Clever-Party-Thrower, une application web hébergée via Kubernetes et Ansible

Louis De Wilde

Technologie de l'informatique 3TL2
Ephec : Av. du Ciseau 15, 1348 Ottignies-Louvain-la-Neuve
Novembre 2022

Table des matières

1 Introduction

Dans le monde d'aujourd'hui, les étudiants et les personnes en général sont souvent amenés à organiser des événements, tels que des soirées ou des fêtes entre amis. Cependant, gérer les aspects logistiques de ces événements peut s'avérer compliqué, notamment lorsqu'il s'agit de fixer une date, de répartir les coûts et de coordonner les participants. Afin de faciliter ce processus et d'améliorer l'expérience de l'organisation de fêtes, j'ai décidé de développer une application web nommée "Clever Party Thrower". Cette application permettra aux utilisateurs de gérer efficacement tous les aspects importants de l'organisation d'un événement, y compris les dépenses, les courses voir meme les covoiturages, la musique et bien plus encore.

Dans ce rapport, je présenterai les différentes étapes du développement de l'application Clever Party Thrower, en commençant par le choix du sujet et en détaillant le cahier des charges, l'analyse de la problématique, la conception et la réalisation du projet. J'aborderai également les aspects liés à la sécurité, la gestion des versions, les tests et la documentation du code.
Un accent particulier sera mis sur l'adoption de pratiques de développement modernes, telles que l'intégration continue (CI) et le déploiement continu (CD), afin d'assurer la qualité et la fiabilité du code tout au long du cycle de développement. De plus, je discuterai des choix d'hébergement pour l'application, en explorant les options d'auto-hébergement voir meme compatible avec le cloud.

Enfin, je terminerai en présentant une conclusion sur le travail accompli, les défis rencontrés et les perspectives d'amélioration pour le futur.

2 Choix du sujet

Dans le contexte actuel, les étudiants et les personnes qui aiment s'amuser en général organisent souvent des soirées ou des fêtes entre amis. Pourtant, planifier et organiser de tels événements peut s'avérer difficile, en particulier lorsqu'il s'agit de fixer une date convenant à la majorité des participants et de gérer les dépenses. Les défis augmentent lorsque le nombre de participants s'accroît.

Afin de résoudre ce problème, j'ai décidé de créer une application web qui facilite l'organisation de petites fêtes estudiantines. Ce sujet présente un intérêt technique, car il implique la création d'une plateforme en ligne accessible à un large public. De plus, il répond à un besoin réel des utilisateurs qui cherchent à simplifier l'organisation de leurs événements et à réduire les efforts nécessaires pour coordonner les participants.

L'application apportera une plus-value en proposant des fonctionnalités spécifiques pour gérer les aspects clés de l'organisation d'une fête, telles que la répartition des coûts, la gestion des courses et le choix d'une date. En outre, elle offrira un positionnement unique par rapport aux solutions existantes en offrant toutes ces fonctionnalités au sein d'une meme application.

Enfin, le sujet a été validé par un cours sondage au pres d'une vingtaine d'étudiants et une recherche d'application similaire deja existante, qui ont montré un intérêt pour une telle solution. Ce travail représentera une charge de travail suffisante, estimée à environ 350 heures.

3 Cahier des charges

3.1 Contexte

Dans le contexte des étudiants et des personnes qui aiment s'amuser, l'organisation de petites fêtes estudiantines est souvent une tâche complexe. Il est nécessaire de prendre en compte la date, les dépenses, la logistique et les préférences des participants. Pour faciliter ce processus, une application web est proposée pour assister les organisateurs et les participants dans la gestion des différents aspects de l'événement.

3.2 Besoin du client

— Permet de répartir équitablement les frais engendrés par la fête
— Crée une liste des courses pour la soirée (éventuellement une répartition si toutes les courses ne se font pas au meme endroit ?)
— Connaître qui a amène quoi à la fête
— Crée différents évènements
— Est accessible grâce à un simple lien
— Est pourvu d'un système de connexion et de création de compte simple
— Permet de trouver une date d'évènement qui convient au plus grand nombre de participants

3.3 Fonctionnalités optionnelles

— Permet de créer une playlist commune (sur laquelle chacun a la possibilité d'ajouter des titres)
— Permet au conducteur de connaitre son trajet de covoiturage
— Permet au participant d'organiser facilement des covoiturages
— Calcule et reparti automatiquement les frais de transport entre les différents covoitureurs

3.4 Contraintes

— Accessible sur toutes les plateformes (Android, IOS, Mac, Windows, Linux)
— Le système de création de compte doit être simple
— Les données des utilisateurs seront sécurisées
— L'application doit être auto-hébergable
— Respect des réglementations en matière de protection des données (GDPR)

3.5 Méthodologie

Pour la réalisation du projet, une méthodologie inspirée de Scrum sera utilisée, avec la division des demandes de l'utilisateur en petites tâches. L'outil FocalBoard sera utilisé pour organiser et suivre l'avancement des tâches. Le code sera géré sur GitHub, avec l'intention d'utiliser l'intégration et le déploiement continus, ainsi que des tests automatisés pour certaines parties du projet. L'étudiant travaillera en étroite collaboration avec le rapporteur pour valider les différentes étapes du projet et s'assurer que les attentes sont respectées.

3.6 Interactions avec le client (et/ou le rapporteur)

L'étudiant fournira des mises à jour régulières au rapporteur sur l'avancement du projet, et sollicitera des conseils et des orientations si nécessaire. Des réunions périodiques pourront être organisées pour discuter des problèmes, des progrès et des améliorations possibles.

4 Analyse

4.1 Aspects technologique

Le projet ayant comme contrainte principale d'être disponible sur toutes les plateformes majeures, une base web me paraît évidente. Pourtant, le web implique aussi la limitation principale qu'est le besoin de connexion permanente au serveur.
Pour remédier à ce problème, une PWA (Application Web Progressive) nous permettrait de bénéficier des avantages du web, tout en permettant aux utilisateurs de conserver une version locale de l'application. De plus, une PWA permet d'être installée comme une application native Android ou iOS pour les plateformes mobiles, ce qui facilite l'accès. Cependant, les PWA augmentent aussi énormément la complexité du projet et limitent le développeur dans son choix d'outils. La première version de l'application sera donc un site web responsive qui pourrait éventuellement être transformé en PWA par la suite.

Il ne faut cependant pas perdre de vue le second objectif de cette application qu'est la possibilité de la déployer simplement.

Toute application web nécessite plusieurs composants :

— Une interface utilisateur
  un gestionnaire de données
  un gestionnaire de requêtes
  une libraire de composant graphique
— Une Couche de calculs applicatifs et de persistance de données
  Une Base de Donnée
  Un ORM
  Une API définie
  Un système de matching pour les dettes
  Un système d'équilibrage des dépenses entre les utilisateurs

4.1.0.1 TypeScript

Avant de regarder quels outils utiliser pour les différents composants, je pense qu'il est important de choisir un langage de programmation.

Pour ce projet, j'ai décidé d'utiliser le même langage pour les 2 composants majeurs que sont le front-end et le back-end. Plusieurs choix s'offraient alors à moi, JavaScript, Kotlin, TypeScript ou n'importe quel langage compilable en web assembly.

JavaScript n'était pas vraiment un bon choix vu qu'il ne propose pas de typage, ce qui rend le code moins prédictible et robuste.

Kotlin est un bon candidat, il offre un typage statique, mais propose de l'inférence de type, peut être transpilé vers du JavaScript ou du web assembly, les outils de développement pour Kotlin sont aussi très bons, mais la documentation pour une utilisation web est plutôt pauvre, comparée aux autres options.

TypeScript a quant à lui tous les avantages de Kotlin, grâce au typage fort, et profite aussi de la documentation et des outils JavaScript grâce à sa nature de superset. Lorsque l'on développe un produit web, surtout pour un développeur full-stack, TypeScript est

l'idéal.
Ses outils, l'environnement JavaScript, la sécurité des types et du null, la documentation, sont toutes de bonnes raisons d'utiliser TypeScript. Pour toutes ces raisons, TypeScript est le langage idéal pour le développement Web.

4.1.1 Front-end

Pour ce qui est de l'interface utilisateur (le front-end) j'ai décidé d'utiliser Angular car ce framework utilise TypeScript nativement tout en supportant aussi les PWA. Angular étant un framework orienté, il m'oblige aussi à suivre une architecture propre qui permet de rendre independents les différents composants du projet .

4.1.1.1 Gestion des données

Pour conserver les données hors ligne et facilité l'accès à des données à jour lors d'une connexion j'ai décidé d'utiliser NgRx. Ce dernier est un framework de gestion de données qui simplifie l'accès et le caching de données

4.1.1.2 Gestionnaire des requêtes

Lorsque l'on veut interagir avec une API graphql il est préférable d'utiliser un client.

Il existe une multitude de clients GraphQl mais le plus populaire et le mieux documenté reste Apollo. De plus il existe un wrapper pour le client Apollo qui l'intègre dans Angular nommé Apollo Angular. Je vais donc utiliser cette librairie pour récupérer mes données au pres de mon serveur .

4.1.1.3 librairie CSS

Utiliser une librairie de composants permet de simplifier la création de l'interface grâce à des blocks de construction appelé composants. Angular permet d'utiliser des librairies de composants avec leur propre logique, structure, et CSS. Ces librairies fournissent aussi un ensemble de classe CSS pour rendre nos propres composants jolis.
De plus, elles permettent de rendre notre application responsive sans devoir y penser pour chaque composant. Il existe plusieurs librairies majeures, les principales sont :

- Angular Material
- PrimeNg
- NgBootstrap

Angular Material est la librairie officielle, elle est d'origine Google, offre beaucoup de composants différents en plus de plusieurs composants de layout qui permettent de d'organiser d'autres composants sur la page.

PrimeNg offre aussi un grand nombre de composants, mais n'offre pas de layout par contre, PrimeNg permet d'utiliser un autre thème que Material.

NgBootstrap est un simple wrapper pour Bootstrap css et propose tous les composants de bootstrap. Vu que mon objectif final serait de créer une PWA qui sera installable

sur Android, utiliser un thème similaire à ceux des applications Google intégrera mieux l'application.

4.1.2 Back-end

Le serveur back-end doit être capable de mettre à disposition les données nécessaires au fonctionnement de l'application. Cependant, le back-end sera aussi responsable des calculs d'équilibrage des dépenses, des matching de covoiturage et des trajets.

Connaissant déjà Express.js et Spring (Kotlin) je me suis naturellement tourné vers ces derniers, néanmoins je désirais utiliser du TypeScript pour ce projet. Express.js est compatible avec TypeScript mais Spring ne l'est pas. Express.js est leger, minimaliste et propose une bonne documentation, Spring est généraliste propose des outils de génération de code et utilise un système d'injection de dépendance. Ce qu'il me fallait était une hybride entre Spring et Express.js. Après quelque recherches, j'ai trouvé un framework appelé Nest.js, ce dernier supporte nativement TypeScript, propose une documentation excellente, propose un CLI permettant de générer du code boiler-plate, reste simple et leger, Utilise de l'injection de dépendance et est très utilisé et apprécié par d'autres développeurs. J'ai donc décidé d'utiliser Nest.Js pour ce projet.

4.1.2.1 ORM

Nest.Js (que je vais appeler Nest à partir de maintenant) n'est pas un framework batteries included comme le sont Spring ou Laravel, Il m'a alors fallu choisir aussi un ORM pour interfacer avec ma base de données. Dans la Documentation de Nest, il y a des exemples de configuration avec les ORM les plus utilisés avec Nest et TypeScrip. Ces derniers sont Sequelize, TypeOrm ainsi que Prisma. Ayant eu une experience avec Sequelize plutôt mauvaise lors de mon projet de dev3, j'ai décidé de regarder du côté de prisma.

Prisma est une ORM un peu particulière dans le sens où il propose d'utiliser un schema prisma pour définir ses entités au lieu d'utiliser du code TypeScript. Cella me paraissait une très bonne chose vu que prisma générait alors lui-même les différents DTO nécessaires. Il s'est cependant avéré qu'à l'utilisation prisma n'est pas l'idéale surtout lorsque l'on désire le coupler à une API graphQl.

À cause de son schema particulier, je devais définir le forma de mes entités à 2 endroits et tout de même créer manuellement mes DTO. Sans compter que les opérations de CRUD dans les entités étaient rendues très complexe dû aux liens entre elles et que Prisma ne permet pas d'utiliser des identifiants.

J'ai donc décidé de jeter un œil à TypeOrm, celui-ci propose de définir les entités via des annotations TypeScript. Ce qui me permet d'utiliser une seule classe pour définir mon entité d'ORM et de graphQl de plus TypeOrm est mieux documenté grâce au support d'une communauté plus large.

4.1.2.2 L'API back end

Il existe plusieurs types d'API très utilisées pour le Back-end, mais les plus populaires sont le REST et le SOAP. SOAP est de moins en moins utilisé dû à la complexité intrinsèque du xml et à sa lourdeur.

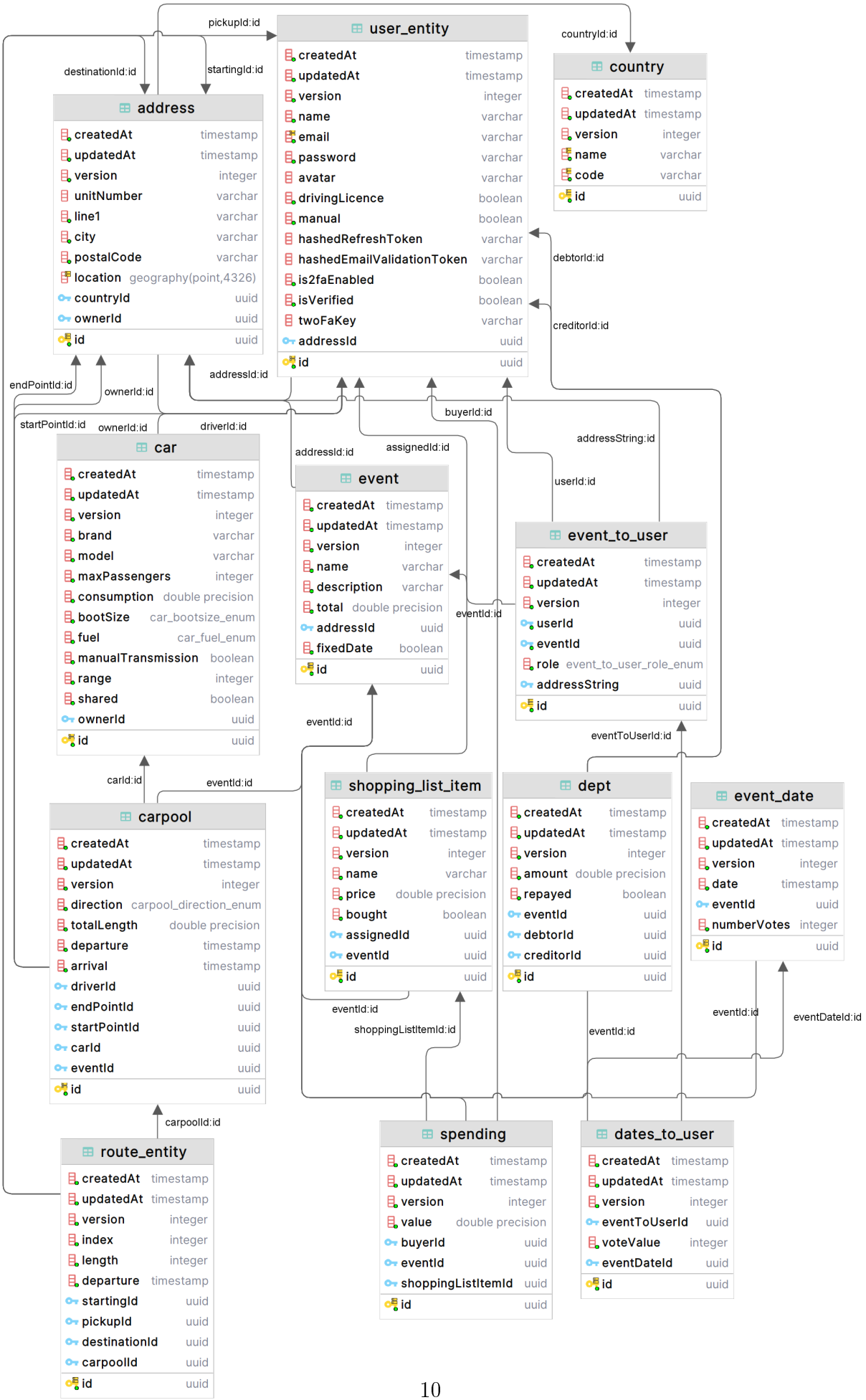
Il nous reste alors REST qui est une bonne solution, mais nécessite pour presque chaque requête de récupérer des données qui ne seront pas toujours consommées par le client. Avec Rest lorsque l'on récupère une entité complete tous les champs sont renvoyés. C'est pour cela que graphQl a été crée, non seulement graphQL utilise un typage fort via un schema défini et exposé par le serveur, mais en plus graphql permet de ne récupérer que les champs nécessaires au client grace au GQL .

4.1.2.3 La base de donnée

Ma base de donnée devait me permettre 2 choses, stocker des données géographiques et faire des operations sur ces données. Le seul choix qui s'offrait à moi compatible avec TypeOrm était Postgres avec une extension nommée POSTGIS. Postgres est une base de donnée très utilisée en production, très performante et stable.

4.1.2.4 Structure de donnée

Pour stocker toutes les données nécessaires au fonctionnement de l'application, il nous faut une structure de donnée cohérente. Voici un schema1 de la base de donnée et de sa structure :



Architecture de la base de donnée

4.1.3 Hébergement

Afin de permettre aux utilisateurs souhaitant auto-héberger l'application, il est essentiel de fournir un environnement préconfiguré ou indépendant de l'hôte. La meilleure solution, en l'occurrence, consiste à conteneuriser l'application.

4.1.3.1 Conteneurisation

Pour mettre en conteneur l'application, diverses options sont disponibles. Nous pourrions, par exemple, créer une image Docker englobant l'ensemble de l'application. Cette image comprendrait à la fois la base de données, le service d'API et le serveur web hébergeant l'interface utilisateur. Bien que cette solution soit loin d'être idéale, elle offre un avantage considérable pour l'utilisateur, en permettant la mise en place de l'application en une seule commande.

Cependant, cette approche ne facilite pas l'ajout de services supplémentaires, tels qu'un serveur de messagerie ou un autre type de base de données, par d'éventuels futurs développeurs. Afin de conserver les avantages de cette solution monolithique tout en séparant les différents services (serveur web, API, base de données, etc.) dans des conteneurs distincts, nous pourrions utiliser des outils tels que Docker Swarm, Docker Compose ou d'autres systèmes d'orchestration, tels que Kubernetes.

Un script Docker Compose simple semble être la meilleure option pour permettre aux utilisateurs souhaitant auto-héberger l'application de manière aisée. Néanmoins, il serait également intéressant d'offrir la possibilité aux utilisateurs plus avancés d'héberger l'application avec une haute disponibilité.

4.1.3.2 Orchestration

Afin d'assurer une haute disponibilité, il est essentiel que l'application ne présente aucun point de défaillance unique. Par exemple, sans haute disponibilité, si le système d'exploitation du serveur subit un crash, l'ensemble de l'application serait impacté. Pour renforcer la résilience face à ce type de problème, nous introduirons de la redondance à tous les niveaux du système d'hébergement en utilisant un outil d'orchestration de conteneurs sur plusieurs machines.

Plusieurs outils d'orchestration de conteneurs pour un cluster de machines sont disponibles, mais Kubernetes se distingue par sa simplicité d'approche. Nous mettrons donc en place un cluster Kubernetes pour héberger l'application avec une haute disponibilité. De plus, ce cluster facilitera également la gestion des certificats TLS.

Toutefois, la mise en place d'un cluster Kubernetes ne garantit pas la portabilité de notre cluster. Une fois établi, il demeure unique, à l'instar d'un flocon de neige. Pour résoudre ce problème, nous ferons appel à l'infrastructure en tant que code (Infrastructure as Code - IaC).

4.1.3.3 Infrastructure as code

Comment rendre notre cluster et par extension notre application déployable en une seule commande ? J'ai opté pour l'utilisation d'Ansible afin de déployer mon cluster, indépendamment de l'état de la machine. Ansible offre effectivement une approche agnostique qui facilite le processus de déploiement.

Un simple script shell aurait pu être utilisé, mais il ne serait pas agnostique par rapport à l'état de l'OS. Par exemple, si une commande demande de créer le répertoire /home/foo

mais qu'il existe déjà, le script shell échouerait. En revanche, Ansible poursuivra son exécution sans problème, car pour lui, si le répertoire existe déjà, cela signifie que l'objectif est atteint.

4.2 Aspect non technologique

4.2.1 Tests unitaires et tests d'intégration

Le code de l'application est testé à l'aide de tests unitaires et de tests d'intégration pour le backend. Ces tests permettent de vérifier le bon fonctionnement des fonctionnalités individuelles et de leur intégration au sein de l'ensemble du système.

4.2.2 Sécurité et confidentialité

4.2.2.1 Sécurité

L'application n'utilise pas HTTPS vu que cela nécessiterait des certificates SSL, cependant il est vivement conseiller d'utiliser un reverse proxy pour réaliser la terminaison SSL.
Néanmoins, les mots de passe sont chiffrés et salés avant d'être stockés en base de données grâce à Argon2.
Angular et NestJs permettent d'empêcher les attaques de type SQL injection. En ce qui concerne les attaques de type DoS ou DDoS, aucune protection n'est mise en place au niveau applicatif, car il est plus simple d'implémenter cette sécurité au niveau du réseau.

4.2.2.2 GDPR

En ce qui concerne le GDPR, aucune donnée récoltée n'est revendue ni redistribuée. Seules les données fournies directement par les utilisateurs via les formulaires sont récoltées et utilisées uniquement dans l'application. Ces données peuvent être supprimées définitivement avec une simple requête à l'API.

4.2.3 Accessibilité

Le site web utilise des composants HTML standards, ce qui permet à tous de l'utiliser. De plus, grâce aux librairies CSS, le contraste est garanti, ce qui améliore l'accessibilité pour les utilisateurs ayant des problèmes de vision.

4.2.4 Optimisation et performance

L'application web intégrera un système de cache grâce à Apollo Angular et sera minifiée pour la mise en production. Ces optimisations permettent d'améliorer les performances et la rapidité de l'application.

4.2.5 Documentation

Le projet sera documenté via le fichier README et directement dans le code. Des noms de variables et de fonctions explicites, ainsi qu'un typage fort, assurent une solide

base de documentation. Les fonctions plus complexes nécessitant une documentation plus approfondie seront documentées avec de la Javadoc.

Pour ce qui est de l'API, le schéma GraphQL sert de documentation.

De plus un linter est mis en place pour assurer le respect des bonnes pratiques.

4.2.6 Gestion de projet

Focalboard a été choisi comme outil de gestion de projet. Il est très similaire à Trello, mais il est open-source et est auto-hébergé sur mon serveur. Cela permet de suivre l'évolution du projet et de gérer les tâches à accomplir efficacement.

De plus le code sera gere au moyen de repository git

4.2.7 Modularité et évolutivité

L'application étant construite avec Angular et NestJs, elle est très modulaire par nature. Cela facilite son évolution et permet d'ajouter ou de modifier des fonctionnalités sans impacter l'ensemble du système.

5 Hebergement et aspects reseaux

Afin d'héberger Clever Party Thrower, plusieurs possibilités s'offrent au client. Celles-ci dépendent de son budget, de l'infrastructure dont il dispose et du nombre d'utilisateurs qui devront être servis.

5.1 docker-compose

L'application, étant packagée dans des conteneurs Docker, peut être déployée très facilement dans un environnement restreint grâce à Docker et Docker Compose. Durant le développement de l'application, j'ai souhaité permettre à des utilisateurs potentiels du service de tester certaines parties de l'interface. J'ai donc décidé d'héberger un stack Docker Compose sur un petit serveur de mon homelab. Le stack que j'ai choisi de décrire dans la documentation est minimaliste, il comprend le strict nécessaire pour héberger et utiliser l'application. Le fichier docker-compose comprend le backend, la base de données et le frontend hébergé par un serveur nginx.

5.2 kubernetes

Lorsque le client dispose d'une infrastructure performante, il a la possibilité de déployer l'application sur un ou plusieurs clusters Kubernetes. Durant le développement de l'application, je me suis intéressé à l'orchestration de conteneurs avec Kubernetes, ce qui m'a conduit à proposer des fichiers de configuration pour héberger le site web et son serveur.

En parallèle de ma formation sur Kubernetes, j'ai également exploré Ansible et ai adapté un projet open-source existant afin de permettre le déploiement de Clever Party Thrower sur un cluster Kubernetes en une seule commande. Ce playbook Ansible facilite le déploiement d'un cluster K3S complet, avec ou sans haute disponibilité. De plus, le playbook déploie également des applications telles que kubeVip, MetalLB, Rancher, Traefik, Cert-manager, LongHorn et ArgoCD.

5.2.1 Le Playbook

Le Playbook permet de déployer Kubernetes sur plusieurs machines, appelées nodes, afin d'ajouter de la résilience au cluster. Plus il y a de nodes, plus le cluster peut en perdre sans interruption de service. Les différentes applications utilisées pour permettre cette haute disponibilité sont les suivantes :

- **Kubernetes** : Kubernetes synchronise et répartit toutes les configurations et les ressources entre les différentes nodes pour garantir que chacune d'entre elles peut à tout moment gérer le trafic.
- **Kubevip** : Kubevip sert à créer une adresse virtuelle pour la gestion du cluster. Cette adresse virtuelle permet de conserver une connexion avec le cluster tant qu'au moins une des nodes master est en ligne et fonctionnelle .
- **MetalLB** : MetalLB est un load balancer qui, comme son nom l'indique, permet d'équilibrer la charge de travail. Il attribue également des adresses IP à des pods (conteneurs dans Kubernetes) en fonction de leur nom ou espace de nom et ce, dans une plage définie .

— **Cert-manager** : Cert-manager permet de créer et renouveler les différents certificats via Let's Encrypt, en plus de permettre à Kubernetes de les gérer comme toute autre ressource, et donc de les synchroniser entre les différentes nodes .
— **Traefik** : Traefik sert de reverse proxy, protégeant ainsi les différents services du cluster. Il gère également la distribution de la charge pour les requêtes HTTPS .
— **Longhorn** : Longhorn permet de créer des volumes partagés et disponibles sur plusieurs nodes en même temps, garantissant ainsi aux pods exploitant ces volumes qu'ils seront toujours accessibles.

5.3 Les Conteneurs

Le conteneur back-end est relativement simple. Il utilise une image node :latest. Une fois le code source transpilé, le résultat de cette compilation est copié dans l'image. Le conteneur pour la base de données utilise quant à lui une simple image de PostGIS. PostGIS est une extension de PostgreSQL qui permet la manipulation de points géographiques directement sur la base de données. Le conteneur front-end est plus complexe. Basé sur une image nginx, il intègre le résultat de la compilation Angular dans le dossier où nginx cherche les fichiers à servir. Nginx sert donc le front end, et agit également comme un proxy pour permettre au front end d'accéder au backend.

5.4 Mon choix

En tant que client, j'ai choisi d'héberger l'application via docker-compose. À l'origine, mon objectif était de créer un cluster Kubernetes distribué entre plusieurs machines virtuelles pour simuler diverses machines physiques, et de déployer l'application sur ce cluster. Cependant, Kubernetes, et surtout la haute disponibilité, exige beaucoup de ressources, que mon serveur ne pouvait pas supporter. Les différentes machines virtuelles manquaient constamment de mémoire ou d'espace disque. J'ai ainsi décidé d'utiliser un simple docker-compose pour optimiser les ressources de mon serveur, permettant d'héberger toutes mes applications plutôt que de se limiter à une seule avec des performances médiocres. À l'avenir, si les limites matérielles ne sont plus un obstacle, j'envisagerai de déployer une infrastructure Kubernetes via Ansible, avec l'ajout d'un système de surveillance.

5.5 Sécurité et hébergement

Étant donné que le client est potentiellement responsable de l'hébergement de l'application, une grande partie des responsabilités en termes de sécurité lui revient. L'application doit être hébergée derrière un reverse proxy pour assurer la terminaison SSL, par exemple. Les deux différentes méthodes d'hébergement assurent la sécurité de la base de données via le réseau Docker : seul le backend a accès à la base de données.

Les différentes applications sont maintenues à jour soit via ArgoCD sur Kubernetes, soit via Watchtower sur Docker Compose. Ces deux applications surveillent l'état du dépôt Docker Hub en temps réel et mettent à jour les services lors d'un changement.

Pour mon installation, j'ai choisi de faire confiance à Cloudflare pour sécuriser mon hébergement. J'utilise un tunnel et leur reverse proxy afin de garantir la sécurité de mes services. Cloudflare gère non seulement les certificats, mais permet aussi de bloquer les attaques de type DDoS. De plus, grâce au tunnel, je n'ai pas à configurer mon pare-feu ni à exposer mon adresse IP publique.

À l'avenir, j'aimerais utiliser un pare-feu comme PfSense/OpenSense pour me permettre de faire du port forwarding directement tout en garantissant la sécurité de l'application. De plus, lorsque l'application sera hébergée sur un cluster Kubernetes, la gestion des certificats et de la terminaison SSL sera effectuée par le cluster. La configuration actuelle obtenue via le playbook Ansible met déjà en place un certificat privé pour l'application, géré par Certmanager, ce qui le rend hautement disponible. Cette disponibilité me permettra d'utiliser plusieurs instances concurrentes de Traefik réparties entre les différentes nodes du cluster.

5.6 Performance et résilience à la charge

Pour assurer la performance et la résilience à la charge de l'application Clever Party Thrower, plusieurs stratégies ont été mises en place .

— **Réplication de conteneurs** : Grâce à Kubernetes, il est possible de déployer plusieurs instances de l'application sur différentes nodes . Cette approche permet d'améliorer la disponibilité de l'application en cas de panne sur une node, et également de répartir la charge utilisateur entre plusieurs instances .

— **Load balancing** : Utiliser un load balancer comme MetalLB dans un environnement Kubernetes permet de distribuer efficacement le trafic réseau entre plusieurs pods. Cela améliore la répartition de la charge et réduit le risque de surcharge d'une seule instance, ce qui se traduit par une meilleure performance globale de l'application .

— **Scalabilité horizontale** : La scalabilité horizontale, qui consiste à ajouter plus de nodes à un cluster, est une autre stratégie pour gérer une charge croissante . Avec l'aide de Kubernetes et de son fonctionnement basé sur des clusters, cette scalabilité peut être réalisée de manière relativement simple .

— **Gestion des ressources** : Kubernetes offre également des outils pour contrôler la quantité de ressources CPU et de mémoire que chaque pod peut utiliser . Cela permet de prévenir les situations où un pod consomme trop de ressources et affecte la performance des autres pods .

— **Volumes partagés avec Longhorn** : Longhorn assure la haute disponibilité des données en permettant de créer des volumes partagés et accessibles sur plusieurs nodes simultanément . Cela garantit que les pods exploitant ces volumes peuvent toujours y accéder, même en cas de défaillance d'une node .

Ainsi, en utilisant les outils et les principes de conception appropriés, il est possible de créer une application qui peut gérer efficacement une charge élevée tout en conservant une performance satisfaisante. Dans le futur, des outils de surveillance et d'alerte pourraient être ajoutés pour assurer une détection proactive des problèmes de performance et permettre une intervention rapide en cas de problèmes.

6 Développement de l'application

6.1 Introduction

Cette section introduit l'architecture, la documentation du code, les bonnes pratiques de programmation, les tests unitaires et la sécurité de l'application web Clever Party Thrower.

6.2 Architecture

Clever Party Thrower est une application web full-stack. Le front-end est construit avec Angular tandis que le back-end est un monolithe élaboré avec NestJS et TypeORM. À chaque accès au site par un utilisateur, la Single Page Application (SPA) Angular est chargée, initiant les services nécessaires pour interagir avec le back-end. Ces services récupèrent les données requises via l'API GraphQL, en utilisant Apollo.

Durant toute la session de l'utilisateur, ces services persistent et continuent de récupérer les données du back-end au fur et à mesure de la navigation. Chaque service est responsable d'un type de données spécifique. Par exemple, AuthService s'occupe des données liées à l'authentification et à l'utilisateur, tandis qu'EventService gère tout ce qui est directement lié à un événement.

De la même manière, le back-end est divisé en plusieurs services, chacun gérant une ressource unique. Ces services sont rassemblés en modules, associés aux contrôleurs REST et aux résolveurs GraphQL. Cette modularité facilite le test du code en assurant un découplage entre les différents services.

6.3 Documentation du code

Le code de Clever Party Thrower est organisé et documenté pour faciliter sa compréhension et sa maintenance. Grâce à TypeScript, la majorité de la documentation est implicite. Les noms des fonctions et leurs arguments offrent une compréhension rapide de leur utilité. Pour les fonctions plus complexes, des commentaires expliquent la logique algorithmique. Ainsi, même un développeur novice dans le projet peut facilement naviguer dans le code.

6.4 Bonnes pratiques de programmation

Clever Party Thrower suit les conventions de codage d'Angular et de NestJS, ce qui comprend l'utilisation appropriée des indentations, des commentaires, de la casse des lettres (camelCase, PascalCase), et des conventions de nommage des variables, des fonctions et des classes. Cette pratique améliore la lisibilité du code et facilite le travail en équipe.

De plus, le code est divisé en modules et services distincts, chacun étant responsable d'une fonctionnalité spécifique. Cela favorise la modularité, permet une meilleure gestion du code et facilite la maintenance et l'extension de l'application.

Les principes DRY (Don't Repeat Yourself) et SOLID sont également respectés pour minimiser la duplication de code et assurer la prédictabilité du comportement du logiciel. Quelques concepts de programmation fonctionnelle sont utilisés dans le front-end, tels que l'immutabilité et l'utilisation maximale de fonctions pures.

La gestion des erreurs en JavaScript et TypeScript est plus basique que dans d'autres langages de programmation. Contrairement à Java, où les erreurs doivent être déclarées dans la définition de la fonction, TypeScript ne possède pas cette fonctionnalité. Il est donc difficile de savoir quelle fonction peut lancer une erreur. La meilleure pratique est d'utiliser des blocs try-catch au niveau le plus élevé de l'application pour capturer les erreurs une fois qu'elles sont propagées.

6.5 Tests Unitaires

Les tests unitaires sont une partie essentielle de Clever Party Thrower. Chaque module et service dispose de ses propres tests unitaires, garantissant ainsi que chaque partie de l'application fonctionne comme prévu. Les tests sont écrits avec Jest, un framework de test populaire pour JavaScript et TypeScript.

Le découplage des services et des modules rend les tests plus simples et plus efficaces, car chaque test se concentre sur une seule unité de code.

6.6 Sécurité

La sécurité est une préoccupation majeure pour Clever Party Thrower. Parmi les mesures de sécurité mises en place, on compte l'utilisation de HTTPS pour toutes les communications via un reverse proxy dans l'infrastructure d'hébergement, l'authentification et l'autorisation basées sur des tokens JWT, la validation des entrées côté serveur pour prévenir les attaques par injection, et le chiffrement des données sensibles comme les mots de passe avant leur enregistrement en base de données. Les vulnérabilités potentielles sont régulièrement évaluées et des mises à jour de sécurité sont appliquées dès que nécessaire via npm audit et npm audit fix.

7 Base de donnees

7.1 Introduction

Dans le contexte de mon application Clever Party Thrower, une base de données est nécessaire pour gérer diverses informations générées par les utilisateurs, ainsi que pour stocker d'autres données essentielles au bon fonctionnement de l'application. Cette section fournit un aperçu détaillé de la conception, du choix, des opérations et de la structure de la base de données.

7.2 Choix de la base de données

Le choix du système de base de données est un aspect crucial de tout projet, car il influence directement les performances et la fonctionnalité de l'application. Comme je l'ai précisé dans la section d'analyse, j'ai choisi un système de base de données de type SQL, plus précisément PostgreSQL, pour ce projet. La haute disponibilité offerte par PostgreSQL, grâce à ses fonctionnalités avancées de réplication de données, a été l'une des principales motivations de ce choix.

De plus, PostgreSQL offre une grande extensibilité. Son système d'extensions permet d'ajouter facilement des fonctionnalités à la base de données. Par exemple, l'extension PostGIS, qui facilite le stockage et la manipulation des données géographiques, a été particulièrement utile pour mon application, en particulier pour la fonction de covoiturage qui nécessite le calcul de la distance entre deux points géographiques.

En outre, le caractère open source de PostgreSQL, sa réputation solide et ses performances de haut niveau en font un choix idéal pour mon projet. Sa compatibilité avec de nombreux langages de programmation et ORMs, dont TypeOrm que j'ai utilisé pour le développement de l'application, a consolidé ce choix.

7.2.1 Justification du choix d'une base de données relationnelle

Le choix d'une base de données relationnelle pour le projet Clever Party Thrower repose sur plusieurs raisons. Tout d'abord, les données de l'application sont structurées et présentent de nombreuses relations entre elles. Une base de données relationnelle, comme PostgreSQL, est idéale pour gérer ce type de données. De plus, les bases de données relationnelles fournissent des mécanismes de transactions puissants, ce qui est crucial pour maintenir l'intégrité des données lors de l'exécution des opérations CRUD. Enfin, PostgreSQL supporte l'ACIDité (Atomicité, Cohérence, Isolation, Durabilité), ce qui garantit que toutes les transactions sont traitées de manière fiable.

7.3 Opérations sur la base de données

Les opérations sur la base de données sont principalement de nature CRUD (Create, Read, Update, Delete). Cependant, avec l'introduction du système de covoiturage, des opérations plus complexes sont devenues nécessaires. Par exemple, le calcul des distances entre différents points géographiques est une opération essentielle pour la coordination du covoiturage.

7.4 Structure de données

7.4.1 Normalisation de la base de données

La base de données du projet a été conçue en suivant les principes de la normalisation afin d'éliminer la redondance des données et d'assurer l'intégrité des données. Nous avons atteint la 3NF (Third Normal Form) qui assure que chaque colonne d'une table est non-transitivement dépendante de la clé primaire. Cela signifie que toutes les données non-clé sont complètement dépendantes de la clé primaire, ce qui aide à réduire la redondance et à améliorer l'intégrité des données.

7.4.2 Structure de la base de données

La structure de la base de données est illustrée dans la figure dans la section analyse4. Cette structure a été conçue pour garantir une exécution efficace des opérations de la base de données, tout en assurant la cohérence et l'intégrité des données. Elle comporte plusieurs tables, dont chaque instance contient des informations pertinentes et est liée à d'autres tables pour créer des relations significatives entre les différentes données. Voici une brève description ce chaque table de la basse de donnees

- address : Contient des informations sur les adresses. Chaque instance d'adresse est liée à une instance de " country " (via " countryId ") et à une instance de " user_entity" (via "ownerId").
- car : Contient des informations sur les voitures. Chaque instance de voiture est liée à une instance de "user_entity" (via "ownerId").
- carpool : Contient des informations sur le covoiturage.
- Chaque instance de covoiturage est liée à une instance de "address" (via "startPointId" et "endPointId"), une instance de "car" (via "carId"), une instance de "event" (via "eventId"), et une instance de "user_entity" (via "driverId").
- country : Contient des informations sur les pays. Il est lié à " address" (via "id").
- dates_to_user : Contient des informations sur les dates liées aux utilisateurs. Chaque instance est liée à une instance de "event_date" (via " eventDateId") et à une instance de "event_to_user" (via " eventToUserId").
- dept : Contient des informations sur les dettes. Chaque instance est liée à une instance de "event" (via " eventId"), et à deux instances de " user_entity " (via " creditorId" et "debtorId").
- event : Contient des informations sur les événements. Chaque instance est liée à une instance de "address" (via "addressId") .
- event_date : Contient des informations sur les dates d'événement. Chaque instance est liée à une instance de "event" (via "eventId").
- event_to_user : Contient des informations sur la relation entre les événements et les utilisateurs. Chaque instance est liée à une instance de "address" (via "addressString"), une instance de "event" (via "eventId"), et une instance de "user_entity" (via "userId").
- route_entity : Contient des informations sur les itinéraires. Chaque instance est liée à deux instances de "address" (via "startingId" et "destinationId"), une instance de "carpool" (via "carpoolId"), et une instance de "user_entity" (via "pickupId") .
- shopping_list_item : Contient des informations sur les articles de la liste de courses. Chaque instance est liée à une instance de "event" (via " eventId") et

à une instance de "user_entity" (via "assignedId").
— spending : Contient des informations sur les dépenses. Chaque instance est liée à une instance de "event" (via "eventId"), une instance de "shopping_list_item" (via "shoppingListItemId"), et une instance de "user_entity" (via "buyerId").
— user_entity : Contient des informations sur les utilisateurs. Chaque instance est liée à une instance de "address" (via "addressId").
Le schema est disponible ici1

7.5 Gestion des transactions et de la concurrence

Pour gérer la concurrence et assurer l'intégrité des transactions, PostgreSQL utilise un modèle MVCC (Multi-Version Concurrency Control). Cela signifie que chaque transaction opère sur une "instantané" de la base de données, ce qui permet d'éviter les conflits lors de l'accès simultané à la base de données par plusieurs transactions. De plus, PostgreSQL supporte les propriétés ACID, ce qui garantit que toutes les transactions sont complètement exécutées ou complètement annulées, même en cas de panne du système.

7.6 Sécurité de la base de données

La sécurité de la base de données a été un aspect crucial de la conception du système. PostgreSQL offre une gamme de fonctionnalités pour la gestion des accès, ce qui nous a permis de définir des rôles et des permissions uniquement pour le back-end, ce dernier est le seul a avoir access a la base de donnée . De plus, toutes les données sensibles sont chiffrées à l'aide de protocoles de sécurité robustes via argon2 avant d'etre inserée dans la base de donnée. Enfin, des mesures on ete mise en place pour prévenir les injections SQL via typeORM qui utilise des requêtes paramétrées et évite l'interpolation de chaînes dans nos requêtes.

7.7 Performance et optimisation

Afin d'optimiser les performances de la base de données, nous avons utilisé plusieurs techniques. Par exemple, l'indexation est mise en place pour accélérer les requêtes sur des tables avec de grands volumes de données grace a typeORM. De plus, PostgreSQL utilise un mécanisme de cache efficace qui améliore la vitesse d'exécution des requêtes fréquentes.

7.8 Stratégies de sauvegarde et de restauration

Pour l'application Clever Party Thrower, aucune stratégie de sauvegarde intégrée n'a été mise en place, et ce, pour plusieurs raisons. L'une des raisons principales est la spécificité des besoins en sauvegarde qui peuvent varier grandement en fonction du client et de son infrastructure informatique. Il n'est pas toujours idéal d'imposer une stratégie de sauvegarde spécifique dans le cadre du projet lui-même, car cela pourrait ne pas s'adapter parfaitement à l'infrastructure du client.

L'infrastructure utilisée pour héberger l'application est en effet un facteur déterminant dans le choix d'une stratégie de sauvegarde. Par exemple, le choix entre une sauvegarde sur site, hors site ou dans le cloud, ou une combinaison de celles-ci, dépend largement de l'infrastructure existante, des capacités de stockage disponibles et des exigences en matière de temps de récupération.

Malgré l'absence d'un mécanisme de sauvegarde intégré dans le projet, une stratégie de sauvegarde est néanmoins mise en place dans mon infrastructure. Plus précisément, j'ai mis en place un mécanisme de sauvegarde des volumes des conteneurs, ce qui inclut par conséquent aussi les données de PostgreSQL. Les fichiers de sauvegarde, qui incluent les données de la base de données ainsi que d'autres données d'application, sont stockés de manière sécurisée dans un emplacement de sauvegarde dédié.

Pour une restauration en cas de besoin, ces fichiers de sauvegarde peuvent être utilisés pour rétablir l'état de l'application à un état précédent. La précision de cet état dépend de la fréquence de la sauvegarde : plus la sauvegarde est fréquente, plus l'état restauré est proche du moment de la défaillance.

Il convient de noter qu'il est possible de mettre en place des mécanismes de sauvegarde plus sophistiqués si le client le souhaite. Par exemple, des sauvegardes incrémentielles, qui sauvegardent uniquement les données qui ont changé depuis la dernière sauvegarde, peuvent économiser de l'espace de stockage et améliorer l'efficacité. Des sauvegardes en temps réel ou quasi réel peuvent également être mises en place pour les applications nécessitant une très haute disponibilité.

7.9 Défis rencontrés

La conception et la mise en œuvre de la base de données, bien qu'en fin de compte réussies, n'ont pas été exemptes de défis. Au début du projet, j'ai choisi d'utiliser Prisma ORM pour gérer ma base de données, anticipant qu'il fournirait une interface pratique pour interagir avec les données. Cependant, j'ai rapidement découvert que Prisma ne répondait pas aussi bien que prévu aux besoins spécifiques de mon projet. Plus précisément, j'ai trouvé que l'itération sur la structure de données était plus complexe que prévu, ce qui a entraîné des inefficacités dans le processus de développement.

Confronté à ces difficultés, j'ai pris la décision de passer à TypeORM. Ce changement a marqué un tournant positif dans le projet. Contrairement à Prisma, TypeORM s'est avéré être un outil beaucoup plus adapté à mes besoins. Il a grandement facilité les différentes interactions avec la base de données, me permettant de surmonter les défis initialement rencontrés avec Prisma.

Un autre défi rencontré lors de la conception de la base de données a été la nécessité de migrer de PostgreSQL à PostGIS. Ce changement a été rendu nécessaire par le besoin d'exploiter les fonctionnalités avancées de PostGIS pour le traitement des données géographiques dans mon application. Cette migration, qui aurait pu être une tâche complexe et source d'erreurs, s'est avérée être une transition relativement lisse grâce à TypeORM. En fait, le changement de conteneur s'est déroulé sans problèmes majeurs, ce qui m'a permis de me concentrer sur d'autres aspects importants du développement de l'application.

En résumé, bien que la conception et la mise en œuvre de la base de données aient présenté certains défis, ces difficultés ont été surmontées grâce à des choix judicieux d'outils et à la capacité d'adapter rapidement la stratégie de développement en fonction des exigences du projet.

8 Historique du projet

8.1 Introduction

Dans cette section, nous examinerons l'évolution du projet, depuis sa phase de conceptualisation jusqu'à sa réalisation finale, en soulignant les différentes difficultés rencontrées en cours de route. Nous mettrons également en lumière les jalons clés du projet, les ajustements de la planification et les interactions avec le rapporteur.

8.2 Chronologie du projet

Le projet a officiellement débuté en septembre 2022, marqué par une première phase d'échanges intenses avec de potentiels utilisateurs et de définition des objectifs. Des étapes clés ont jalonné notre progression, telles que la finalisation du cahier des charges et la sélection des technologies en octobre 2022, le commencement du développement dans la même période, et enfin, le déploiement de la premiere version l'application en juin 2023.

8.3 Réunions et interactions avec le client/rapporteur

Au cours du projet, j'ai eu plusieurs réunions avec mon rapporteur. Ces réunions ont été l'occasion de discuter de l'avancement du projet, de recueillir des retours constructifs et d'adapter notre approche en conséquence. Par exemple, lors de la réunion de validation du sujet en octobre, nous avons convenu que la conception de l'application devrait être suffisamment flexible pour permettre l'ajout de nouvelles fonctionnalités à l'avenir, comme un système de covoiturage.

8.4 Évolution des choix techniques et stratégiques

Au fil du projet, certaines modifications ont été nécessaires. Ces ajustements étaient principalement dus à RxJS. Initialement, j'avais prévu de concevoir le projet de manière à ce que l'application soit réactive sur l'ensemble du stack. Cependant, après une discussion éclairante avec mon rapporteur, M. Noel, nous avons décidé de construire le projet de manière classique tout en envisageant l'ajout de la réactivité dans une phase future.

8.5 Defits lors du developement

Lors du develepement du projets plusieurs defit on du etre surmonter

8.5.1 L'algorithme des dettes

Le développement d'un algorithme capable de répartir les différents coûts d'un événement entre les utilisateurs s'est avéré être un défi. Non seulement il fallait que les différentes dépenses existantes à un instant T soient réparties entre les différents participants, mais il fallait aussi permettre aux participants de continuer à ajouter des dépenses, même une fois qu'un utilisateur a remboursé ses dettes.

Au départ, l'algorithme était plutôt simple : récupérer toutes les dépenses pour un événement donné dans la base de données, calculer le total des dépenses pour l'événement et créer une dette par participant, dont la valeur est le total divisé par le nombre de participants.

Cependant, cet algorithme n'était pas satisfaisant pour plusieurs raisons, principalement le manque de possibilité de spécifier à qui les dettes doivent être remboursées. J'ai donc décidé de changer d'approche. Au lieu de calculer le total des dépenses pour un événement, nous allons calculer la balance totale de chaque participant.

Pour cela, lors du calcul des dettes, on crée une carte associant un participant à une balance. Une fois que la balance de chaque participant est créée sur la base de toutes les dépenses, on parcourt cette carte afin de créer des paires de participants. L'objectif est de trouver le participant avec la balance la plus positive et celui avec la balance la plus négative. Une fois que l'on a cette paire, on peut créer une dette entre ces deux utilisateurs de sorte à ce que l'un ou les deux voient leur balance revenir à zéro. Après avoir créé cette dette, on met à jour la balance des deux utilisateurs. On continue ainsi jusqu'à ce que toutes les balances des utilisateurs valent zéro

À première vue, cette stratégie fonctionne. Cependant, comme je l'ai découvert plus tard dans le développement, elle présente plusieurs problèmes majeurs. Premièrement, cet algorithme ne permet pas aux utilisateurs de marquer une dette comme remboursée et que cela persiste au travers des différents calculs. Deuxièmement, les participants ne peuvent pas créer de dépenses entre eux et donc un participant qui aurait par exemple participé à un covoiturage ne paierait pas plus qu'un participant qui se serait déplacé par ses propres moyens. Troisièmement, une dépense est obligatoirement répartie entre tous les participants.

Pour remédier à ces problèmes, il a fallu modifier les dépenses afin de permettre de sélectionner un acheteur et un bénéficiaire. Désormais, chaque achat dans la liste des courses, par exemple, crée une dépense entre l'acheteur et chacun des bénéficiaires. Un champ permettant de conserver la balance d'un utilisateur a été ajouté à l'entité EventToUser afin d'optimiser les calculs (plus besoin de parcourir toutes les dépenses pour pouvoir générer les dettes). Pour gérer le remboursement des dettes, nous profitons de l'ajout d'un acheteur à une dépense. Lorsqu'un participant marque une dette comme remboursée, une dépense de valeur opposée est créée entre le créancier et le débiteur afin de contrebalancer la dette. Cette dernière est alors supprimée.

La solution mise en place permet de gérer efficacement le remboursement des dettes et de répartir les dépenses parmi les consommateurs, optimisant ainsi les calculs de remboursement. Grâce à une modification récente, les participants peuvent désormais diviser une dépense de manière inégale ou choisir de ne pas la répartir entre tous les participants, ajoutant une couche supplémentaire de flexibilité à cet algorithme.

8.5.2 le cluster kubernetees

La conception et la mise en place d'un cluster Kubernetes ont probablement été les défis majeurs de ce projet. L'apprentissage des différents outils, la conception du cluster et son déploiement ont été des étapes cruciales. Au début du projet, j'étais totalement novice avec Kubernetes et Ansible n'était pour moi qu'un nom.

J'ai donc dû m'initier au fonctionnement de Kubernetes et aux différents outils pouvant être utilisés en parallèle. Mes recherches m'ont conduit à découvrir une immense communauté de passionnés qui ont appris à utiliser Kubernetes et ont créé de la documentation à ce sujet pour le plaisir ou pour des fins professionnelles. Techno Tim, l'un d'entre eux, a créé une série de vidéos où il partage les différentes configurations mises en place dans son micro datacenter qu'il gère chez lui. C'est en découvrant ces vidéos que j'ai commencé à apprendre comment configurer Kubernetes et utiliser Ansible.

Grâce à ces nouvelles connaissances, j'ai décidé de mettre en place sur l'un de mes serveurs un cluster qui, grâce à la virtualisation, serait hautement disponible. En réalité, seule la redondance du cluster Kubernetes est intégrée ; il n'y a pas de redondance au niveau matériel car je n'avais pas les moyens de l'implémenter.

J'ai donc commencé par créer des machines virtuelles (VM) sur mon hyperviseur (Proxmox). Après une mauvaise manipulation, j'ai décidé de recommencer depuis le début. Je me suis alors rendu compte que créer les VM manuellement n'était pas l'idéal, mais je ne voulais pas mettre en place un système de MaaS (Metal as a Service) qui aurait trop sollicité mon serveur. J'ai donc choisi une solution intermédiaire : je n'ai pas complètement automatisé la configuration et le provisionnement de machines virtuelles, mais j'ai cherché à simplifier considérablement le processus.

C'est alors que j'ai découvert deux outils formidables : les templates de VM qemu et CloudInit. Grâce à ces outils, je pouvais facilement créer des machines virtuelles avec mes paramètres, et modifier ceux-ci via CloudInit avant même de démarrer la machine virtuelle.

Fort de cette nouvelle découverte, j'ai mis en place sept VM dédiées au cluster Kubernetes : trois serviront de plans de contrôle et quatre de nœuds d'exécution.

Le projet nécessitant une infrastructure d'hébergement et étant conteneurisé, tout s'alignait parfaitement.

Après des jours d'expérimentation manuelle avec Kubernetes via kubectl, Lens et Portainer, j'ai décidé de ne plus vouloir d'un cluster unique et fragile comme un flocon de neige. Je me suis tourné vers Ansible, un outil dont j'avais entendu beaucoup de bien, afin de définir mon infrastructure en tant que code.

J'ai cherché comment configurer un cluster Kubernetes via Ansible et j'ai découvert l'existence d'une communauté de passionnés qui ont créé et publié en open-source des playbooks Ansible complets permettant de mettre en place un cluster.

Dans cette communauté, j'ai retrouvé Techno Tim qui a, lui aussi, contribué en adaptant un playbook existant afin de déployer un cluster hautement disponible. J'ai donc décidé de partir de son projet pour construire mon infrastructure. J'ai ajouté au script original cert-manager, Traefik, Rancher, Longhorn et ArgoCD.

L'ajout de cert-manager et Traefik a été plutôt simple, mais l'intégration de Rancher s'est avérée plus complexe pour plusieurs raisons.

Premièrement, mon serveur était très limité en ressources pour ce type de tâche, encore plus à cause de la virtualisation et du niveau de redondance nécessaire à la haute disponibilité. Deuxièmement, la version de Rancher que j'essayais d'installer à ce moment-là n'était tout simplement pas compatible avec K3S (l'implémentation de Kubernetes que j'avais décidé de déployer). Enfin, après avoir changé de version de Rancher et sélectionné une version compatible avec K3S, je me suis rendu compte que cette version n'était pas compatible avec la version spécifique de K3S que j'avais choisie. J'ai donc dû changer une fois de plus de version de Rancher. J'ai finalement pu finaliser la mise en place de mon cluster.

Deux mois plus tard, j'ai voulu apporter une modification au cluster, plutôt simple : j'ai voulu ajouter un certificat TLS. Au lieu de simplement modifier le cluster existant, j'ai décidé de tirer profit de mon IaC pour pérenniser ce changement. J'ai donc modifié les détails dans le playbook et je l'ai lancé. Malheureusement, le cluster ne répondait plus...

En fait, en raison d'un manque chronique de ressources, certains processus essentiels à Kubernetes n'avaient pas pu être exécutés à temps, ce qui a entraîné l'arrêt de son fonctionnement. J'ai donc décidé de recréer le cluster à partir de zéro. Cependant, mon

playbook, qui deux mois plus tôt fonctionnait parfaitement, ne fonctionnait plus du tout.

Après de longues heures de débogage, je n'arrivais pas à trouver la cause du problème, même après être revenu à la dernière version du playbook qui avait servi à déployer le cluster. Impossible de redéployer le cluster. Rien n'avait changé dans le reste de l'infrastructure pourtant.

Le playbook affichait toujours la même erreur : "helm : jetstack repository not found". Le problème est que la commande d'ajout de ce référentiel est bien exécutée et sans erreurs et que lorsque je teste manuellement, le référentiel est bien ajouté à Helm et disponible.

J'ai consacré de nombreuses heures à essayer de résoudre ce problème, mais sans succès. C'était frustrant et déroutant, car je n'avais apporté aucun changement majeur au playbook depuis son dernier déploiement réussi.

Face à ce blocage, j'ai été contraint de reconsidérer l'approche technique pour le déploiement de mon projet. J'ai donc décidé de mettre Kubernetes de côté et de me concentrer sur Docker-compose. Bien que ce ne soit pas la solution initiale envisagée, Docker-compose m'a permis de poursuivre le développement du projet tout en conservant l'aspect conteneurisé de l'application.

8.5.2.1 Le choix de Docker-compose

Docker-compose s'est révélé être une alternative efficace à Kubernetes, grâce à sa simplicité d'utilisation et à sa moindre consommation de ressources. Il m'a permis de définir et de gérer plusieurs conteneurs comme un ensemble de services interconnectés, ce qui correspondait parfaitement aux besoins de mon projet. De plus, la préparation d'un fichier docker-compose était prévue dans le cadre du projet pour accompagner les clients potentiels qui ne seraient pas en mesure de déployer le projet sur Kubernetes mais qui pourraient aisément l'exploiter via Docker-compose.

La migration vers Docker-compose s'est faite sans difficultés majeures, étant donné ma familiarité avec Docker et le fait que mon application était déjà conteneurisée. Quelques ajustements dans le fichier docker-compose.yml ont été suffisants pour rendre mon application pleinement opérationnelle.

Ce revirement de situation m'a libéré du temps pour me concentrer sur d'autres aspects du projet, tels que le développement de nouvelles fonctionnalités et l'amélioration de l'expérience utilisateur. Malgré une certaine déception de ne pas avoir pu implémenter Kubernetes comme initialement prévu, je reste satisfait du résultat final. Docker-compose a parfaitement répondu à mes besoins et m'a permis de mener à bien mon projet.

Pour conclure, bien que la tentative d'implémentation de Kubernetes ait été un défi majeur qui n'a pas abouti comme je l'espérais, ce projet reste une riche expérience d'apprentissage. J'ai pu approfondir ma compréhension des outils de déploiement et d'orchestration de conteneurs et développer des compétences précieuses en matière de débogage et de résolution de problèmes. Malgré les difficultés rencontrées, je suis fier du travail accompli et je reste déterminé à explorer davantage Kubernetes dans le cadre de futurs projets.

8.5.3 le client graphql

Comme, je l'ai mentionné dans les sections précédentes, le développement du front-end a été réalisé à l'aide d'Angular. Ce dernier doit interagir avec le back-end par le biais d'une API GraphQL. C'est pour cela que j'ai choisi d'utiliser Apollo Angular.

Pour configurer Apollo Angular, il est nécessaire de créer un fichier graphql.module.ts contenant les configurations minimales recommandées par la documentation. Afin de permettre l'authentification auprès du back-end via JWT, un header doit être configuré avec le token.

Cependant, j'ai rencontré un problème : le token est géré par l'un de mes services et la configuration d'Apollo se fait hors du champ d'action de l'injection de dépendance d'Angular. De plus, en cas d'erreur de requête, je souhaitais pouvoir capturer cette erreur et agir différemment en fonction du message reçu. Par exemple, une erreur 400 devrait simplement être consignée dans la console, tandis qu'une erreur 401 devrait rediriger l'utilisateur vers la page de login.

Malheureusement, je me trouvais dans l'impossibilité de récupérer mes tokens et de rediriger mes utilisateurs vers la page appropriée grâce au routeur Angular.

Finalement, j'ai contourné ces problèmes en utilisant directement les éléments JavaScript (window) et le local storage. Cela m'a permis de récupérer le token et de gérer les redirections adéquates lors des erreurs.

9 Gestion de version et Intégration et Déploiement Continu

9.1 Gestion de version

La gestion de version, ou versioning, est une pratique essentielle dans tout projet de développement. Pour ce projet, nous avons utilisé Git comme système de gestion de versions et GitHub pour héberger notre code source.

Au départ du projet, j'ai choisi d'utiliser une seule branche pendant la majeure partie de la première phase. Cette approche était suffisante pour répondre aux besoins du projet à ce stade, permettant de maintenir une simplicité de gestion.

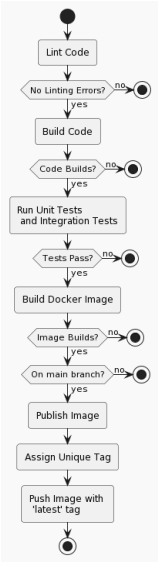
Cependant, une fois le projet publié sur DockerHub et la mise en place du CI/CD, il est devenu évident que l'utilisation d'une seule branche n'était plus la meilleure option. J'ai alors créé une branche de développement distincte. Cette branche a permis de tester et de sauvegarder le code sans perturber la version en production.

Pour la rédaction de ce rapport, j'ai également créé une branche spécifique. Cette approche a permis de travailler sur le rapport sans interférer avec le code de l'application.

Pour la gestion de l'infrastructure du projet, un second repository GitHub a été utilisé. Ce choix a été motivé par le fait que l'infrastructure du projet était basée sur un autre projet existant que j'ai forké. Ce repo distinct pour l'infrastructure a permis de maintenir une séparation claire entre le code de l'application et la gestion de l'infrastructure.

9.2 Intégration et déploiement continu (CI/CD)

Pour l'intégration continue (CI), j'ai mis en place des tests automatisés sur une grande partie des composants du backend et du frontend.



Workflow d'intégration continue

Le workflow de CI est le suivant :

- Lint du code : vérification des erreurs de syntaxe.
- Build du code : si aucune erreur de linting n'est détectée, je peux construire le code.
- Tests unitaires et d'intégration : si le code se construit correctement, je lance les tests.
- Construction de l'image Docker : si les tests passent, je construis l'image du conteneur.
- Publication de l'image : si la construction de l'image réussit, je la publie sur DockerHub (seulement si je suis sur la branche principale).

Pour le déploiement continu (CD), j'utilise un conteneur Watchtower qui, via le socket Docker, vérifie régulièrement quelle version du conteneur est utilisée et si elle correspond bien à celle demandée. Watchtower vérifie également la signature du conteneur en plus de son tag, ce qui lui permet de mettre à jour un conteneur avec le tag "latest" lorsque l'image change. Watchtower compare régulièrement la version courante des conteneurs contre celle hébergée sur DockerHub et si elles ne correspondent pas, alors le conteneur est mis à jour.

9.3 Conclusion

La gestion de version a été un aspect crucial de ce projet. L'adoption d'une approche adaptative, passant d'une seule branche à plusieurs branches en fonction des besoins du projet, a prouvé son efficacité. De même, l'implémentation des pratiques de CI/CD a grandement amélioré le flux de travail et la qualité de l'application, rendant le processus de développement plus fluide et fiable.