la propriété canActivate. J'ai aussi créé un deuxième guard, **AuthAdminGuard**, pour cette fois n'autoriser l'accès qu'aux utilisateurs ayant le rôle d'administrateur :

# DOSSIER PROJET

```
@Injectable()
export class AuthAdminGuard implements CanActivate {

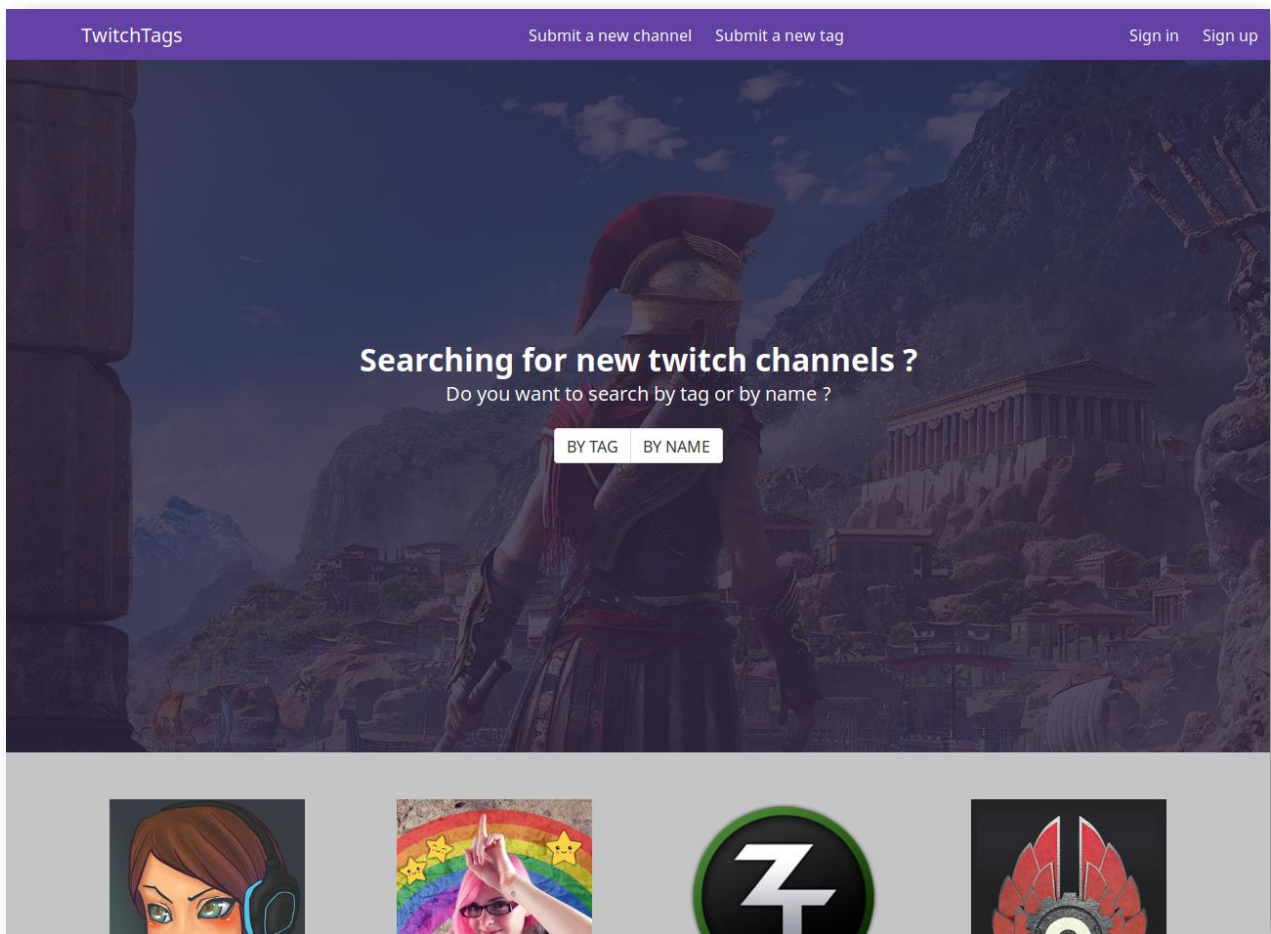
  constructor(private tokenService: TokenStorageService) { }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.tokenService.isAdmin();
  }
}
```

## RESPONSIVE

Pour finir, j'ai amélioré l'apparence de mon interface grâce au framework CSS **Bulma**, et j'ai modifié quelques éléments pour que l'application soit bien responsive.

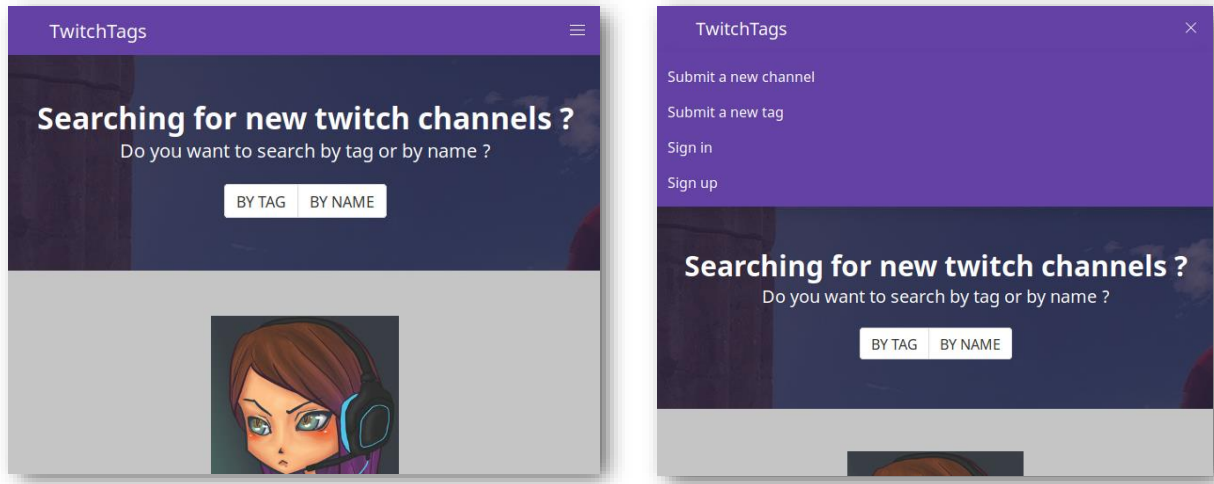
Par exemple, ma barre de navigation en mode desktop, devient un menu burger en mode mobile :



Site en mode desktop



# DOSSIER PROJET



Site en mode mobile

Lorsque l'on clique sur le bouton burger, cela change la valeur de la variable **toggled**, ce qui a pour effet de rendre active la div **navbarLinksMobile** et ainsi l'afficher.

```
export class HeaderComponent implements OnInit {
  private roles: string[];
  private _authority: string;
  toggled: boolean = false;

  constructor(private tokenStorage: TokenStorageService, private router:
Router) { }

  ngOnInit() {
    if(this.tokenStorage.getToken()){
      this.roles = this.tokenStorage.getAuthorities();
      this.roles.every(role => {
        if(role === 'ROLE_ADMIN') {
          this._authority = 'admin';
          return false;
        }
        this._authority = 'user';
        return true;
      });
    }
  }

  toggleMenuBurger(){
    this.toggled = this.toggled != true;
  }
}
```

Classe TypeScript de HeaderComponent

# DOSSIER PROJET

```
<nav class="navbar header">
  <div class="navbar-brand">
    <a class="navbar-item" routerLink="/app">TwitchTags</a>
    <div aria-label="menu" aria-expanded="false" class="navbar-burger burger" data-
target="navbarLinks" (click)="toggleMenuBurger()" [ngClass]="{'is-active': toggled}">
      <span></span>
      <span></span>
      <span></span>
    </div>
  </div>
  <div id="navbarLinks" class="navbar-menu">
    <div class="navbar-start">
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/submitchannel">Submit a new channel</a>
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/submittag">Submit a new tag</a>
    </div>
    <div class="navbar-end" *ngIf="!authority; else LoggedIn">
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/signin">Sign in</a>
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/signup">Sign up</a>
    </div>
    <ng-template #LoggedIn>
      <div class="navbar-end">
        <a class="navbar-item" routerLinkActive="active-link" routerLink="/app/admin"
*ngIf="authority === 'admin'">Admin Dashboard</a>
        <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/account">My Account</a>
        <a class="navbar-item" style="cursor: pointer" (click)="logout()">Logout</a>
      </div>
    </ng-template>
  </div>
  <div id="navbarLinksMobile" class="navbar-menu" [ngClass]="{'is-active': toggled}"
*ngIf="toggled" (click)="toggleMenuBurger()">
    <div class="navbar-end">
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/submitchannel">Submit a new channel</a>
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/submittag">Submit a new tag</a>
    </div>
    <div class="navbar-end" *ngIf="!authority; else LoggedIn">
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/signin">Sign in</a>
      <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/signup">Sign up</a>
    </div>
    <ng-template #LoggedIn>
      <div class="navbar-end">
        <a class="navbar-item" routerLinkActive="active-link" routerLink="/app/admin"
*ngIf="authority === 'admin'">Admin Dashboard</a>
        <a class="navbar-item" routerLinkActive="active-link"
routerLink="/app/account">My Account</a>
        <a class="navbar-item" style="cursor: pointer" (click)="logout()">Logout</a>
      </div>
    </ng-template>
  </div>
</nav>
```

Template HTML de HeaderComponent

# DOSSIER PROJET

## REFERENCEMENT

Concernant le référencement de mon application, j'ai choisi de modifier dynamiquement le contenu de ma balise **<title>** selon les différentes pages de mon application pour qu'elle corresponde le mieux possible au contenu.

```
export class HomeComponent implements OnInit {  
  constructor(private router: Router, private title: Title) { }  
  
  ngOnInit() {  
    this.title.setTitle('TwitchTags - Home');  
  }  
}
```

*La balise <title> est mise à jour à l'initialisation du composant HomeComponent*

J'ai aussi utilisé les attributs **ALT** pour décrire le contenu de mes quelques images :

```
<div class="column is-half">  
  <img [src]="channel.avatar" alt="{{ channel.name }}"s avatar">  
</div>
```

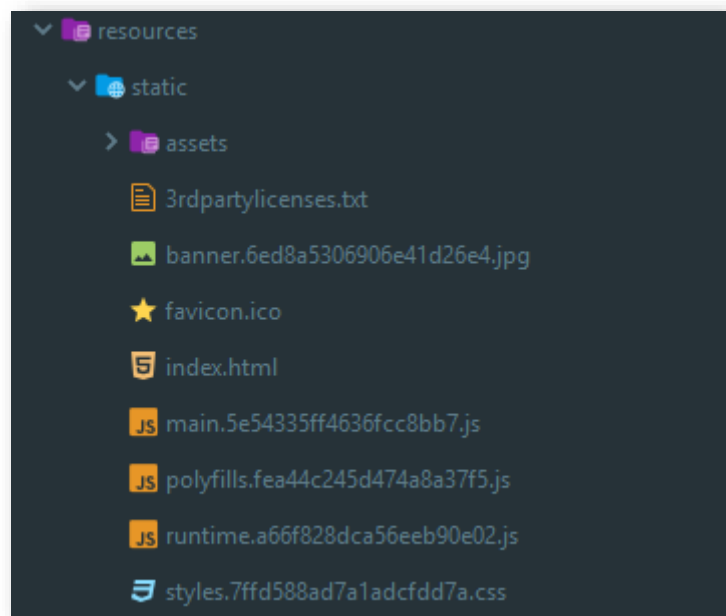
*L'attribut ALT de l'image dans le composant ChannelProfile est bien défini*

# DOSSIER PROJET

## DEPLOIEMENT

En ce qui concerne le déploiement de mon application, j'ai choisi d'utiliser le service **Heroku** car celui-ci possède un plan gratuit et il est compatible avec le langage Java. Aussi il propose de déployer une application à partir d'un repo GitHub, c'est donc très pratique dans mon cas.

Pour déployer mon application, j'ai d'abord modifié mon fichier **angular.json** afin que les fichiers générés lors du build aillent directement dans un dossier **static**, situé dans **resources**. En effet, **SpringBoot** est configuré pour servir le contenu static du dossier resources. Ensuite j'ai donc lancé la commande **ng build --prod** pour qu'Angular génère les fichiers nécessaires.



*Les fichiers générés lors du build se retrouvent bien dans le dossier static*

Puis j'ai créé un fichier nommé Procfile à la racine de mon projet :

```
web: java $JAVA_OPTS -Dserver.port=$PORT -jar target/*.jar --p $PORT
```



Ce fichier spécifie les commandes devant être exécutées par les **dynos** de l'application sur **Heroku**. Les **dynos** sont des conteneurs virtuels et isolés spécifiques à Heroku, ils sont chargés d'exécuter le code spécifié par les utilisateurs. Ensuite j'ai pushé mon application sur une nouvelle branche sur **GitHub**, créée spécialement pour le déploiement.


# DOSSIER PROJET

Sur le site de **Heroku**, j'ai créé une nouvelle application, j'ai choisi son adresse puis je l'ai liée à mon repository sur GitHub et j'ai choisi la branche que je souhaitais déployer :

### App connected to GitHub

Code diffs, manual and auto deploys are available for this app.

Connected to  [GwenRspl/twitch-tags](#) by  [GwenRspl](#) [Disconnect...](#)

 Releases in the [activity feed](#) link to GitHub to view commit diffs



### Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

#### Enable automatic deploys from GitHub

Every push to the branch you specify here will deploy a new version of this app. **Deploys happen automatically:** be sure that this branch is always in a deployable state and any tests have passed before you push. [Learn more.](#)

**Choose a branch to deploy**

 master 

☐ Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

Enable Automatic Deploys



### Manual deploy

Deploy the current state of a branch to this app.

#### Deploy a GitHub branch

This will deploy the current state of the branch you specify below. [Learn more.](#)

**Choose a branch to deploy**

 master 

Deploy Branch

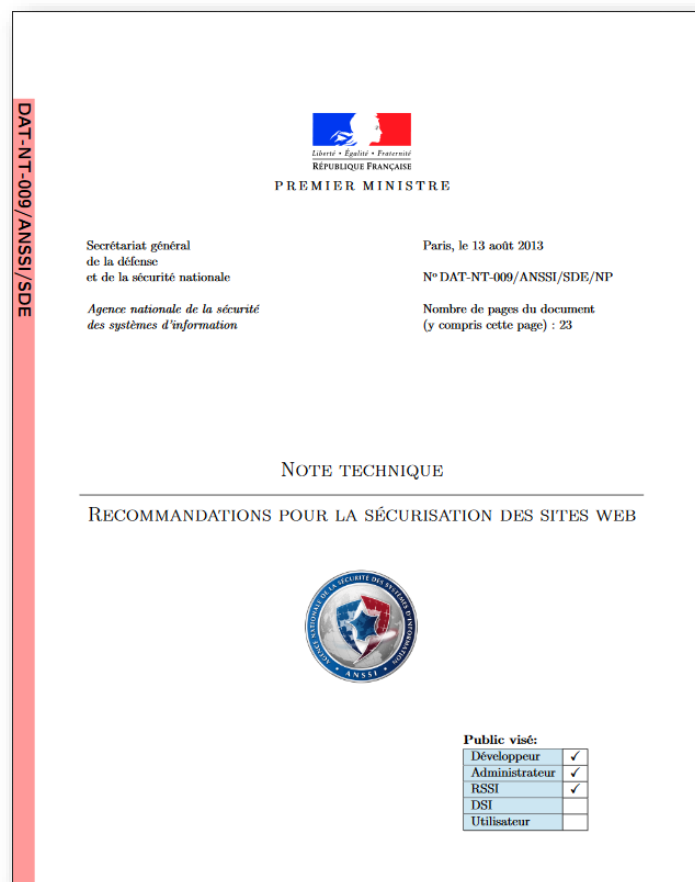
Mon application est donc visitable à l'adresse [twitch-tags.herokuapp.com](https://twitch-tags.herokuapp.com)

# **VEILLE TECHNOLOGIQUE**

# DOSSIER PROJET

## SECURITE

Quand j'effectuais ma veille sur les vulnérabilités et la sécurité pour mon application, je suis allée sur le site de l'**ANSSI** (Agence Nationale de la Sécurité des Systèmes d'Information) et j'ai cherché des informations parmi leurs guides de bonnes pratiques. Leur guide pour la sécurisation de site web m'a été utile :



Un exemple de règle du guide que j'ai suivi est de ne pas stocker les mots de passes utilisateurs en clair dans la base de données, et de les hasher en y ajoutant un sel aléatoire :

**R22** Les mots de passe ne doivent pas être stockés en clair mais dans une forme transformée par une fonction cryptographique non réversible conforme au [Référentiel Général de Sécurité](#).

**R23** La transformation des mots de passe doit faire intervenir un *sel* aléatoire.

# DOSSIER PROJET

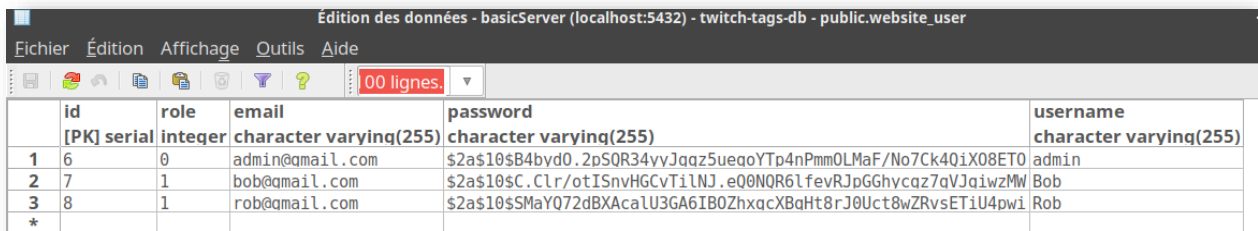
C'est une règle que j'ai appliqué dans mon application : le mot de passe de l'utilisateur est hashé puis stocké dans la base de données. Mon encoder de mot de passe est un encoder **BCrypt**, contrairement à d'autres implémentations comme **SHAPasswordEncoder** par exemple, le client n'a pas besoin de lui spécifier le sel, **BCrypt** le génère tout seul aléatoirement. Cela veut dire que chaque appel aura un résultat différent.

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

*Le PasswordEncoder est défini dans la classe WebSecurityConfig*

```
User user = new User(signupRequest.getUsername(), signupRequest.getEmail(),
encoder.encode(signupRequest.getPassword()));
```

*Le hashage du mot de passe a lieu dans le AuthController*



Édition des données - basicServer (localhost:5432) - twitch-tags-db - public.website\_user

	id [PK] serial	role integer	email character varying(255)	password character varying(255)	username character varying(255)
1	6	0	admin@gmail.com	\$2a\$10\$B4byd0.2pSQR34vyJqaz5uegoYTp4nPmmOLMaF/No7Ck4QiX08ET0	admin
2	7	1	bob@gmail.com	\$2a\$10\$C.Clr/otISnvHGCvTilNJ.eQ0NQ6lfevRJpGGhycaz7qVJaiwzMW	Bob
3	8	1	rob@gmail.com	\$2a\$10\$SMaYQ72dBXAcalU3GA6IB0ZhxacXBqHt8rJ0Uct8wZRvsETiU4pwi	Rob
*					

*Dans la base de données, les mots de passe sont bien cryptés*

Pour avoir plus de renseignements sur le fonctionnement du hachage de mot de passe, j'ai visité le site <https://crackstation.net/hashing-security.htm>. Il explique les bonnes pratiques, comme générer le sel avec un **Générateur de Nombres Pseudo-Random Sécurisé Cryptographiquement** (Cryptographically Secure Pseudo-Random Number Generator) si notre encoder ne le fait pas seul, et les choses à ne pas faire, comme écrire soi-même son programme de hachage.



## SITUATION DE TRAVAIL AYANT NECESSITE UNE RECHERCHE

### CONTEXTE DE RECHERCHE

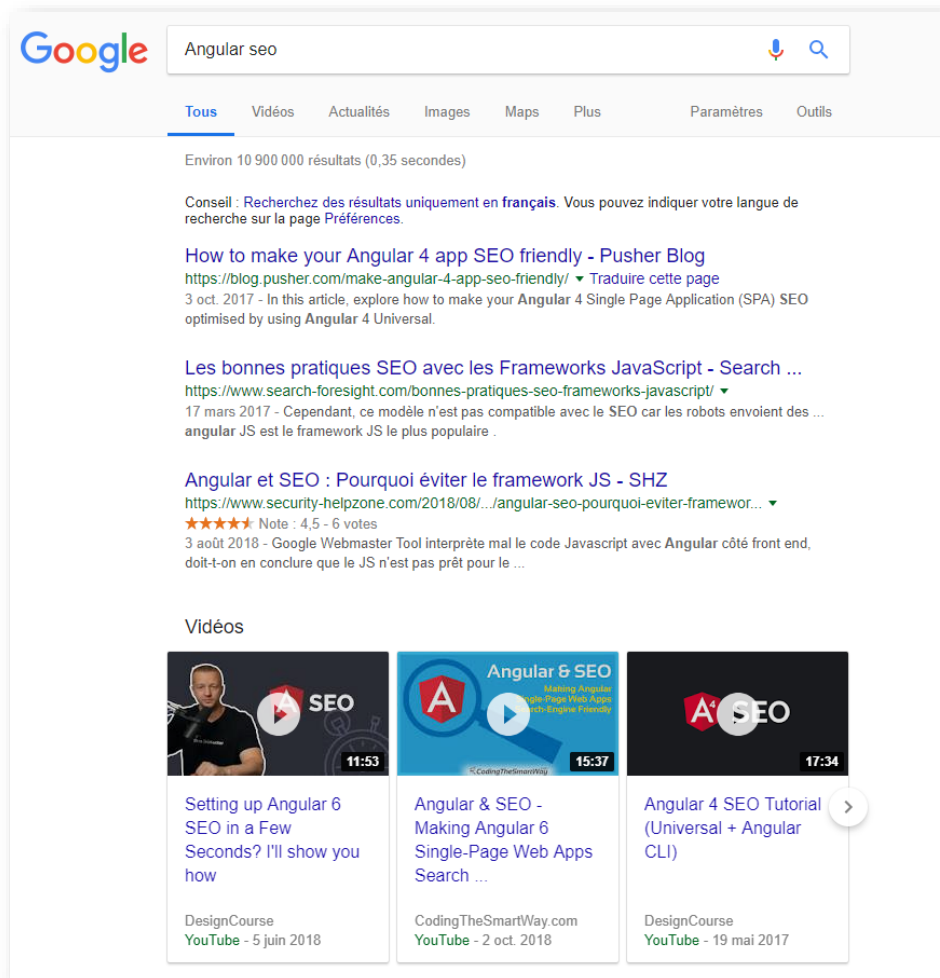
Pour mon exemple de recherche de solution, je vais prendre le moment où j'ai voulu améliorer le référencement naturel de mon application. En effet, les applications **Angular** étant des **applications mono-page**, leur contenu peut ne pas être visible par certains robots car leur contenu change dynamiquement grâce à JavaScript. Si on regarde le code source de mon application, on ne voit effectivement rien dans l'app-root :



```
1 <!doctype html>
2 <html lang="en">
3
4 <head>
5   <meta charset="utf-8">
6   <title>TwitchTags</title>
7   <base href="/">
8
9   <meta name="viewport" content="width=device-width, initial-scale=1">
10  <meta name="author" content="GwenRspl">
11  <meta name="keywords" content="twitch, tag, channel, vote, streamer, viewers">
12  <meta name="description" content="This is an Angular 6 application about Twitch channels.">
13  <link rel="icon" type="image/x-icon" href="favicon.ico">
14
15  <link href="https://fonts.googleapis.com/css?family=Noto+Sans:400,700" rel="stylesheet">
16  <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.3.1/css/all.css" integrity="sha384-
mzrmE5qonljUremFsqc0tSB46JvROS7bZs3IO2EmfFsd15uHvIt+Y8vEf7N7fWAU" crossorigin="anonymous">
17
18  <link rel="stylesheet" href="styles.7ffd588ad7a1adcfdd7a.css"></head>
19
20 <body>
21   <app-root></app-root>
22   <script type="text/javascript" src="runtime.a66f828dca56eeb90e02.js"></script><script type="text/javascript"
src="polyfills.fea44c245d474a8a37f5.js"></script><script type="text/javascript" src="main.5e54335ff4636fcc8bb7.js">
</script></body>
23
24 </html>
25
```

# DOSSIER PROJET

J'ai donc fait des recherches pour rendre le contenu de mon site visible. Pour ça j'ai tapé "Angular seo" dans google, qui m'a renvoyé cette page de résultat :

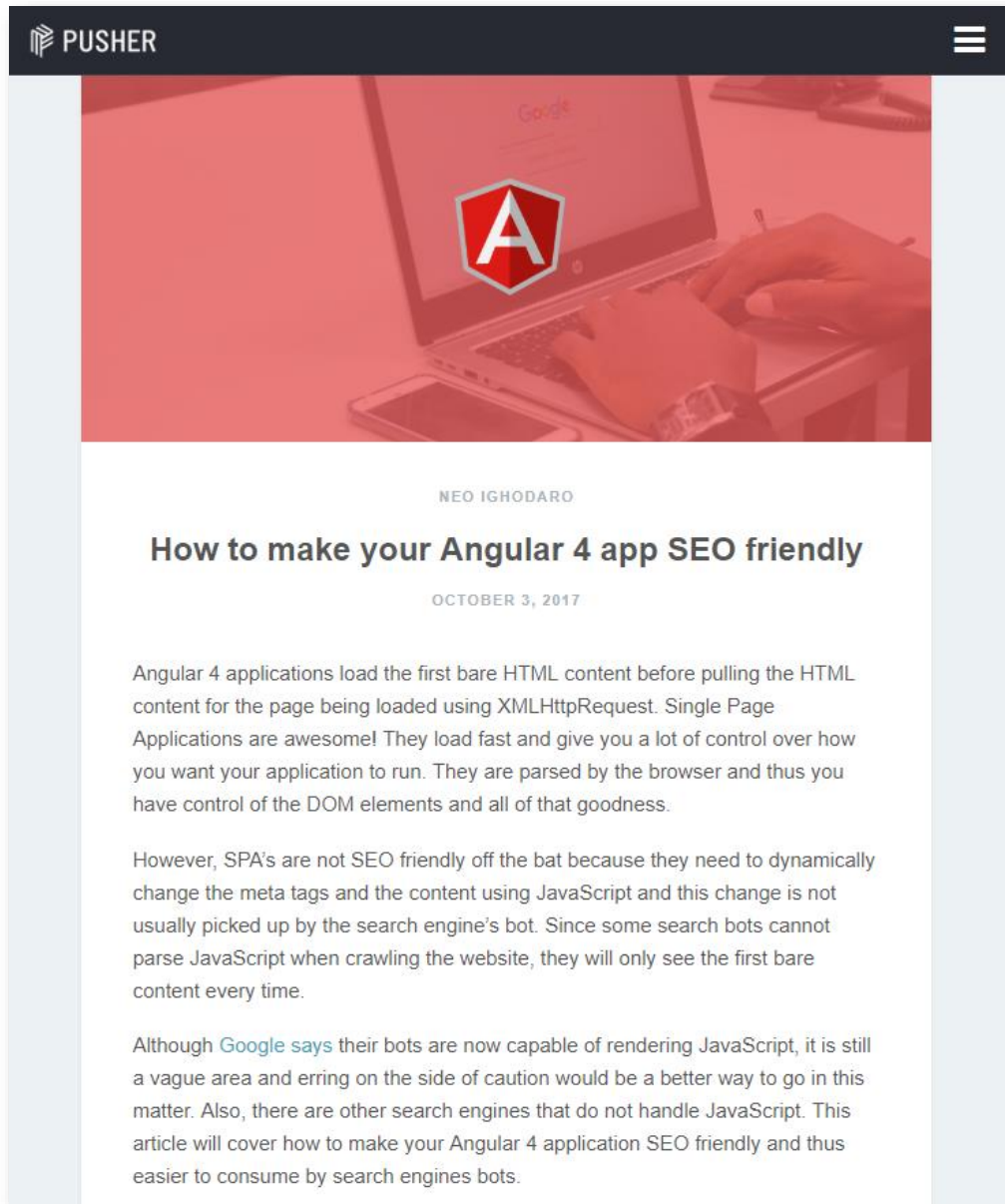


La meilleure solution est d'utiliser **Angular Universal** pour que mon application soit dans un premier temps générée sur un serveur, puis ensuite transmise au client. J'ai d'abord essayé la solution détaillée par cette page : <https://coursetro.com/posts/code/155/Angular-6-SEO-Tutorial-from-Scratch---It%27s-Super-Simple-Now!>

C'est la plus simple et elle est compatible avec la version d'Angular qu'utilise mon application. J'ai réussi à l'implémenter sur un projet nouvellement créé après avoir résolu plusieurs erreurs dues à des versions de dépendances incompatibles entre elles. Cependant, je n'ai pas réussi à l'implémenter sur mon application, sa structure étant plus complexe qu'un simple projet tout neuf.

J'ai trouvé une autre solution sur le site <https://blog.pusher.com/make-angular-4-app-seo-friendly/>, mais celle-ci a l'air plus longue et difficile à mettre en place, j'ai donc préféré remettre à plus tard son implémentation.

## TRADUCTION D'UN SITE AYANT SERVI A LA RECHERCHE



*Extrait d'un site utilisé pendant ma recherche*

Traduction de l'extrait :

Les applications Angular 4 charge d'abord le premier contenu HTML basique avant de récupérer le contenu HTML de la page étant en train d'être chargée en utilisant XMLHttpRequest. Les Applications Mono-Page sont géniale ! Elles chargent rapidement et vous donne beaucoup de contrôle sur la façon dont vous voulez que votre application fonctionne. Elles sont analysées par le navigateur et de cette façon vous avez le control sur les éléments du DOM.

Cependant, les applications mono-pages ne sont pas pratique pour le référencement naturel parce qu'elles ont besoin de dynamiquement changer les tags meta et le contenu en utilisant JavaScript, et ce changement n'est généralement pas repéré par les robots des moteurs de recherche. Comme certains robots ne peuvent pas analyser le JavaScript quand ils parcourent le site web, ils voient seulement le premier contenu de base à chaque fois.

Même si Google dit que ses robots sont désormais capables de déchiffrer le JavaScript, c'est encore un domaine vague et il vaudrait mieux être prudent concernant ce problème. Aussi, il y a d'autres moteurs de recherche qui ne supportent pas le JavaScript. Cet article explique comment rendre votre application Angular 4 plus optimisé pour le référencement naturel et donc plus pratique à analyser pour les robots des moteurs de recherche.

## CONCLUSION

---

Mon année de formation chez Simplon.co et mon alternance chez PayinTech auront été un grand challenge pour moi, mais surtout un pari réussi. Quand j'ai quitté mon ancien poste pour entamer ma reconversion, je ne connaissais pas grand-chose à l'informatique et au code. Je m'étais formée en autodidacte sur des langages mais je n'avais aucune idée de comment réaliser un projet professionnel. J'ai pu acquérir les bases qu'il me manquait, je comprends mieux comment le front-end et le back-end s'articulent et je suis fière d'avoir pu réaliser un projet ainsi, toute seule de bout en bout.

Mes semaines en entreprise m'ont permis d'appréhender le travail en équipe et m'ont fait découvrir des outils professionnels et m'ont permis de m'améliorer aussi sur ma façon d'utiliser Git.

Concernant mon application, elle n'est pas encore tout à fait finie, il ne me reste plus qu'une fonctionnalité à implémenter pour qu'elle soit totalement opérationnelle. L'application est bien conforme aux maquettes et aux user stories que j'ai écrites pendant la phase de conception. J'ai aussi quelques idées de fonctionnalités supplémentaires que je pourrais implémenter dans une itération future, comme par exemple la possibilité de lier le compte Twitch de l'utilisateur à son compte sur l'application, ou encore faire un système de validation des tags ajoutés par les utilisateurs, afin qu'ils ne soient visibles que par la personne qui l'a soumis le temps qu'il soit validé par un administrateur.

Je pense avoir encore beaucoup à apprendre, mais c'est aussi une partie du travail de développeur que j'aime, les technologies étant en constante évolution, le développeur est en quelque sorte un éternel étudiant.

## REMERCIEMENTS

---

Je tiens à remercier mon équipe au sein de PayinTech : Pierre Adam pour m'avoir accueilli en alternance mais aussi pour m'avoir permis de commencer mon contrat plusieurs mois avant le début de la formation, afin que je ne me retrouve pas sans moyens, et malgré le fait que je partais presque de zéro niveau programmation. Mon tuteur, Olivier Buiron, pour son aide, sa confiance et ses retours sur mon code. Et bien sûr à mes autres collègues du bureau des développeurs, pour leur accueil et la bonne ambiance.

Je remercie aussi mes formateur.rice.s de Simplon.co, Sara Guerric et Yoann Blanchet, pour leur accompagnement, ainsi que mes collègues de formation pour leur soutien et leur aide sans lesquels je n'en serai pas là.

