POLYTECH NICE SOPHIA

Programmation Web

ANTHONY BARNA

https://github.com/Anthony-Barna

Année 2021 / 2022

Rapport réalisé dans le cadre du double module : programmation web client / programmation web serveur.

L'équipe est composée de :

- Anthony Barna
- Hadrien Bonato--Pape
- Jérôme Froment
- Leo Burette

Dans ce projet, mes efforts ont essentiellement été concentrés pour implémenter la partie serveur.

Technologies choisies

React

React est un des frameworks web frontend les plus utilisés.

Nous l'avons choisi car :

- Complet et performant
- Développé et maintenu par Facebook principalement, open source
- Possède une grande communauté
- Est enseigné dans le module de programmation web client

NestJS

Nous avons choisi NestJS comme framework backend pour plusieurs raisons :

- C'est un des frameworks backend basé sur NodeJS les plus utilisés
- Le développement se fait en TypeScript
- Son CLI permet d'implémenter rapidement des modules
- Il embarque un environnement de développement et de test
- La documentation est claire, la communauté est grande
- Il embarque les outils de base nécessaires à un backend (injection de dépendances, client http…)
- Il peut se lier nativement à énormément de librairies (swagger, schedule, jwt…)

En somme, il permet de gagner beaucoup de temps sur des tâches de démarrage, fournit des outils de développement. Tout cela est implémenté en utilisant les avantages de TypeScript.

MongoDB

Ce type de base de données est, certes, imposé par le sujet. Cependant, l'utilisation d'une base NoSQL est pertinent et adapté ici, car le sujet demande principalement de traiter quelques milliers d'entités qui ne possèdent pas de contraintes entre elles.

Ainsi, pour ce qui est de filtrer, trier ou encore rechercher, ce type de base se révèle être extrêmement performant. Nous laissons de côté la perte de performances qui serait engendrée par la gestion de clés primaires, étrangères au autres contraintes d'intégrité des données implémentées par des moteurs de base SQL.

TypeORM

Un mapping object-relationnel fonctionnant parfaitement avec NestJS, performant et tirant partie de TypeScript. Nous avons trouvé sa prise en main plus rapide qu'avec Mongoose, et sa documentation plus claire.

Branching strategy

En ce qui concerne notre workflow Git, nous en avons implémenté un proche du Gitflow. Une feature ou un bug équivaut à une branche qui donnera suite à une pull request pour être mergée à une branche de développement, puis une branche de production.

Le projet est réparti en deux répertoires GitHub : frontend et backend.

Fonctionnalités implémentées

Récupération des données ~ 4h

Chaque jour à 3h30 du matin, une requête sa lance automatiquement, récupérant le dernier fichier à jour d'informations sur le prix de l'essence en France, depuis l'api du gouvernement.

Ensuite :

- Le zip récupéré est décompressé
- Le fichier xml contenu est parsé
- Un nouveau modèle de données est créé, des champs sont formattés
- Les informations sont persistées en base

Un import de données est aussi possible manuellement en uploadant un fichier xml.

Recherche de points de vente ~ 5h

Il est possible de rechercher des points de vente de carburant selon ces filtres :

- Nom du carburant
- Prix maximal du carburant
- Distance maximale par rapport à un point donné
- Point de vente sur route ou autoroute
- Limite de résultats

Une indexation des positions prenant en compte la courbure de la terre est réalisée à la fin de l'import de données. Il permet de trier de manière très efficace les points de vente par distance en fonction d'un point donné.

Statistiques sur les points de vente ~ 5h

Lors de l'import de données, des statistiques du prix moyen par carburant sont générées au niveau national, régional et départemental.

Un blocage a été rencontré sur la création d'un pipeline d'aggrégation MongoDB permettant de calculer les prix moyens par carburant sur des points de vente.

Dans un premier temps les points de ventes sont triés par code postal selon une expression régulière.

Ensuite des calculs de moyenne sont réalisés pour chaque carburant existant dans l'échantillon de points de ventes retenu.

Finalement, les résultats sont triés par nom de carburant.

Authentification par JWT ~ 6h

Une authentification par JSON Web Token (jwt) a été implémentée.

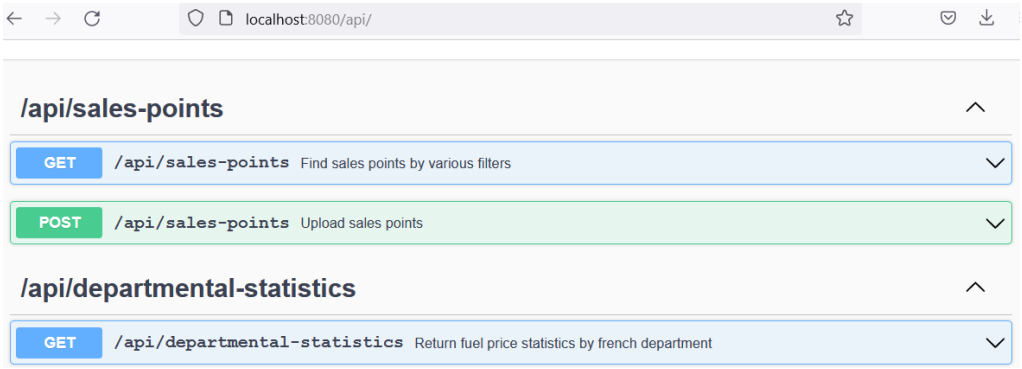
Lors de la création d'un compte utilisateur, son mot de passe est encrypté avec la fonction de hachage « bcrypt » avec 10 rounds, puis l'utilisateur est persisté en base de données.

Lors du login, si les identifiants sont valides, l'API retourne un jwt. La validité de ce token est fixée à une heure. Ce dernier peut être décrypté et contient le payload de l'utilisateur connecté. Renseigner ce token dans un header permet de s'authentifier et accéder aux endpoints protégés (c'est-à-dire tous sauf le login et le création de compte utilisateur).

Documentation via Swagger ~ 2h

En utilisant la librairie « @nestjs/swagger », des annotations ont été posées sur les différents modèles et contrôleurs de manière à documenter notre API.

Ainsi, lorsque le serveur est lancé, il est possible d'accéder à cette documentation.



Participation au frontend ~ 4h

Mon travail lié au frontend s'est concentré sur l'interfaçage entre le client React et notre API, la définition de cette API REST, et la gestion des JWT dans l'envoie de requêtes.

Composant optimal

```typescript
@Entity()
@TableInheritance( options: { column: { type: "varchar", name: "type" } })
export abstract class Statistic {

  public static readonly DEPARTMENTAL_TYPE = "departmental";
  public static readonly REGIONAL_TYPE = "regional";
  public static readonly NATIONAL_TYPE = "national";

  @ObjectIdColumn()
  id: ObjectID;

  @ApiProperty()
  @Column()
  type: string
```

```typescript
@ChildEntity()
export class DepartmentalStatistic extends Statistic {

    constructor() {
        super(Statistic.DEPARTMENTAL_TYPE);
    }

}
```

Ici, un patron de conception « union » est implémenté, mettant en commun des champs pour les statistiques. La valeur discréminante est donc le type de la statistique. Cela permet de manipuler un modèle objet fortement typé sur la couche typescript, tout en persistant ces entités dans la même collection au niveau de la couche base de données.

Composant non optimal

Nous avons pris le temps d'optimiser les différents modules dans notre API. Nous aurions pu cependant tirer plus profit du système d'injection de dépendances qu'offre NestJS et rendre injectable des classes utilitaires au lieu d'en créer des instances manuellement.

```typescript
@Cron( cronTime: '0 30 3 * * *')
async persistDistantSalesPoints(): Promise<void> {
  const unzipUtil: UnzipUtil = new UnzipUtil();
  await this.persistSalesPoints(await unzipUtil.getUncompressedGasPriceString());
}
```

Dans ce cas précis, UnzipUtil aurait pu être configuré en tant que composant injectable puis injecté via le constructeur du service dans lequel il est instancié.