Dossier Remise de mi-Parcours projet Web

Boudrouss Réda

Groupe 2 n°28712638

Mouine Youssef

Groupe 2 n°28706232

Sorbonne Université
France

Contents

Introduction **1**

Outils utilisés **1**

Structure générale du projet **2**

le dossier Helper **2**

Coté client dans `app` (React) **2**
Structure du dossier `app` . . . 3
`components` . . . 3
Les Pages . . . 4

Base de donnée et tables **5**
Table . . . 5
User . . . 5
Post . . . 6
Likes . . . 6
Follows . . . 6
Session . . . 6
Image . . . 6

Coté Serveur dans `pages/api` (Nextjs) **7**
`image` . . . 7
`user` . . . 8
`post` . . . 8

Introduction

Ce projet est un projet universitaire qui a pour but de créer un site web comparable à Twitter. Ce projet est réalisé dans le cadre de l'UE technologie du web (LU3IN017) à Sorbonne Université.

Une version en ligne est disponible à l'adresse suivante : https://birdy.rboud.com.

Outils utilisés

Ce projet est réalisé avec les frameworks et outils suivants :

- Typescript pour le développement. Typescript est équivalent au Javascript mais avec des types statiques. Cela permet de développer plus rapidement et de trouver plus facilement les erreurs.
- React pour la création de l'interface utilisateur coté client.
- Next.js pour la gestion de la backend et de la génération du site web. Nextjs offre la possibilité aussi de prégénérer certain composant coté serveur pour améliorer les performances. Il s'occupe aussi du routing et de la génération des pages.

- Prisma pour la gestion de la base de données. Prisma est un ORM qui permet de gérer la base de données, ses migrations, ses requêtes et est particulièrment adapté au Typescript car il permet de faciliter le typage.

Structure générale du projet

Au root du projet vous trouverez plusieurs dossiers *(ignorons les fichiers de configuration)* :

- `src` : contient le code source du projet
- `public` : contient les fichiers statiques du projet
- `prisma` : contient les fichiers de configuration de la base de données (et la base de données si sqlite est utilisé). Le fichier `schema.prisma` contient la définition de la base de données.

Concernant le dossier `src` il contient plusieurs sous-dossiers :

- `app` : contient le coté ""client"" de l'application, toutes les pages et composants sont dans ce dossier.
- `pages/api` : contient le coté serveur et plus précisément l'API.
- `helper` : contient toutes fonctions utilitaires.

le dossier Helper

Ce dossier inclut plusieurs fichiers et fonctions utilitaires utilisés dans le projet (nous ne rentrerons pas dans les détails des fonctions).

- `constants.ts` : contient les constantes utilisées dans le projet (certaines peuvent être modifiées).
- `fetchWrapper.ts` : contient les fonctions qui englobe la fonction `fetch` par défaut de javascript. Ces fonctions permettent de simplifier l'utilisation de `fetch` et de gérer les erreurs.
- `userService.ts` : contient les fonctions utilisées pour les services de l'utilisateur (login, logout, register, création de post, etc…).
- `APIwrapper.ts` : contient les objets principaux utilisés pour communiquer avec l'API. Une classe `APIUser` et `APIPost` qui servent à la fois d'object contenant des informations que des fonctions pour communiquer avec l'API.
- `backendHelper.ts` : contient des fonctions utilisées et conçu pour être utilisé en backend.
- `ImageHelper.ts` : contient des fonctions utilisées pour gérer les images.
- `cookieHelper.ts` : contient des fonctions utilisées pour gérer les cookies.

Coté client dans `app` (React)

Un des avantages de Next13+ et le dossier `app` est le fait qu'il s'occupe pour nous du routing de l'application de part la structure du dossier. Par exemple le fichier `app/login/page.tsx` est la page atteignable à l'adresse `/login`. Pour plus d'information, voir la documentation de Next.js

Structure du dossier `app`

Dans le dossier `app`, tout les dossiers serv à déterminer le chemin de la page. Excepté le dossier `components` qui contient les composants réutilisables.

`components`

Les components utilisés dans la pages sont : (leurs props seront détaillés si nécessaire, pour une liste complète voir l'image ci-dessous)

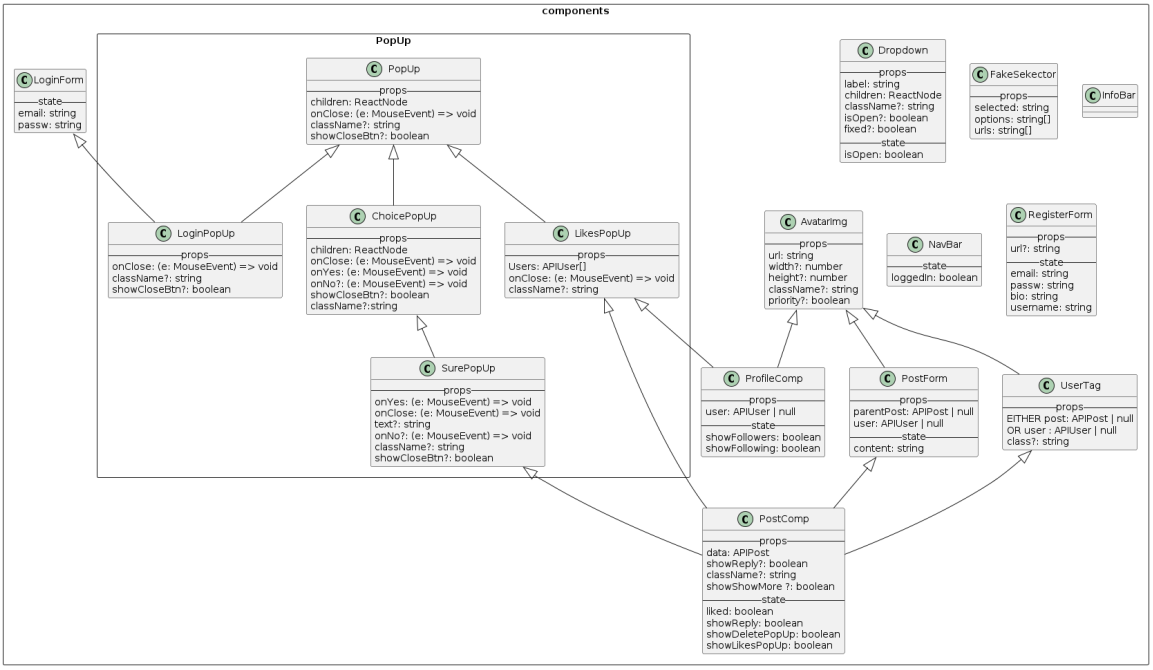

Figure 1: figure des dépendances des components

- `AvatarImg` : Le composant de l'avatar de l'utilisateur. Il prend en paramètre l'url de l'image et la taille. Il est possible de passer une classe css pour le modifier.
- `Dropdown` : Liste déroulante, affiche/cache les enfants quand on clique dessus. Prend en paramètre le label affiché, les composants react à afficher et l'état par défaut. (utilisé dans la page `settings`)
- `FakeSelector` : Composant qui donne l'impression d'un choix fluide mais est en réalité que des liens. Prend en paramètre les choix à afficher ainsi que leurs URLs et le choix séléctionné (Solution temporaire)
- `InfoBar`: Barre d'information affiché à droite sur la version desktop. Vide pour l'instant mais nous comptons y ajouter la bar de recherche et des recommandations.
- `LoginForm`: Formulaire de connexion. Prend rien en paramètre.
- `NavBar`: Barre de navigation affiché à gauche sur la version desktop. Contient les liens vers les pages principales de l'application et le nom de l'utilisateur connecté.

- `PopUp` : Composant qui affiche un composant prit en paramètre au milieu de l'écran. Prend en paramètre le composant à afficher et une fonction exécuté en cas de fermeture. Certain composant "hérite" de celui là.
  - `ChoicePopUp` : Un PopUp qui affiche le composant prti en paramètre suivi par un choix "oui" ou "non". Prend en paramètre les mêmes composant que `PopUp` ainsi que les fonctions à exécuter en cas de choix.
  - `SurePopUp` : hérite de `ChoicePopUp` et affiche un message de confirmation. Prend en paramètre les mêmes composant que `ChoicePopUp` ainsi que le message à afficher.
  - `LoginPopUp` : Affiche un PopUp du form de connexion. Prend en paramètre les mêmes composant que `PopUp`.
- `PostComp` : Composant d'un post. Prend en paramètre le post à afficher.
- `PostForm` : Formulaire de création de post. Prend en paramètre l'utilisateur connecté et le post parent (si il y en a).
- `ProfileComp` : Composant d'un profil. Prend en paramètre l'utilisateur à afficher. Affiché en haut dans la page `profile`.
- `RegiserForm` : Formulaire d'inscription. Prend en paramètre optionnellement l'url de l'API à utiliser.
- `UserTag` : Composant d'un tag utilisateur. Prend en paramètre l'utilisateur à afficher OU le post auquel on veut générer un UserTag.

Les Pages

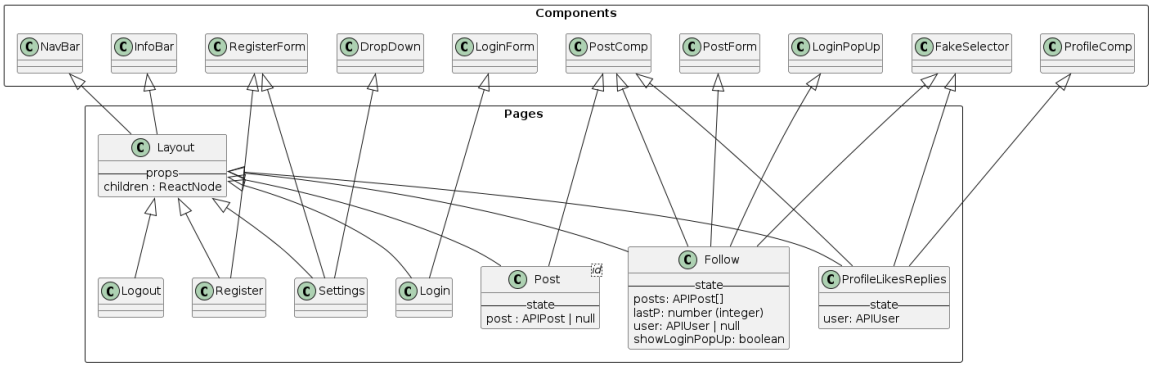

Figure 2: figure des dépendances des pages

Le path d'une page est déterminé par le chemin des fichiers `page.tsx`. Par exemple, le fichier `app/login/page.tsx` est la page atteignable à l'adresse `/login`. Les fichiers `layout.tsx` sont des composants qui servent à définir le layout de la page, que toutes les pages héritent. Pour le coup, dans le fichier `layout.tsx` nous avons l'`InfoBar` et la `NavBar` qui sont affichés sur toutes les pages.

- `app/page.tsx` : Page d'accueil à l'url `/`. Affiche les posts récents principalement, et est le premier point d'accés pour un utilisateur pour poster.

- `app/follow/page.tsx` : Équivalente à la page d'accueil mais affiche les posts des utilisateurs que l'utilisateur connecté suit. L'url est `/follow`.
- `app/login/page.tsx` : Page de connexion. L'url est `/login`.
- `app/register/page.tsx` : Page d'inscription. L'url est `/register`.
- `app/logout/page.tsx` : Page de déconnexion. L'url est `/logout`.
- `app/settings/page.tsx` : Page des paramètres, permet de modifié son compte, mettre une photo de profile ou de converture, ou supprimer son compte. L'url est `/settings`.
- `app/post/[id]/page.tsx` : Page où on affiche le post <id>, le post auquel il répond directement et les réponses directes au post (id étant un nombre > 0). Nous traitons les posts avec une profondeur de 1 pour l'instant. L'url est `/post/<id>`.
- `app/profile/` (le profil est séparé en plusieurs pages, c'était une solution facile utilisant le `FakeSelector` à terme ceci dit nous aimeront trouver une solution plus propre)
  - `app/profile/[id]/page.tsx` : Page où on affiche le profile de l'utilisateur <id> avec ces derniers posts, excluant les réponses (id étant un nombre > 0). L'url est `/profile/<id>`.
  - `app/profile/[id]/replies/page.tsx` : Page où on affiche le profile de l'utilisateur <id> et ces derniers postes en incluant les réponses. L'url est `/profile/<id>/replies`.
  - `app/profile/[id]/likes/page.tsx` : Page où on affiche le profile de l'utilisateur <id> et les postes qu'il a liké. L'url est `/profile/<id>/likes`.

Base de donnée et tables

En développement, nous utilisons une base de donnée sqlite. Pour la production, nous utilisons une base de donnée postgresql. Pour la gestion de la base de donnée, nous utilisons prisma qui fait les migrations et les requêtes pour nous.

Table

Notre table est décrite dans le fichier `schema.prisma` qui se trouve dans le dossier `prisma`. N'hésitez pas à regarder le fichier pour plus de détails.

User

La table User contient les informations de l'utilisateur.

Le mot de passe est stocké en hash avec bcrypt. Nous n'utilisons pas de salt pour l'instant, c'est un ajout que l'on compte faire.

id	username	email	password	bio	createdAt	nbFollowers	nbFollowing	nbLikes
int	string	string	string	string	date	int	int	int

Post

La table Post contient les informations d'un post, que ça soit un post normal ou une réponse.

id	content	createdAt	authorId	replyId	nbLikes	nbReplies
int	string	date	int	?int	int	int

Likes

La table Likes contient les informations d'un like, c'est une association entre un utilisateur et un post.

createdAt	userId	postId
date	int	int

Follows

La table Follows contient les informations d'un follow, c'est une association entre un utilisateur et un autre utilisateur.

createdAt	followerId	followingId
date	int	int

Session

La table Session contient les informations d'une session. Une session est un token généré qui permet de s'authentifier sur le site. Il a une date d'expiration et un utilisateur peut en avoir plusieurs.

id	createdAt	expires	userId
string	date	date	int

Image

La table Image contient les informations d'une image. Une image est un fichier qui est stocké sur le serveur et qui est associé à un utilisateur.

Nous différencions après les images de profil et les images de post dans une autre table. Les images sont stockés dans le dossier `public/uploads` et sont donc accésible via l'url `/uploads/<imageId>`.

id	createdAt	userId
string	date	int

postImage La table postImage est une association entre une Image et un Post.

postId	imageId	userId
int	string	int

ppImage De même, une association entre une Image et un User.

userId	imageId
int	string

Coté Serveur dans `pages/api` (Nextjs)

Un peu comme le coté client, les urls des API du coté serveur sont définies par le chemin des fichiers à partir de `pages`. Par exemple le fichier `pages/api/user/login.ts` correspond à l'url `/api/user/login`. Les fichiers `index.ts` sont utilisés pour le root de l'url.

Tout les API retourne une réponse json avec le format suivant :

```
{
  isError: boolean,
  status: number,
  message: string,
  data: "\<selon l'API\>",
}
```

Les types que vous verrez ci-dessous sont les équivallents json des tables de la base de données.

Nous décrirons après les types et les méthodes de requêtes dans la forme suivante : `<format de la requete> Method <type de data dans la réponse>`

`image`

- `/api/image/index.ts`; url : `/api/image`
  - `GET<HTML>` : retourne de l'html, un form pour pouvoir poster une image (utilisé à des fins de débug seulement <!>)
  - `<FORMDATA/IMG>POST<string>` : permet de poster une image. Retourne un objet json avec l'url de l'image (qui est aussi son id). Le body de la requête doit être un form-data avec un champ `image` qui contient l'image. Nous vérifions les cookies pour savoir qui est l'auteur de l'image.
- `/api/image/all.ts`; url : `/api/image/all`
  - `GET<Image[]>` : retourne une liste de toute les images stocké et leur auteurs.

user

- `/api/user/all.ts`; url : `/api/user/all`
  - `GET<User[]>` : retourne une liste de tout les utilisateurs.
- `/api/user/login.ts`; url : `/api/login`
  - `POST<{session: Session, user: User}>` : permet de se connecter. Retourne un objet json avec la session et l'utilisateur. Vérifie si le mot de passe correspond au hash stocké dans la base de donnée et créé une session si c'est le cas.
- `/api/user/register.ts`; url : `/api/register`
  - `<username, email, password, bio : string>POST<User>` : permet de s'enregistrer, retourne l'utilisateur créé.
- `/api/user/all.ts`; url : `/api/user/all`
  - `GET<User[]>` : retourne une liste de tout les utilisateurs.
- `/api/user/[id]/index.ts`; url : `/api/user/:id`
  - `GET<User>` : retourne l'utilisateur avec l'id `:id`.
  - `<username, email, password, bio : string>PUT<User>` : permet de modifier l'utilisateur avec l'id `:id`. Retourne l'utilisateur modifié.
  - `DELETE<User>` : permet de supprimer l'utilisateur avec l'id `:id`. Retourne l'utilisateur supprimé. (<!> pas encore implémenté)
- `/api/user/[id]/follow.ts`; url : `/api/user/:id/follow`
  - `<author:int>POST<Follow>` : permet de suivre l'utilisateur avec l'id `:id`. Retourne les id de l'utilisateur et de l'auteur.
- `/api/user/[id]/unfollow.ts`; url : `/api/user/:id/unfollow`
  - `<author:int>POST<Follow>` : permet de ne plus suivre l'utilisateur avec l'id `:id`. Retourne les id de l'utilisateur et de l'auteur.
- `/api/user/[id]/profile.ts`; url : `/api/user/:id/profile` (pas le meilleur nom)
  - `<FORMDATA/IMG>POST<string>` : Modifie l'image de profile de l'utilisateur avec l'id `:id`. Retourne l'id de l'image.

post

- `/api/post/recent.ts`; url : `/api/post/all`
  - `<all, n, skip, replies, follow : QUERIES>GET<Post[]>` : retourne la liste des postes. Les paramètres sont optionnels et sont décrits dans le fichier `pages/api/post/recent.ts`.
- `/api/post/create.ts`; url : `/api/post/:id/create` (nous devrions regrouper ça dans le fichier `index.ts`)
  - `<author:int, content:string>POST``<Post‘>` : permet de créer un nouveau post. Retourne le nouveau post créé.

- `/api/post/[id]/index.ts`; url : `/api/post/:id`
  - `GET<Post>` : retourne le post avec l'id `:id`.
- `/api/post/[id]/like.ts`; url : `/api/post/:id/like`
  - `<author:int>POST<Like>` : permet de liker le post avec l'id `:id`. Retoune les deux id du post et de l'utilisateur.
- `/api/post/[id]/unlike.ts`; url : `/api/post/:id/unlike`
  - `<author:int>POST<Like>` : permet de ne plus liker le post avec l'id `:id`. Retoune les deux id du post et de l'utilisateur.
- `/api/post/[id]/reply.ts`; url : `/api/post/:id/reply` (nous devrions regrouper ça dans le fichier `index.ts`)
  - `<author:int, content:string>POST<Post>` : permet de répondre au post avec l'id `:id`. Retourne le nouveau post créé.
- `/api/post/[id]/delete.ts`; url : `/api/post/:id/delete` (nous devrions regrouper ça dans le fichier `index.ts`)
  - `DELETE<null>` : permet de supprimer le post avec l'id `:id`. Retourne null