VéAPI

I. Introduction

1. Rappel du sujet

Dans le projet annuel il nous a été demandé de faire une API, dans notre cas cette API n'est en rap port avec l'application java de gestion de projet agile.

Nous avons voulu faire une API uniquement en lien avec le projet Vécolo.

Le cœur de ce projet et d'interagir avec l'application Angular pour lui transmettre des données.

Cette application fait aussi bien d'autre chose, elle a une interface en ligne de commande pour que l'on puisse écrire des scripts que les développeurs puissent lancer. Dans notre cas, nous nous en sommes servis pour faire des « seeds » et remplir la base de données avec des données des tests.

Elle exécute des tâches planifiées pour les factures, et le nettoyage de la base de données.

Le sujet était très libre quant à implémenter cette API en Node JS.

La véritable consigne était celle que l'on se donne. Elle dépend du sujet que l'on avait choisi. Une API c'est le point d'accès à la base de données seul, elle peut à la fois servir pour diverse application frontend mais c'est aussi l'occasion de centralisé son code métier.

II. Focus sur l'application

1. Fonctionnalités

Cette application a pour but principal de permettre à l'application Angular d'interagir avec la base de données et les différents services de Vécolo.

Actuellement, cette application est utilisée par l'application Angular et les Stations autonomes, mais dans notre idée, il est clair qu'elle joue un rôle central sur le projet, elle représente le

domaine d'application.

L'application a deux portes d'entrée :

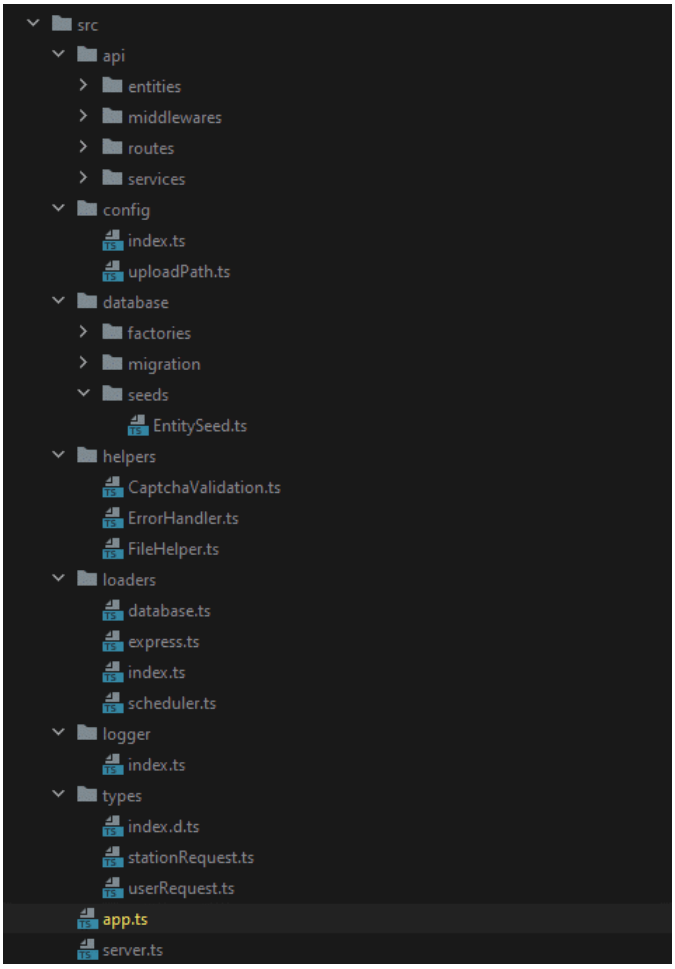
- Via l'API web (Application Angular & Stations autonomes)
- Via la ligne de commande (Seed de la BDD & migrations)

L'application, quant à elle, communique directement avec la base de données.

2. Architecture du code

Nous avons adopté une architecture MVC pour ce projet.

- Le dossier API contient nos routes, nos services ainsi que nos classes d'entités
- Config se charge de récupérer les variables d'environnements utiles à notre application
- Database va contenir nos migrations ainsi que certaines fonctions utilitaires pour les seeds
- Helper contient quelques fonctions utilitaires comme la validation d'un Captcha, un FileHelper...
- Loader contient tout ce qui doit être chargé au démarrage du serveur (Démarrage express, connexion à la BDD, démarrage des routines, ...).
- Logger contient notre classe permettant d'avoir des los fichier partout dans notre application.



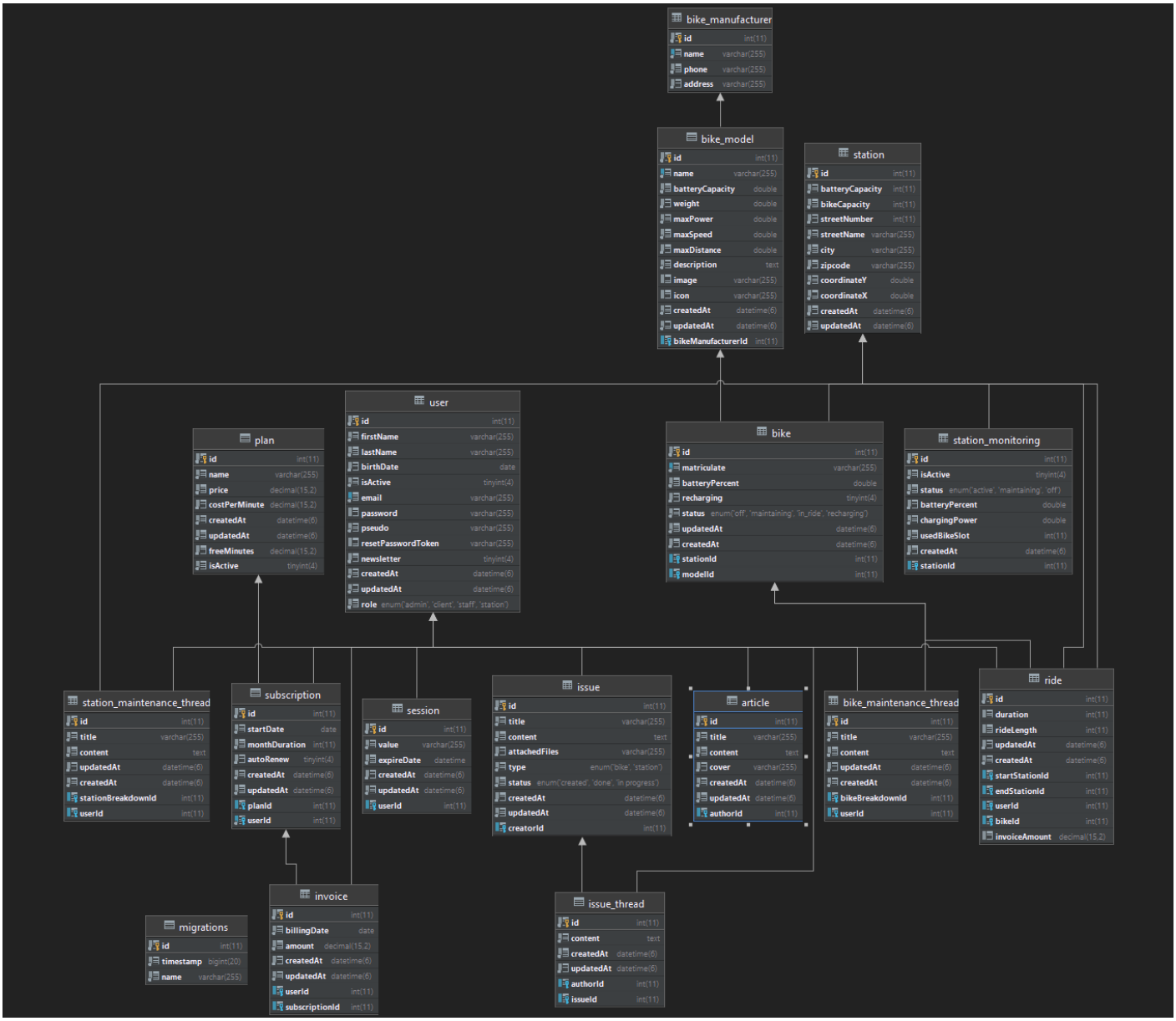
3. Schéma de base de données

Nous avons séparé la base de données de Vécolo de celle de l'outil de gestion de projet.

VéAPI est la seule application en lien direct avec la base de données.

Toutes les autres applications passent par VéAPI quand il s'agit de manipuler des données persistantes.

Dans notre implémentation actuelle, nous utilisons le SGBD MariaDB. La question s'est posée plusieurs fois d'utiliser PostgreSQL mais nous n'avons pas fait le changement.



III. Choix d'implémentations

1. Injection de dépendance

Ce projet fonctionne avec de l'injection de dépendance. Pour ce faire, nous avons utilisé TypeDI qui nous permet de l'implémenter avec des annotations Typescript.

L’injection de dépendance est au cœur des projets Vécolo c’est l’une des nouveautés que l’on essaye d’implémenter désormais. Dans l’univers de javascript, c’est quelque chose d’assez atypique.

Ne sachant pas à quoi nous attendre, nous avons pris le risque de nous aventurer vers l’inconnu et de mettre en place une architecture basée sur ce système, nous ne connaissions pas du tout la librairie contrairement à Spring pour java où nous avions quelques notions, ici la découverte a été totale.

C’est vrai que les systèmes ne fonctionnent pas exactement de la même manière et que cette implémentation est un petit peu plus « root », cela nous a permis d’en apprendre davantage sur le fonctionnement interne de l’injection de dépendance.

2. Typescript

Pour ce projet, nous utilisons Typescript et pas directement javascript, pour l'apport en termes de syntaxe via les types et les annotations. La question ne s’est pas vraiment posée ici, étant donné que si l’on veut utiliser Angular c’est une case quasi obligatoire alors autant aussi utiliser cette technologie en backend aussi.

3. TypeORM

Dans ce projet, nous avons utilisé un ORM, pour s’abstraire des spécificités de la base de données.

Encore une fois ici, on tire avantage des annotations Typescript pour décrire nos entités et nos tables.

On a utilisé les migrations via le système de typeORM et la détection de changement dans les entités.

4. Authentification

On a choisi JWT **(JsonWebToken)** pour mettre en place l'authentification ; et ce pour plusieurs raisons.

Premièrement, sa capacité à stocker les sessions sur le client et qui donc nous permet d'avoir des sessions coté client. Ce qui est nécessaire étant donné qu’actuellement l'API est en fait un Cluster de 4 instances régi par un Load Balancer.

Une autre spécificité dans notre utilisation de JWT est le fait que nos stations fassent des requêtes authentifiées à l'API. Pour que cela fonctionne, chaque station a son token qui est à durée de vie illimitée et qui lui permet de s’authentifier sur les routes pour le monitoring. Toutes les autres routes leur sont interdites.

Pour répondre à la problématique d'authentification d'utilisateur, on utilise les librairies : **jsonwebtokenet** et **express-jwt**.

5. Problématique des rôles

Pour répondre à la problématique des rôles et de l'accès au point d'entrée de l'API, nous avons utilisé des middlewares sur les routes pour filtrer les requêtes.

```
route.post(
    path: '/',
  isAuth,
  checkRole(Role.STAFF),
  paramsRules,
  async (req : Request<ParamsDictionary, any, any, ParsedQs, Record<string, any>>, res : Response<any, Record<string, any>>, next : NextFunction) => {
    const service = Container.get(defaultService);
    try {
      const entityResult = await service.create(req.body);
      return res.status( code: 201).json(entityResult);
    } catch (e) {
      return next(e);
    }
  }
);
```

```
export const getTokenFromHeader = (req: Request) => {
  if (
    (req.headers.authorization &&
      req.headers.authorization.split( separator: ' ')[0] === 'Token') ||
    (req.headers.authorization &&
      req.headers.authorization.split( separator: ' ')[0] === 'Bearer')
  ) {
    return req.headers.authorization.split( separator: ' ')[1];
  }
  return null;
};

const isAuth = jwt( options: {
  secret: config.jwtSecret, // The _secret_ to sign the JWTs
  requestProperty: 'token', // Use req.token to store the JWT
  getToken: getTokenFromHeader, // How to extract the JWT from the request
  algorithms: ['HS256'], // Use the HS256 (HMAC with SHA-256) algorithm
});
```

6. Informations privées

Pour ce projet, nous avons utilisé **dotenv** et des fichiers d'environnement **.env** dans lesquels nous stockons les informations qui ne doivent pas être partagées sur le dépôt public.

Cependant, nous avons un fichier **.env.example** pour indiquer à l'utilisateur comment remplir le fichier **.env**.

Les informations de configuration sont stockées dans un objet « config » qui nous permet de charger les données du fichier au démarrage dans une variable pour pouvoir derrière utiliser ces données partout dans l'application.

7. Lintage

Pour s'assurer une propreté optimale dans notre code, une configuration **EsLint** avec prettier a été installé pour ce projet pour formater le code de manière uniformisé.

Étant donné que le projet est en Typescript, nous utilisons aussi des extensions **EsLint** et **prettier** pour cet usage.

8. Validation des requêtes http

Pour gérer la vérification des données envoyées à l'API, on utilise Celebrate et Joi.

Celebrate encapsule Joi pour en faire un middleware.

On décrit un schéma de ce qui est attendu comme contenu dans la requête entrante avec ces outils et on laisse Celebrate effectuer le travail. C'est lui qui va bloquer les requêtes non conformes et renvoyer un message d'erreur.

9. Upload de fichier

Dans le projet actuellement, nous avons une fonctionnalité d'upload de fichier pour les images des modèles de vélos.

Étant donné qu'il y a 4 instances de l'API et que nous sommes sous docker, nous avons dû monter un volume partagé entre toutes les instances.

10. Logging

Pour les logs, nous avons aussi besoin d'un volume docker partagé entre les instances.

Nous utilisons la librairie winstonjs, qui nous permet de gérer jusqu'à 6 niveaux d'erreurs.

Les logs sont enregistrés dans deux fichiers. Un premier pour les logs de type erreur et un autre pour ceux du niveau informatif.

11. CRON

Parmi les éléments qui sont chargés par l'application, il y a des « schedulers », dans notre cas deux schedulers sont présents :

- Un est présent pour nettoyer de la base de données les enregistrements de monitorings générés des stations par Vemock.

- Un autre est présent au début de chaque mois pour créer de nouvelle facture pour les abonnées dont l'abonnement continue.

12. Seeds

Pour remplir la base de données, nous avons mis en place des scripts de « seeds ».

Ces scripts vont faire appel à des « Factory object » pour insérer des éléments dans la base de données.

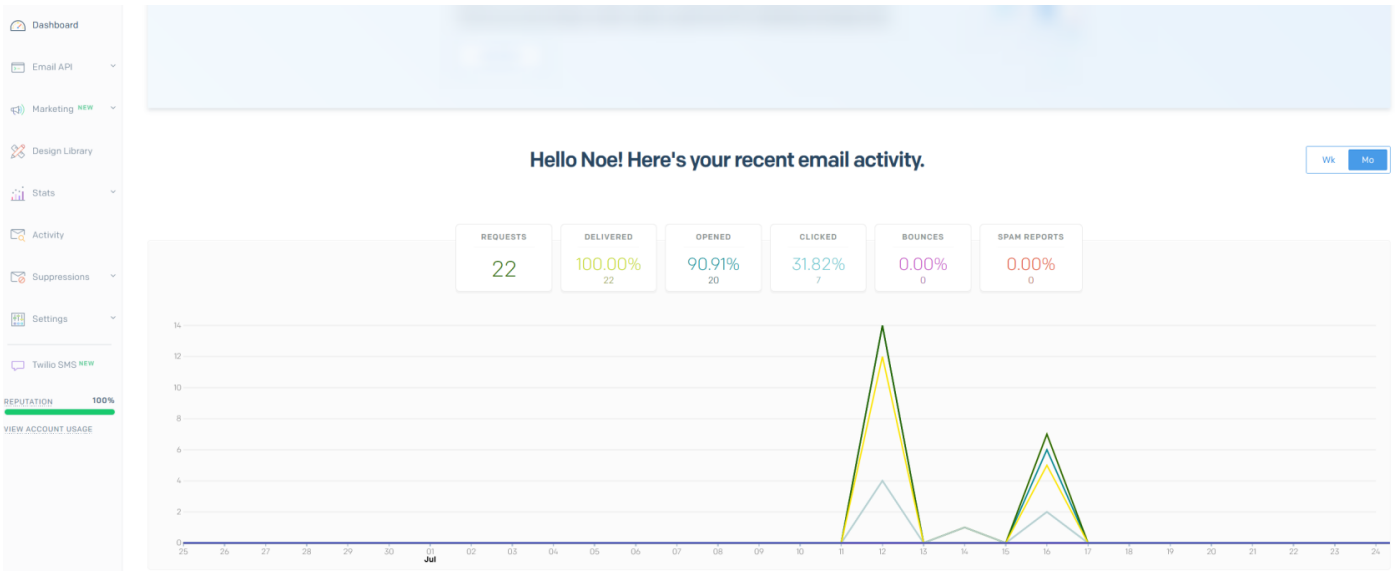
De fausses données sont générées à partir **de faker js**, l'interface CLI (qui permet de nous demander la quantité de donnée désirée a été faite avec **inquirer**.

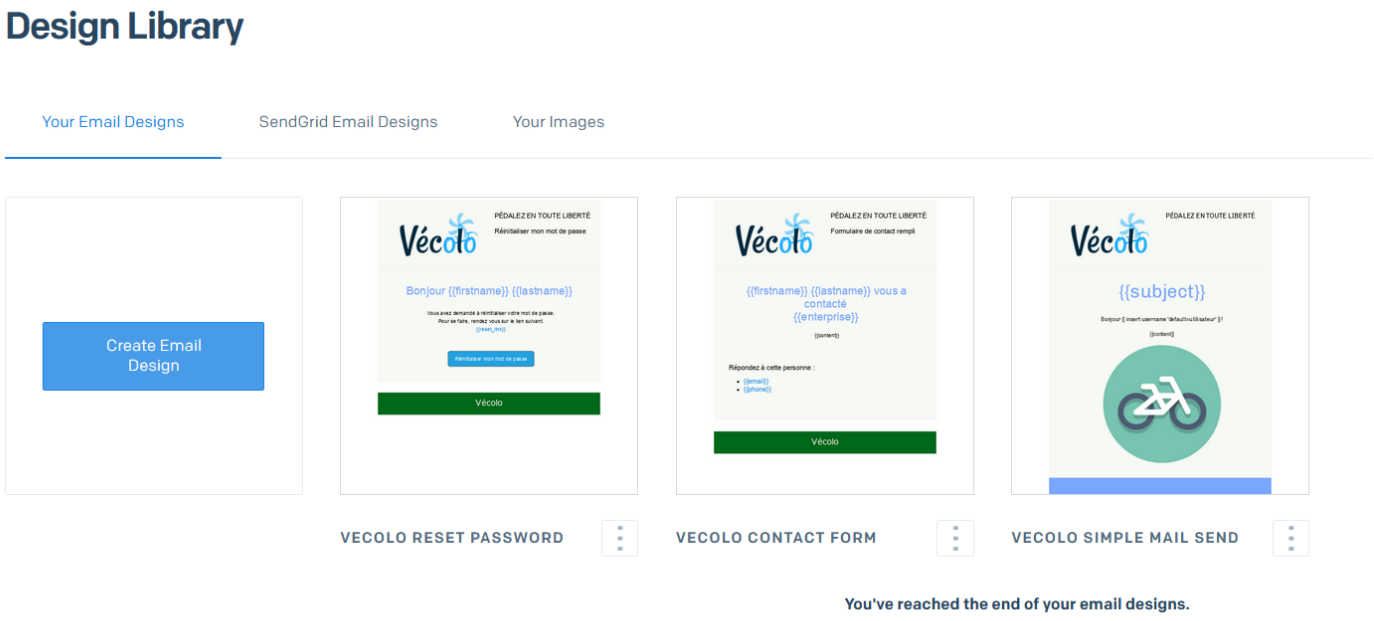
13. Mailing

Lorsque que nous devons envoyer des mails (newsletter, réinitialisation de mot de passe) depuis l'application Angular, c'est à l'API de s'en occuper, de traiter le contenu à envoyer, puis d'utiliser un service externe Sendgrid pour envoyer le mail.

Sendgrid

Sendgrid est un service d'envoi de mails, permettant de construire des modèles dynamiques pour ces derniers. C'est lui qui va se charger d'utiliser notre nom de domaine vécolo pour gérer les envoie de mails. Cela nous permet d'avoir une interface adaptée à ce besoin et d'alléger notre application API.





IV. Bilan du projet

1. Appreciation générale et organisation du projet

Ce projet a été marqué par trois périodes :

- Dans la première ; au moment où le projet n’est pas encore lancé et que l’on était à la genèse de cette API.

On a eu un travail de recherche sur l’architecture de cette application. C’est dans cette étape qu’on a pu retrouver des tentatives avec swagger, que l’on a changé l’interface de ligne de commande, que l’on a créé nos premières entités, nos premiers service auto injectés à partir de TypeDI.

- Dans un deuxième temps, au lancement de l’application Angular, nous avons laissé Clément s’occuper de cette application, c’était son souhait. Clément n’est pas allé jusqu’au bout du projet... Que ce soit un manque d’investissement ou de communication avec les autres membres et un manque d’intérêt pour le sujet ; le fait est que les choses n’ont pas été faites comme elles auraient dû et qu’il a fallu prendre un peu de temps pour pérenniser le travail déjà accompli.

- Dans un dernier temps ; l’application Angular avançant ; nous avions besoin de certains fonctionnalités plus spécifiques sur l’API. Il a fallu repenser certaines choses car ces points d’entrés fonctionnent différemment de la manière dont ils ont été pensés à la base.
  Nous avions aussi besoin de compléter l’API avec des requêtes auxquels nous n’avions pas pensés à l’origine.

2. Problèmes rencontrés

Au fur et à mesure du temps passé sur ce projet, nous avons décelé un souci dans l’architecture de l’application. Dans la structure que nous avons pensé au début, nous définition les routes dans le router et c'est lui qui gérait les entrées et sorties de l'application. Mais au fur et à mesure de l'avancement, et le projet grandissant, notamment sur la gestion des erreurs, nous nous sommes rendu compte que certaines choses n'étaient pas à leur place. Cela s’est vu notamment sur la question des erreurs et du code de retour http qui a parfois finit dans les services, et globalement il aurait été bien de découper le code pour le rendre plus claire. Cela nous aurait aussi aider pour nous y retrouver plus facilement sur les routes.

Un autre problème que nous avons rencontré est un problème avec les migrations typeORM. Les migrations ont besoin d'être exécutées depuis un environnement Typescript par le script de typeORM et sur Windows, nous avons eu des problèmes pour générer les migrations. Heureusement, c'est un problème connu de typeORM, malheureusement sans solution magique. Cependant avec de la recherche, nous avons réussi à trouver la ligne de commande qui nous permet de bien générer nos migrations sur Windows.

Avec typeORM, dans certain cas pour des requêtes un petit peu plus complexe, notamment les "groupe by", nous avons été contraints à utiliser le querybuilder et ça n'a pas forcément été « agréable » car on perdait le côté « utile » de l’ORM.

Avec le temps, nous nous sommes faits à l'utilisation et à la syntaxe, mais le dernier problème que l'on a rencontré est un problème d'organisation. En raison de l'ordre des rendus est des objectifs de chacun il était demandé de finir l'API avant d'avoir fini l'application frontend et donc de s'avancer sur les futurs besoins de l'application, ce qui a été un peu difficile et souvent raté.

3. Conclusion

Pour conclure nous sommes très fiers de l’architecture de cette application c’est un pari risqué car même si l’architecture n’a pas été conçu par nos soins de A à Z, elle a quand même beaucoup été customisé pour répondre à notre besoin, donc nous nous somme approprié le code par la même occasion. C’était encore plus risqué étant donné que l’écosystème javascript est vaste, changeant, et que les librairies n’ont pas toujours été pensées pour fonctionner ensemble. De plus, nous débutons dans l’univers du NodeJS.

Revision #2
Created 27 September 2022 14:44:05 by Noé Larrieu-Lacoste
Updated 27 September 2022 19:00:24 by Noé Larrieu-Lacoste