681 Rue de la Passerelle - BP 72
38400 Saint-Martin-d'Hères CEDEX
https://ensimag.grenoble-inp.fr/

Manon Andre - Julien Bourseau - Ibrahim Taib -
Paul Vernin - Axelle Yriarte
prenom.nom@grenoble-inp.org
Étudiant Ensimag - Année 2

Analyse Conception Objet Logiciel
Construction d'Application Web

Documentation WEB

Cours : Projet CAWEB-ACOL
Année universitaire 2022 - 2023

15 mai 2023

Table des matières

1 Stack

Le projet est architecturé avec la stratégie de monorepo. Contrairement à l'approche classique qui consiste à définir un repository git par projet, ici on aura tout dans le même repository. C'est une architecture qui collait bien avec le fait qu'on ait à rendre le code d'en un seul repository et plutôt facile à mettre en place grâce à pnpm et turborepo pour un projet javascript/typescript.

1.1 Pnpm

Pnpm utilise un système de liens symboliques pour les dépendances partagées, ce qui permet de réduire l'espace disque utilisé par rapport à npm, c'est un outil beaucoup plus optimisé qui évite d'avoir 15 fois react d'installer sur sa machine. Il utilise aussi ce système de lien symbolique entre les différentes versions d'un même package lorsque les fichiers sont identiques d'une version à l'autre.

Un autre point important, c'est la possibilité de gèrer des workspaces qui est l'outil principal permettant la mise en place d'un monorepo. Un workspace correspond à un projet/application ou à un package "interne". Par exemple dans notre projet, nous avons 2 applications correspondant au backend et frontend qui sont dans apps/ et plusieurs packages dans packages/ qui sont utilisés dans les deux applications. On peut voir que dans le package.json de apps/api/, il y a une dependencies qui s'appelle "database" ou encore "config-eslint-wolfo". C'est grâce au système de workspace, ce qui permet une plus grande modularité du code.

Par exemple si on décide de faire un nouveau backend avec graphql, on aura juste à créer l'app dans apps/ et ajouter le package database pour avoir accès à la base de données. Un autre exemple c'est le package types qui définit les types utilisés des deux côtés de l'api (Frontend et Backend). Vu que les deux apps importent le même package interne, on limite grandement les problèmes de typages entre les deux applications.

1.2 Turborepo

Turborepo est un package qui s'intégre à une architecture monorepo.

Turborepo est un outil open-source qui permet de gérer des monorepos de manière efficace et optimisée. Contrairement à d'autres outils de gestion de monorepos, Turborepo se concentre sur la performance et la rapidité en utilisant des techniques comme la parallélisation des tâches et le caching pour accélérer les opérations de développement.

Par exemple lancer la commande turbo run lint va lancer la commande pnpm lint dans tous les projets en parallèle. Dans notre projet on aura donc eslint qui check le code du backend en parallèle d'une autre instance d'eslint pour le frontend et de la cli de prisma qui format et validate le schema de la base de données.

Le package utilise aussi un système de caching qui permet de ne pas relancer une pipeline s'il n'y a pas eu de changement sur les fichiers. Très pratique lorsqu'on a des git hooks qui lint le code avant de commit, ce qui évite de lint le(s) projet(s) qui n'ont pas été changer entre deux commits.

1.3 Tanstack query

Tanstack Query est une librairie de state management. vidéo qui explique tout en 100s, c'est une librairie très adapté pour un projet comme celui-ci où le state de l'application est gérer en majeur partie côté Backend. Exemple : À la place de stocker la game avec un useContext qu'on partage entre les pages (game, vote, power), on va utiliser le même `useQuery` avec la même clée de cache et la librairie va se charger de cache automatiquement les données et de ne pas refetch inutilement le game (ou de le faire uniquement si on décide d'invalider la clée de cache).

Du coup à la place d'utiliser useContext

1.4 Expo router

Expo router apporte les meilleurs concepts de routage du web aux applications iOS et Android natives. C'est une librairie construite pardessus la librairie React Navigation qui permet une vraie navigation native. Le concept est que chaque fichier dans le répertoire de l'application devient automatiquement une route dans la navigation mobile. On peut générer des routes dynamiques (`[idGame].tsx`).

C'est un concept qu'on retrouve dans de nombreux frameworks récent comme Next.js et Nuxt.js.

1.5 Prisma

On a fait le choix de remplacer Sequelize par Prisma, vidéo qui explique tout en 100s. C'est un ORM qui a une developper experience bien plus agréable. L'ORM permet de communiquer avec la base de données mais elle aussi de générer des types TypeScript correspondant aux modèles de la BDD, ce qui fait qu'on ne peut pas faire d'erreur de typo lorsqu'on utilise prisma car toutes nos requêtes sont strictement typées.

2 Installation

En premier temps il est important d'installer pnpm qui est utilisé dans le projet et dans les scripts des différents package.json.
npm install -g pnpm

Une fois installé, on lance la commande pnpm install. Grâce aux différentes configuration comme la définition des workspaces dans package.json et le fichier .npmrc, le gestionaire de package va être capable d'installer les packages des différents sous projets (`api, wolfo, database, types`) automatiquement. Il va installer ces packages dans le `node_module` à la source, ce qui permet d'éviter les packages dupliqués.

2.1 Database

Durant tout le projet, on a utilisé postgres et pour ne pas s'embêter à l'installer sur nos pcs, de configurer le serveur en local, on a mis le serveur postgres dans une image docker.

- Aller dans le dossier `packages/database`
- Copier/coller le `.env.exemple` dans `.env`
- Lancer la commande `pnpm docker:init` pour créer l'image docker
- Lancer la commande `pnpm db:init` qui va créer les tables dans la BDD à partir du schéma dans `prisma/scheam.prisma`

2.2 Backend

Il faut copier/coller le fichier `.env.exemple` dans `.env` dans le dossier `apps/api/` , pour lancer le mode dev, on fait `pnpm dev` depuis la racine et pour build c'est `pnpm build`.

2.3 Frontend

Pour le Frontend, il faut juste se mettre dans le dossier `apps/wolfo` et lancer `pnpm start` et choisir quel type d'application utilisé sachant qu'elle est fonctionnelle sur web, android et ios.

3 Testing

On peut lancer les tests depuis la racine avec `pnpm test:backend` et `pnpm test:frontend`. L'affichage n'est pas idéal donc c'est mieux d'aller dans les dossiers des projets.

Les tests ne sont pas très poussé, on peut le constater par le coverage relativement bas. On a malheureusement pas pris le temps de faire plus de tests.
Néanmoins il est important de prendre en compte qu'en utilisant TypeScript, Prisma et un package "types", nous avons drastiquement réduit les risques d'erreurs ou de bugs du à des problèmes de typages, typo, conversion, etc...
Il y a juste les tests de logique qu'on a testé "à la main".

3.1 Test backend

Les tests sont fait avec jest et supertest. Avec l'option `--coverage` on génère un rapport de coverage qu'on a hébergé sur les gitlab pages ici et on a généré des badges à partir de ce coverage en utilisant la librairie `jest-coverage-badges`.

3.2 Test frontend

Les tests frontend sont fait avec Cypress. Le rapport de test est généré par cloud.cypress et se trouve à cette adresse et le badge est aussi généré par cloud.cypress.

Cependant les rapports et le badge sont tous à `fail` car on a pas réussi à lancer les tests frontend en CI/CD. Après de nombreuses tentatives et le fait qu'on utilise metro et non webpack pour créer le bundle web (car certaines de nos librairies ne fonctionnent pas avec webpack). On ne peut donc pas utiliser `npx expo export:web`. On est limité à utiliser `npx expo web` sauf qu'avec cette commande, le bundle est généré que lorsqu'on visite la page. Du coup ça fait qu'au premier `it()` de nos tests, ça va trigger le bundling qui va prendre un peu de temps. Donc les 4-5 premiers `it()` vont fail et les `it()` suivant dépendant des `it()` ayant fail vont aussi fail.

Néanmoins si on lance les tests en local depuis la racine du projet avec `pnpm test:frontend` (interface graphique) ou `pnpm test:frontend:ci` (terminal). On peut aussi lancer les tests depuis `/apps/wolfo` mais il faut penser à lancer une instance du backend avec `pnpm dev` dans `/apps/api`.

3.3 Rendu

Le backend est disponible à cette adresse. La doc ici.
Les badges de coverage backend et frontend sont affichés sur la page gitlab. Pas de badge eslint parce que de toute façon la librairie husky empêche de commit du code si eslint trouve des erreurs.
Les rapports de coverage sont disponibles lorsqu'on clique sur les badges.

Le code est sur la branche `main` et le code déployé sur scalingo est sur la branche `scalingo` (quelques modifications étaient nécessaires).
Quand on lance l'application en local, elle va utiliser l'API de scalingo, si on souhaite changer et utiliser localhost, il faut modifier `baseUrl` du fichier `apps/wolfo/src/api/api.ts`. Pareil pour les chatroom, il faut modifier l'url dans `apps/wolfo/src/app/games/[gameId]/chatroom/[chat].tsx`.
Normalement les websockets fonctionnent avec scalingo sauf que scalingo change le port dynamiquement quand on reload, on pense pas que ça arrivera car on va pas push après le rendu mais si jamais, on peut récupérer le port à cette adresse https://wolfo-backend.osc-fr1.scalingo.io/port