



Rapport de Projet

Application web de location de voitures

Réalisé par :

- CHAOUB Rayane
- MARAH Abdelouahed
- EL ALAOUI Rachid
- REHAILY Mohammed-Rida

Professeur : M. SAMIR Youcef



**UNIVERSITÉ
DE LORRAINE**

LORRAINE INP
vos talents se lèvent à l'Est

Sommaire

I. Introduction

II. Partie Back-End

- i. Architecture de la partie Back-end
- ii. Configuration du projet

III. Partie Front-End

- i. Structure de notre projet Angular
- ii. Outils utilisés
- iii. Communication avec la Back-end

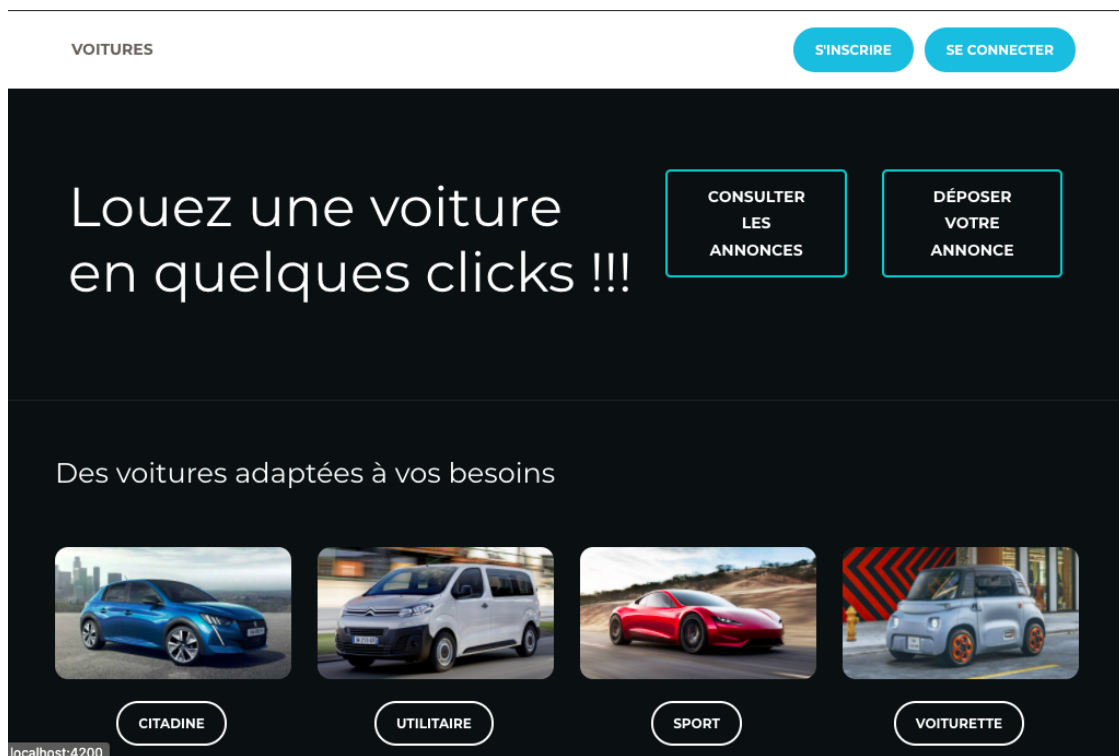
IV. Conclusion

Introduction

La tendance actuelle du web, dans l'évolution technologique comme dans l'évolution sociale, représente une avancée dans ses utilisations. En effet, être présent sur internet est devenu une réalité fréquente de nos jours. De ce fait plusieurs sites internet permettant de connecter des internautes qui proposent des services ou des produits avec d'autres qui en consomment ont vu le jour : (leboncoin, blablacar...)

Nous avons donc choisi pour notre projet tutoré de développer une application web de location des voitures où des utilisateurs pourront consulter les annonces d'autres personnes qui désirent mettre en location leurs voitures, avoir leurs coordonnées pour ensuite pouvoir les contacter, avec également la possibilité de s'inscrire pour pouvoir déposer son propre annonce si l'on souhaite, tout ceci est géré par Admin et Super Admin qui prennent les commandes du site, et qui ont en plus des avantages d'un utilisateur normal de supprimer des annonces ou des utilisateurs.

La page d'accueil de notre application est la suivante:



Architecture générale de l'application :

Pour donner vie à notre application on était amené à faire du développement Full Stack ce un terme désigne le développement du Back-end et du Front-end de notre application à la fois. Le développement la partie Back-end, c'est à dire l'application côté serveur signifie d'implémenter la logique derrière les différentes fonctionnalités que propose notre application ainsi que la manipulation et le traitement des données qui seront stockés dans une base de données, ensuite il fallait s'occuper du développement de la partie Front-end qui majoritairement concerne l'aspect visuel ("boutons, images et couleurs...") et affichage des données dans un certain format, dans notre cas par exemple on affichait les voitures dans des cartes.

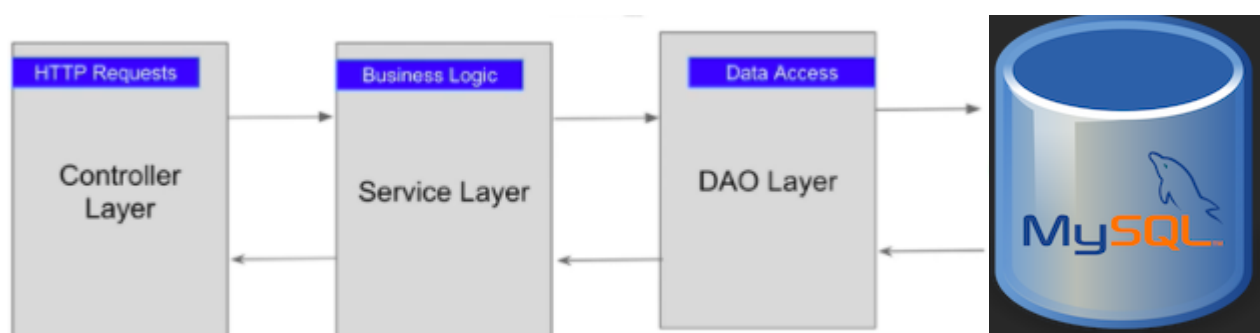
Qu'il s'agisse de développer la partie Frontend ou Backend, on avait une architecture à respecter afin d'assurer la cohérence et d'organiser le développement de l'application.

On va commencer dans ce qui va suivre par présenter dans un premier temps l'architecture de la partie Back-end et on passera par la suite à celle du Front-end.

Remarque : La partie front ne s'est pas réduite sur l'aspect visuel uniquement, on verra dans la suite qu'il y avait toute une logique derrière à implémenter.

Partie Back-End

Schéma modélisant l'architecture de la partie Back end:



Dans la partie back-end le développement est effectué côté serveur. L'image au-dessus illustre l'architecture de cette partie qui est divisée en trois couches chacune ayant ses propres responsabilités.

On commence d'abord par la couche d'accès aux données (DAO) qui est directement liée à la base de données. Cette couche s'occupe du traitement et

de l'accès aux données qui sont persistés au sein d'un SGBD, dans notre cas (MySQL), son implémentation se résumerait à des interfaces qui héritent de JpaRepository. Cette dernière nous fournit un ensemble de méthodes pour interagir avec une base de données.

Il vient ensuite la couche métier qui effectue les traitements métier de l'application; la logique de cette dernière, son implémentation sera sous forme de services qui vont se servir des interfaces de la couche d'accès aux données (DAO) pour effectuer les différents traitements relatifs à la logique de l'application.

Et finalement on a la couche qui est chargée de la redirection des requêtes HTTP, son rôle consiste à rediriger la requête que l'utilisateur envoie depuis son url vers la méthode qui est concernée par cette requête.

On verra dans les détails chaque couche en utilisant notre cas d'application qui nous servira d'exemple illustratif.

Configuration du projet :

Avant d'entamer la phase de développement il faut tout d'abord commencer par configurer l'environnement dans lequel on va travailler, on entend par configuration l'ajout de toutes les dépendances nécessaires et auxquelles on aura besoin dans la suite. La partie back-end a été entièrement développée en utilisant le framework Spring boot car il nous fournit les composants logiciels dont on aura besoin et nous évite d'écrire du code complexe.

On a commencé tout d'abord par générer le projet spring boot avec spring initializr intégré sur IntelliJ le langage de programmation choisi est Java version 1.8 (dit java 8) car il offre des expressions lambdas qui simplifient l'écriture du code, l'outil de construction est maven comme il offre la possibilité de gérer des dépendances vis à vis des bibliothèques nécessaires au projet et d'automatiser certaines tâches. Spring boot nous donne aussi la possibilité de générer notre projet en war ou en jar, on a opté pour un packaging en format jar car tous dont on a besoin pour l'exécuter est une JRE (java runtime environment) en plus ce jar généré contient un serveur tomcat embarqué donc le projet web peut être déployé au sein de ce tomcat embarqué. Ensuite vient la partie du choix des dépendances. Les dépendances dont on avait besoin dans notre projet sont les suivantes :

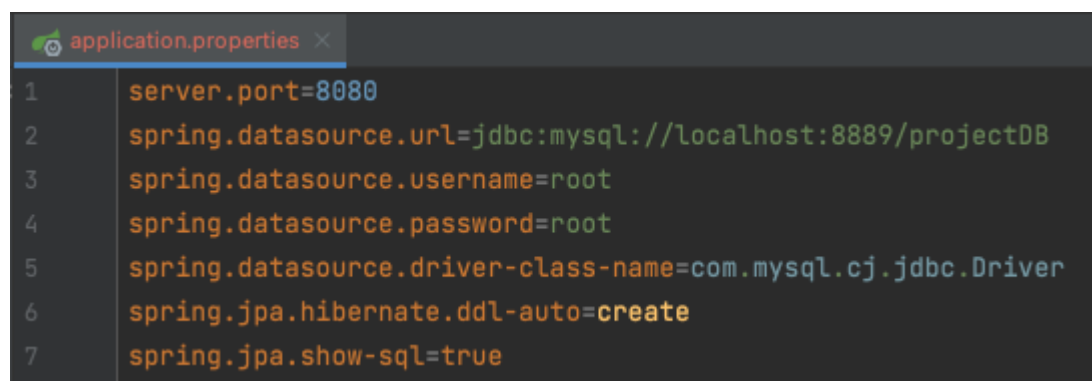
- Spring Web : qui permet de construire des applications web Restful en utilisant Spring mvc.
- Spring Data JPA : qui nous simplifie l'interaction avec le système de stockage de données, qui est dans notre cas une base de données relationnelle et nous évite ainsi de coder l'accès à ce système.

- MySQL Driver : permet de se connecter à la base de données.
- Lombok : permet d'obtenir un code plus lisible et nous libère de la longueur de l'écriture des méthodes que l'on retrouve communément dans les classes objets et métier du projet.
- Spring Boot DevTools : permet d'actualiser automatiquement en cas de modification du code.

Après avoir fini cette configuration qui prend en général très peu de temps grâce à Spring boot, on a notre environnement de développement qui est prêt et on peut entamer la partie développement pure.

Pour commencer la partie développement notre référence est l'architecture de la partie Back-end. On a tout d'abord commencé par organiser la structure de notre projet c'est à dire que pour chaque couche on a créé un package qui contiendra les classes et interfaces qui lui sont associés afin de faciliter le repérage et cela nous a servi comme convention qui sera suivie par tous les membres du groupe afin d'éviter des confusions par la suite.

On commencera d'abord par modifier le fichier "**application.properties**" de la façon suivante pour paramétrer le comportement par défaut de l'application; choisir un numéro de port pour le serveur, configuration de la base de données...

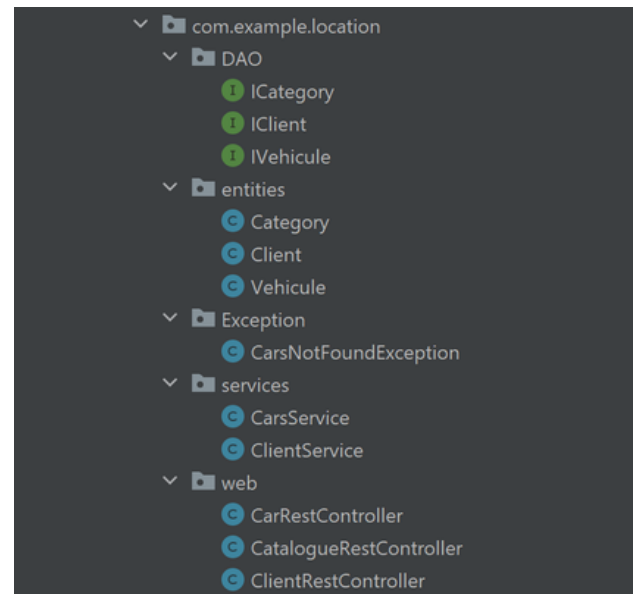
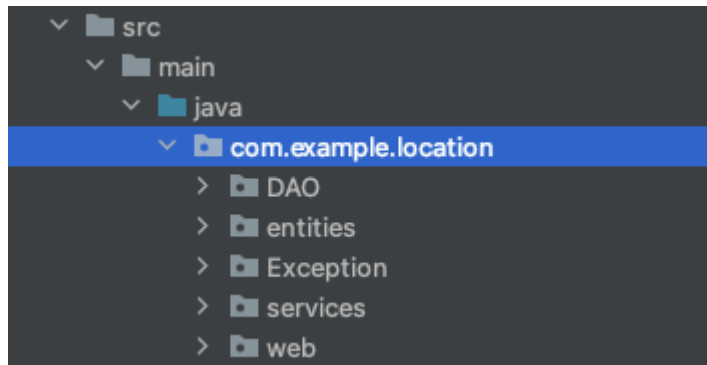


```
1 server.port=8080
2 spring.datasource.url=jdbc:mysql://localhost:8889/projectDB
3 spring.datasource.username=root
4 spring.datasource.password=root
5 spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
6 spring.jpa.hibernate.ddl-auto=create
7 spring.jpa.show-sql=true
```

On commence par indiquer le numéro de port dans lequel notre application va tourner, ici le port 8080, ensuite on a ajouté les configurations relatives à la base de données et la connexion au SGBD en lui renseignant l'url, le username et le mot de passe de la base de données, dans la sixième ligne on lui indique de supprimer à chaque fois que l'application est lancée les anciennes données et en recréer des nouveaux, en utilisant la commande **create**.

Cette simple configuration en quelques lignes nous permet de nous connecter à la base de données et effectuer les traitements qu'on souhaite sur nos données stockées dans celle-ci sans coder ni l'accès au SGBD ou sa configuration, tout ceci grâce à Spring boot.

On passe ensuite à la structuration de notre projet qui a pris la forme suivante:



Chaque package contient les classes qui appartiennent à une couche de l'architecture présentée en haut, en plus d'un package réservé à la gestion des erreurs qui contient la classe qui s'occupe de la gestion des erreurs. Dans la suite, on donnera les détails techniques concernant chacun des packages:

- **Package Entities** : contient les entités de notre application, une entité est simplement une instance d'une classe qui sera persistante, c'est à dire que l'on pourra sauvegarder/charger dans/depuis une base de données relationnel, dans notre cas on a trois entités ; **Client** qui fait référence à l'utilisateur de l'application, **Vehicule** qui désigne les voitures qui seront mises en location dans notre application et **Catégorie** qui représente la catégorie des voitures. Chacune de ces classes a ses propres attributs, pour dire à Spring que notre classe est une entité, JPA (*l'api de persistance de java*) nous met à disposition un ensemble d'annotations dont celle qui nous intéresse actuellement est l'annotation **@Entity** qu'on ajoute sur la classe pour dire à Spring que notre classe est bien une entité. De plus, notre entité JPA doit disposer d'un ou plusieurs attributs définissant un identifiant grâce à l'annotation **@Id** cet identifiant correspondra à la clé primaire dans la table associé, on ajoute également l'annotation **@GeneratedValue** qui indique la stratégie à appliquer pour la génération de la clé lors de l'insertion d'une entité en base de données. Les valeurs possibles sont données par

l'énumération **GenerationType** dans nos classes on l'a mise égale à *IDENTITY* pour faire une auto incrémentation de l'identifiant.

On a également ajouté des annotations de Lombok sur la classe qui sont les suivantes :

@NoArgsConstructor : génère le constructeur sans arguments.

@AllArgsConstructor : génère le constructeur avec tous les arguments.

@Getter : génère tous les getters sur les champs.

@Setter : génère tous les setters sur les champs.

@ToString : génère ToString sur tous les champs.

Ainsi en utilisant ces simples annotations dans toutes les classes du package on a un gain de temps considérable en plus de la réduction du nombre de lignes de code.

En programmation orientée objet il existe plusieurs relations entre objets (*héritage...*). Or pour modéliser ces relations entre des entités qu'on persiste dans des bases de données relationnelles, on a recours à des ORM (*mapping relationnel objet*) , une des fonctionnalités majeur de ces ORM est de gérer les relations entre les objets comme des relations entre tables dans un modèle de base de données relationnelle. JPA définit des modèles de relations qui peuvent être déclarés par des annotations. Dans notre cas puisqu'il s'agit d'un site de location de voitures, l'utilisateur peut disposer d'une seule voiture comme il peut avoir plusieurs, la relation dans ce cas entre l'entité **Vehicule** qui désigne la voiture et **Client** qui désigne son propriétaire est une relation de plusieurs (voitures) à un (client), de même une catégorie de véhicules peut contenir plusieurs voitures ce qui sera représenté dans ce cas par une relation plusieurs (voitures) à un (une catégorie).

On se retrouve donc dans notre projet avec la structure suivante dans la classe **Vehicule** :

```
@ManyToOne @JoinColumn(name= "idClient", referencedColumnName="idClient", nullable = true)
private Client client;
@ManyToOne @JoinColumn(name= "idCategory", nullable = true)
private Category category;
```

les annotations **@ManyToOne** au dessus des attributs indique que la table **Vehicule** contient deux colonnes qui sont des clés étrangères contenant respectivement la clé du client qui la possède et de la catégorie à laquelle elle appartient par défaut JPA s'attend à ce que ces deux colonnes se nomment respectivement **CLIENT_ID** et

CATEGORY_ID mais on peut changer le nom de ces colonnes grâce à l'annotation **@JoinColumn** comme on a fait dans notre cas et ainsi le nom de la colonne prend la valeur qu'on renseigne à l'attribut name dans les paramètres, ce qui est le cas en regardant la table Vehicule, on remarque que les deux colonnes se sont ajoutés et ont les noms qu'on leur a attribué.

	id_category	id_client
L	1	1

Extrait de la table Vehicule

Une fois l'entrée créée nous devons implémenter le code qui va déclencher les actions afin de communiquer avec la base de données, ce code se servira évidemment de nos classes entités qu'on vient d'aborder. Le principe est que ce code va effectuer des requêtes à la base de données, heureusement grâce à Spring il existe un outil puissant qui nous évitera de rédiger ce code, on va passer par une interface en utilisant un outil de Spring qui est **Spring Data JPA**, en effet ce dernier nous permet d'exécuter les requêtes SQL sans avoir besoin de les écrire, on a rassemblé ces interfaces dans le package DAO.

- **Package DAO** : Pour intégrer **Spring data JPA** dans notre projet, il existe une interface **JpaRepository<T,ID>** qui hérite directement de l'interface **CrudRepository<T,ID>** qui fournit un ensemble de méthodes pour interagir avec une base de données, ainsi il suffit just de créer des interfaces qui héritent de l'une des deux interfaces qu'on vient de présenter en ajoutant au dessus de l'interface l'annotation **@Repository** qui est une annotation Spring pour indiquer que la classe est un **Bean** et que son rôle est de communiquer avec une source de données en l'occurrence la base de données, et remplacer le T par le nom de la classe qu'on souhaite manipuler et le ID par le type de l'id de cette même classe. En plus des nombreuses méthodes proposées par ces interfaces dont on hérite, on peut créer n'importe quel méthode qu'on souhaite sans devoir l'implémenter, grâce à **Spring Data** qui utilise une convention de nom pour générer automatiquement le code sous-jacent et exécuter la requête, la requête est déduite de la signature de la méthode on parle de **query methods**.

Après avoir implémenté la couche DAO ainsi que la couche Model ou Entities qui contient nos entités on passera à l'implémentation de la couche service qui sera dédiée au métier. C'est-à-dire appliquer des traitements dictés par les

règles fonctionnelles de l'application, elle est également un pont entre la couche controller et repository, de ce fait nous avons créé le package Service qui contient tous les services de notre application:

- Package Services:

```
@Service
public class ClientService {

    private final IClient iClient;

    @Autowired
    public ClientService(IClient iClient) { this.iClient = iClient; }

    public Client addUser(Client user) { return iClient.save(user); }
    public Client updateUser(Client user) { return iClient.save(user); }
    public List<Client> findAllUsers() { return iClient.findAll(); }
```

Prenons l'exemple de ce service qui va implémenter la logique métier qui concerne le client. Au dessus de la classe on remarque l'annotation **@Service** similaire à l'annotation **@Repository** vu au dessus, son rôle est le même mais son nom a une valeur sémantique pour ceux qui lisent notre code, on sait qu'il s'agit d'un service.

On a d'abord commencé par injecter l'interface qui se charge de la persistance du client dans le constructeur en ajoutant l'annotation **@Autowired** sur le constructeur, cette annotation va permettre d'activer l'injection automatique des dépendances plutôt que de configurer manuellement les beans d'un contexte d'application dans un fichier XML. Ensuite, on a implémenté les méthodes en utilisant l'interface et les méthodes qu'elle nous met à disposition.

Nous arrivons maintenant à la dernière étape du développement de la partie Back-end, il reste à implémenter le contrôleur, en effet, comme on est entrain de développer une **API** c'est à dire une application qui va communiquer avec d'autres applications, il faut fournir aux autres applications qui voudraient communiquer avec nous un moyen de le faire, il s'agit donc de créer des **endpoints**, un endpoint est associé à une URL, lorsqu'on appelle cette URL on reçoit une réponse et cet échange s'effectue via **HTTP**, on va suivre le modèle **REST** notre API sera donc ce qu'on appelle une **API Rest**. Encore une fois grâce à Spring Boot on pourra créer nos endpoints c'est-à-dire notre contrôleur très facilement. On a réunis tous les contrôleurs de notre application dans le package web :

- **Package web** : notre contrôleur est une classe java annotée **@RestController** cette annotation permettra d'abord d'indiquer à Spring que cette classe est un **bean**, elle lui indique aussi d'insérer le retour de la méthode au format **JSON** dans le corps de la réponse **HTTP** pour permettre aux applications qui vont communiquer avec notre API d'accéder au résultat de leur requête en parsant la réponse **HTTP**. On prendra l'exemple du contrôleur associé aux utilisateurs de notre application. Après avoir annoté la classe on a injecté par la suite une instance du service **ClientService** cela permettra d'appeler les méthodes pour communiquer avec la base de données, et finalement on a créé les méthodes **GET, POST, PUT, DELETE** qui sont des méthodes comme toute les méthodes en java sauf qu'on leurs ajoute au dessus des annotations **@GetMapping** pour les méthodes de type **GET**, **@PostMapping** pour les méthodes de type **POST**, **@DeleteMapping** et **@PutMapping** pour des méthodes de types **PUT** et **DELETE** chaque méthode annotée devient donc un end-point ces endpoints vont permettre aux requêtes HTTP de type **GET, POST, PUT, DELETE** à l'url qui est indiqué au milieu des annotations de ces méthodes d'exécuter le code de celles-ci. Voici les méthodes de la classe **ClientController** dans notre cas :

```
@GetMapping("/findClients")
public ResponseEntity<List<Client>> getAllUsers(){
    return new ResponseEntity<>((List<Client>) clientService.findAllUsers(), HttpStatus.OK);
}

@PostMapping("/addClient")
public ResponseEntity<Client> addUsers(@RequestBody Client client){
    return new ResponseEntity<>(clientService.addUser(client), HttpStatus.CREATED);
}

@PutMapping("/updateUser")
public ResponseEntity<Client> updateUser(@RequestBody Client user){
    return new ResponseEntity<>(clientService.updateUser(user), HttpStatus.CREATED);
}

@DeleteMapping("/deleteClient/{id}")
public ResponseEntity<?> deleteClient(@PathVariable("id") Long id){
    clientService.deleteClientById(id);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Pour exécuter la méthode **getAllUsers()** on doit envoyer une requête **HTTP** de type **GET** à l'URL **/findClients** et qui appelle la méthode

findAllUsers() du service **ClientService**, ce dernier appellera la méthode **findAll()** du repository et nous obtiendrons ainsi tous les utilisateurs stockés dans la base de données en format JSON. Comme il s'agit d'une méthode **GET** on peut envoyer la requête depuis n'importe quel moteur de recherche, on obtiendra le résultat désiré. Or pour les trois autres méthodes, puisqu'on doit fournir un objet dans le body (le corps) de la requête on est obligé de passer par un intermédiaire, dans notre cas on utilise **Postman** qui est une application permettant de tester notre API en exécutant des appels HTTP directement depuis une interface graphique, il suffit de renseigner l'URL et la méthode **HTTP**. Les principales sont : (**GET,POST,PUT,DELETE**). Finalement pour éviter les erreurs des permissions d'accès, car dans de nombreux cas, l'hôte qui sert le front-end sera différent de l'hôte qui sert les données (back-end). On a ajouté la notation **@CrossOrigins("url")** qui permet le partage de ressources d'origine croisée (**CORS**) ce qui permet la communication entre domaines, on lui donne comme argument l'url de l'application avec qui elle va communiquer ou on met simplement une étoile * à la place de l'URL pour donner l'autorisation à n'importe quelle application de faire des appels au back-end sans qu'elle soit interrompue.

Finalement, on a ajouté un dernier package qui contient les classes relatives à la gestion des erreurs, ces classes héritent de la classe **RuntimeException** qui est la superclass de toutes les exceptions qu'on peut rencontrer en développant en java, elle est pas primordiale dans le développement de notre application mais elle nous permet d'identifier facilement nos erreurs au lieu de se perdre dans des très longs lignes d'erreurs. À l'intérieur de la classe "**CarsNotFoundException**" on a créé notre méthode qui va nous retourner un message d'erreur personnalisé si on ne trouve pas une voiture précise dans la base de données, c'est la méthode **CarsNotFoundException(String message)**.

Après avoir implémenté tous ces packages de notre application, la partie Back-end est quasiment complète et on peut se lancer dans le développement de l'application du côté client c'est à dire le Front-end de notre application et revenir effectuer des modifications si nécessaire sur notre partie Back-end.

Pour récapituler on les couches suivantes et leurs fonction :

Couche	Fonction
Controller	Réceptionner la requête et fournir la réponse
Service	Exécuter des traitements métiers
repository	Communiquer avec la source de données
model	Contient les objets métiers, les entités

En ce qui concerne les annotations des méthodes HTTP on a utilisé les annotations suivantes

Annotation	HTTP méthode	Objectif
@GetMapping	GET	Lecture des données
@PostMapping	POST	Envoie de données, création d'un nouvel élément
@PutMapping	PUT	Remplacement complet de l'élément envoyé
@DeleteMapping	DELETE	Pour la suppression de l'élément envoyé

Enfin, en ce qui concerne la configuration de la base de données, on a travaillé avec l'environnement du serveur local **MAMP**, qui nous met à disposition l'utilitaire *phpmyadmin* qui permet de gérer les bases de données et tables SQL de type MySQL et nous fournit une interface nous permettant de créer des bases de données et manipuler les données et ce qui était très utile surtout lors des test de notre API avec **Postman** pour évaluer si nos méthodes HTTP fonctionnent ou pas.

On parlera dans la suite des étapes du développement de la deuxième phase de l'application, le développement de la partie Front end.

Partie Front-End

Pour créer la partie Frontend de notre application web (proche rendue client), on a intégré un Framework Javascript. Parmi les différents Frameworks utilisés(React, Vue...), on a décidé de travailler avec Angular qui est aujourd'hui le plus utilisé pour les larges projets d'entreprise.

Angular

Tout d'abord, Angular est un Framework côté client créé par Misko Hevery, et le nom « Angular » vient du fait que les balises HTML (< >) sont angulaires.

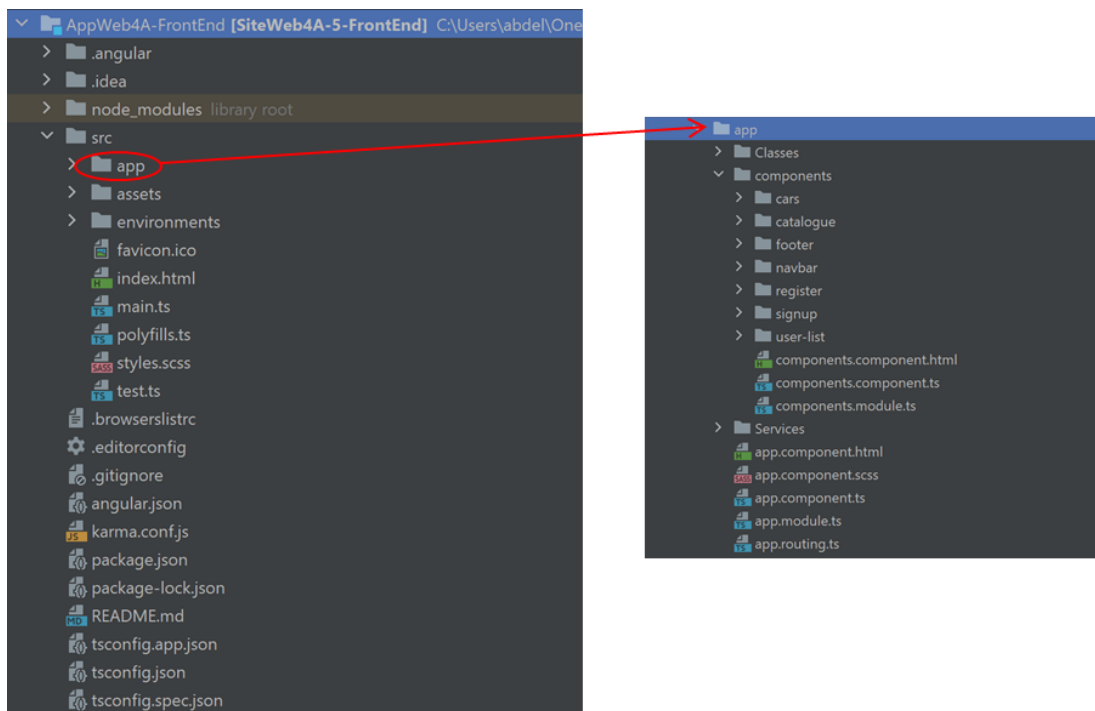
Parmi les avantages qui ont dirigés notre choix vers Angular, on trouve :

- Sa capacité à développer de larges applications Web en promouvant une architecture facilement maintenable et lisible pour un travail en équipe optimal.
- Sa performance de réaliser, à partir de la même base de code, des applications utilisables sur mobiles et tablettes, appelées PWA.

La première version d'Angular est AngularJS, cette ancienne version est basée sur une architecture MVC et ses applications sont écrites en JavaScript. La réécriture de AngularJS a été appelée "Angular" sans le "JS", en référence aux versions 2 et plus(qui est aujourd'hui à la version 13). Selon ses créateurs, Angular est réimaginé AngularJS pour le Web moderne, en prenant en compte tout ce qui a été appris avant. De plus, les applications de Angular sont écrites en TypeScript qui est transpilé en JavaScript avant d'être exécuté, ce qui permet une compatibilité avec tous les navigateurs, même ceux qui ne sont pas à jour des dernières versions.

Ce langage typeScript a pour objectif d'améliorer et de sécuriser la production de code JavaScript, ainsi qu'il permet aux développeurs habitués aux langages typés (.NET, Java et C# par exemple) d'être plus à l'aise que s'ils avaient à développer en pur JavaScript.

Structure de notre projet Angular



Parmi les différents fichiers et dossiers qu'on trouve dans la structure de notre projet Angular, on définit les suivants :

angular.json :

C'est un fichier essentiel qui possède de nombreuses propriétés et permet de configurer l'espace de travail. On peut configurer par exemple le nom et l'emplacement du dossier où sera buildé notre projet.

package.json :

Ce fichier permet de décrire les dépendances d'un projet. En effet, on trouve une liste de toutes les dépendances de notre application. On cite par exemple, **@angular/router** qui permet de gérer les routes, et **@angular/forms** qui gère les formulaires. De plus, on trouve dans ce fichier un certain nombre de scripts prédéfinis comme **start** qui permet de démarrer l'application

package-lock.json :

Ce fichier contient un arbre exact des dépendances et de leurs propres dépendances, permettant de réinstaller exactement les mêmes versions.

karma.conf.js :

On peut réaliser dans ce fichier les tests unitaires qui seront utilisés lors de l'exécution de la commande suivante : **ng test**.

editorconfig :

C'est une configuration simple pour l'éditeur pour assurer que tous ceux qui utilisent le projet auront la même configuration de base.

gitignore :

C'est un fichier git permet d'assurer que les fichiers générés automatiquement, comme les dépendances (/node_modules), ne seront pas prises en compte.

Le dossier node_modules :

Il contient toutes les dépendances installées par **npm** (Node Package Manager). Comme vu précédemment, ces dépendances correspondent à celles déclarées dans **package.json**.

Dans le dossier **src**, on trouve :

styles.scss :

Un fichier qui contient les styles globaux qui affectent tout l'ensemble de notre application

assets :

C'est un dossier qui englobe toutes les images qui seront utiles dans notre projet Angular.

environments :

Ce dossier contient un fichier pour chacun des environnements de destination (généralement développement ou production), chacun exportant des variables de configuration simples à utiliser dans notre projet.

app :

Une grande partie du code de notre application Angular est située dans ce dossier. On trouve :

- **app.component :**

C'est le composant racine du projet. Comme chaque composant d'Angular, il contient :

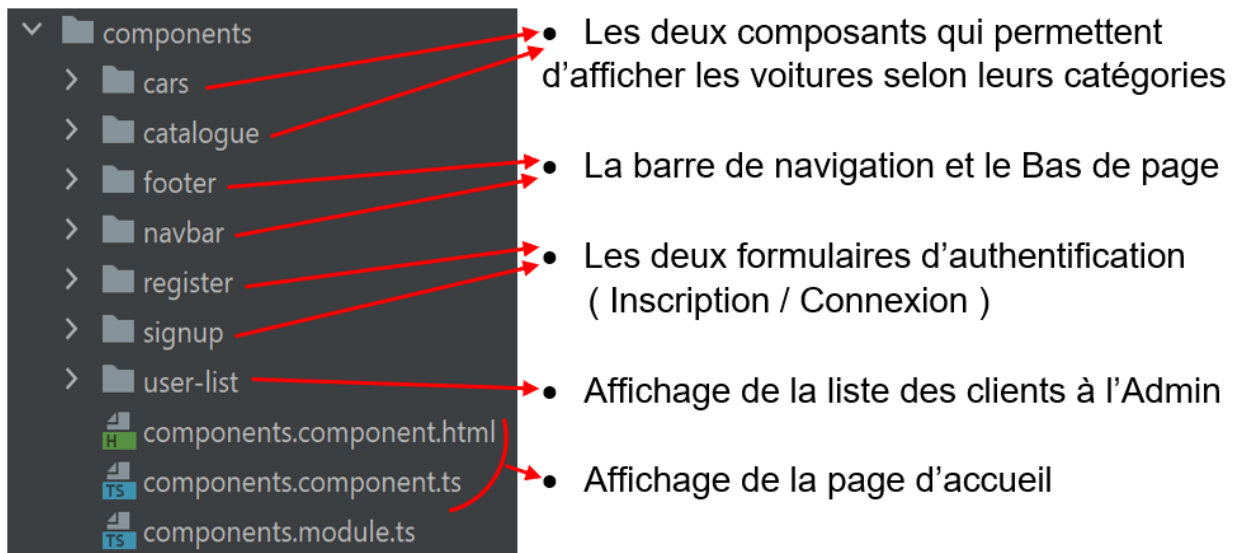
- Un Template contenant l'interface utilisateur en HTML.
- Une feuille de style (dans notre cas : app.component.**scss**).
- Une Classe contenant le code associé à la vue, des propriétés et méthodes logiques qui seront utilisées dans la vue.

Le composant peut aussi contenir un test unitaire (.spec.ts).

- **app.module.ts :**

C'est le module racine qui indique à Angular comment assembler l'application. À l'initialisation du projet, il déclare uniquement l'AppComponent. Mais après, il y aura évidemment plus de composants déclarés.

Dans notre projet Angular, on a rassemblé tous les composants utilisés dans le dossier 'components'



- **app.routing.ts :**

Ce fichier sert à déclarer nos différentes routes(Home, Register...) dans le but de gérer l'affichage via l'URL dans le navigateur.

Le routeur angulaire permet la navigation entre les différents composants lorsque les utilisateurs exécutent des tâches d'application.

Ce système de routing est un service facultatif qui présente une vue de composant particulier pour une URL donnée.

Il y a d'autres fichiers importants dans notre projet Angular qu'on va aborder par la suite, comme le fichier **main.ts** qui est le point d'entrée principal de notre application, et aussi **index.html**, la page HTML principale.

Voyons maintenant brièvement comment fonctionne Angular et comprenons les différents outils que nous avons intégrés à notre projet.

Mode de fonctionnement de Angular

Tout d'abord, Angular est un Framework permettant de construire des Single Page Applications (SPA). De ce fait, une grande majorité d'applications ont adopté cette architecture, on cite par exemple Gmail et Dropbox

Une SPA est une application qui contient une seule page HTML (**index.html**) récupère du serveur. Ce type d'application fonctionne dans un navigateur sans que l'utilisateur n'ait besoin de recharger la page, ce qui permet d'augmenter la rapidité et la fluidité d'un site internet.

Le scénario de chargement commence par le fichier **index.html** qui est la première page lancée par un navigateur. Ensuite, Angular exécute le module principal **main.ts** qui déclenche par la suite le module **AppModule**. Finalement, à l'intérieur de **app.module.ts**, Angular cherche le premier composant à déclencher (dans notre cas c'est **AppComponent**).

Les outils utilisés dans notre projet Angular

Lors de la création de notre projet Angular, nous avons choisi d'utiliser **SCSS** au lieu de CSS. Ce langage sert à apporter un lot de fonctionnalités pour faciliter la vie et alléger le code. L'écriture avec SCSS est mieux organisée et moins répétitive ce qui rend le travail sur les fichiers plus agréable.

Comme le SCSS est compatible avec toutes les versions de CSS. On a intégré à notre projet une bibliothèque css très connue : **Bootstrap** (version5). Ce framework nous a aidé à développer notre application avec un design responsive qui s'adapte à tout type d'écran.

On a également utilisé une Template de Angular qui est « Paper Kit 2 PRO ». Ce module combine plusieurs composants réactifs et propose des exemples d'utilisation, il nous a fourni les éléments essentiels pour coder un projet Web.

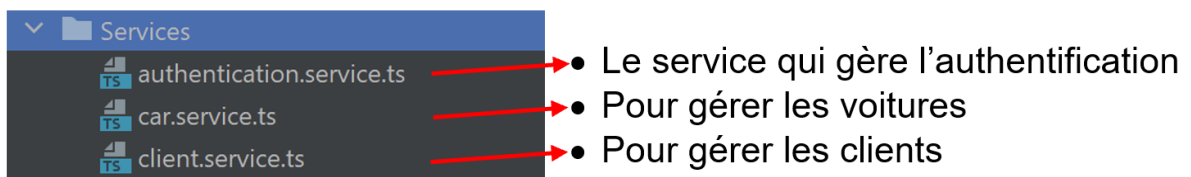
On parlera dans la suite sur les services que nous avons utilisé dans notre projet et leur utilité pour assurer la liaison avec la partie Backend.

Les services

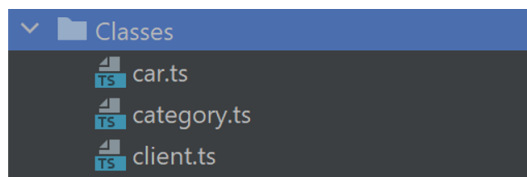
Un service est une classe TypeScript composée d'attributs et de méthodes, dont l'instanciation est gérée par Angular. Généralement, les composants se limitent à l'affichage et à la gestion des événements utilisateurs dans la vue du composant. Les autres opérations comme l'exécution des traitements en local ou en Backend sont attribués aux services.

L'utilisation d'un service se fait via le principe de l'injection des dépendances. On peut injecter un service dans un composant ou dans un autre service.

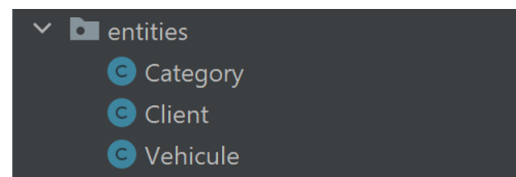
Dans notre projet, on a utilisé trois services :



De plus, pour bien gérer les traitements, on a créé des classes similaires aux celles de la partie Backend



FrontEnd



BackEnd

Communication avec la partie BackEnd

Généralement, la liaison entre les deux parties : BackEnd et FrontEnd, est assurée par les services en envoyant des requêtes HTTP.

Prenons l'exemple du service de clients, pour récupérer les clients de la base de données, nous avons envoyé une requête Get vers le Backend.

```
//on inject 'http' pour envoyer des requetes HTTP au Backend
public getClients(): Observable<Client[]> { //returner un objet de type Observable
  return this.http.get<Client[]>({ url: this.host+"/findClients"});
}
```

Dans notre cas, l'host est :

```
this.host = 'http://localhost:8080';
```