Card Board Games

Titre: Concepteur et développeur d'application

Date : 24/06/2022

Table des matières

Compétences du référentiel couverte par le projet

Le projet couvre les compétences énoncées ci-dessous:

Pour l'activité 1, **"Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité"**:

- Maquetter une application.
- Développer des composants d'accès aux données.

Pour l'activité 2, **"Concevoir et développer la persistance des données en intégrant les recommandations de sécurité"**:

- Concevoir une base de données.
- Intégrer les recommandations de sécurité.
- Développer des composants dans le langage d'une base données.

Pour l'activité 3, **"Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité"**:

- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement.
- Concevoir une application.
- Développer des composants métier .
- Construire une application organisée en couches.
- Développer une application mobile.
- Préparer et exécuter les plans de tests d'une application.
- Préparer et exécuter le déploiement d'une application.

Résumé (anglais)

Card board games is a school project that we made as a team of 4 students. We wanted to make an application able to host card and board games. We created a database and builded the front-end and the back-end in this way.

To implement these games we built a multi-layered application with NestJS , React-Native and a socket server. Our approach allows us to build real-time services and gaming. The majority of the data that make the players able to play are stored in the component of the react-native part and the database hydrates the component with the values that are selected by the player and shared with the others.

The cards game is the only type of game implemented but it's just a question of time because the system can easily handle the implementation of the boards game type. Each cards has been designed by us. We also draw two playground to positionate players and cards.

At the end, player can create and join a lobby. Play a game with other people and then see the score of his game.

Everyting has been developped in javascript (NodeJs, TypeScript)

Périmètre du projet

L'application mobile sera faite en français et disponible sur toutes les plateformes.

Expression des besoins

Nous voulons construire une application mobile divertissante et dans l'air du temps. De plus, nous aimerions une application qui pourrait viser tout type de personne. Nous nous sommes naturellement tournés vers les jeux. Plus spécifiquement une application de jeu de carte et de plateau.

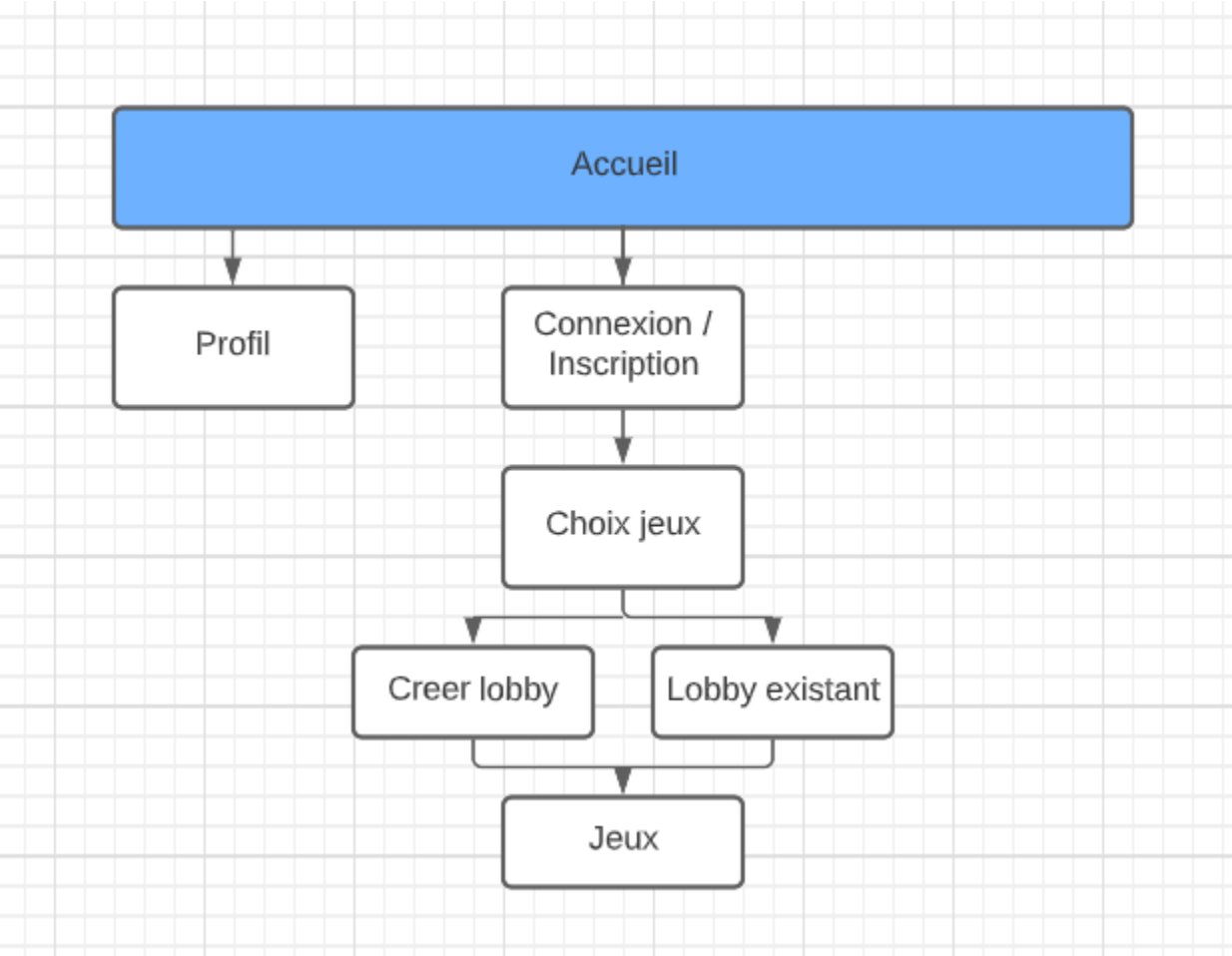
Spécifications fonctionnelles

Cibles visées par le site

L'application card board games vise tous types de personnes du plus jeune au plus vieux qui souhaite jouer à des jeux pour se détendre ou même défier les autres.

Arborescence du site

L'arborescence du site est la suivante:



Collaboration dans le projet

Lors du développement de ce projet nous étions quatre. Il a fallu organiser les tâches en fonction de notre planning d'alternance et en dehors des cours que l'on recevait. Nous avons tout d'abord brainstormé sur les différents sujets qui nous intéressaient avant de choisir le thème du jeu de carte en ligne et en temps réel.

Ce projet comporte beaucoup de technologies que nous ne maîtrisons pas. Pour éviter de se laisser déborder, nous avons organisé des équipes tournantes.

Deux personnes devaient mettre en place la partie BACK-END de l'application tandis que 2 autres devaient mettre en place le FRONT-END. Le but était de maîtriser les technologies front et back avant de faire un échange de connaissance entre les deux équipes. Ainsi en progressant chacun de notre côté, il ne nous resterait plus qu'à appliquer les conseils des uns et des autres pour compléter toutes les tâches que l'on s'était fixées. Pour connaître les tâches à long terme que nous aurions à réaliser nous avons produit un diagramme de Gantt. Celui-ci nous permet de visualiser toutes les tâches à effectuer sur une sorte de planning annuel. Ainsi à chaque fois que l'on rentrait d'alternance nous savions sur quoi nous concentrer et quelle serait la prochaine étape à remplir avant de pouvoir s'intéresser à une autre technologie ou fonctionnalité.

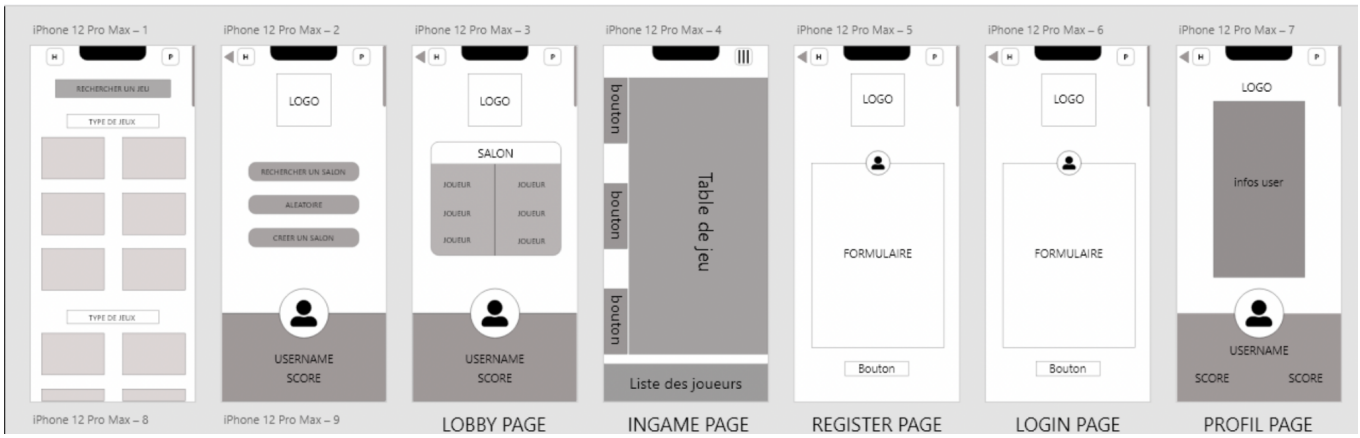
Dans une échelle de temps à court terme nous avons également utilisé Trello. Le but était que les 2 sous équipes BACK-END/FRONT-END puisse s'échanger leurs besoins à court terme. A chaque fois qu'une personne rencontrait un problème ou qu'elle remarquait qu'il fallait ajouter une fonctionnalité ou l'arranger à un endroit, une carte a été créée. Certaines cartes servent également aux ressources de documentation et nous permettent d'apprendre et d'avancer dans la même voie même si l'on ne se voyait pas régulièrement à cause de notre rythme d'alternance.

Front-end

Maquettage

Après avoir réfléchi sur le concept de notre application, il a fallu réfléchir aux pages nécessaires au bon fonctionnement de notre future application.

Nous avons alors réalisé une wireframe / zoning de l'application.



Conception et développement de l'application

A l'aide de la maquette, nous avons développé les pages de l'application en suivant le style graphique défini.

Nous avons créé des composants pour chaque pages tout en respectant les bonnes pratiques de développement Objet;

Ces composants sont des boutons, des entrées utilisateurs, tout ce qui permet à l'utilisateur d'interagir avec l'application, il vont être utilisés de nombreuses fois afin de développer notre interface utilisateur.

Voici la page d'accueil de l'application:

La page Connexion:

La page Inscription:

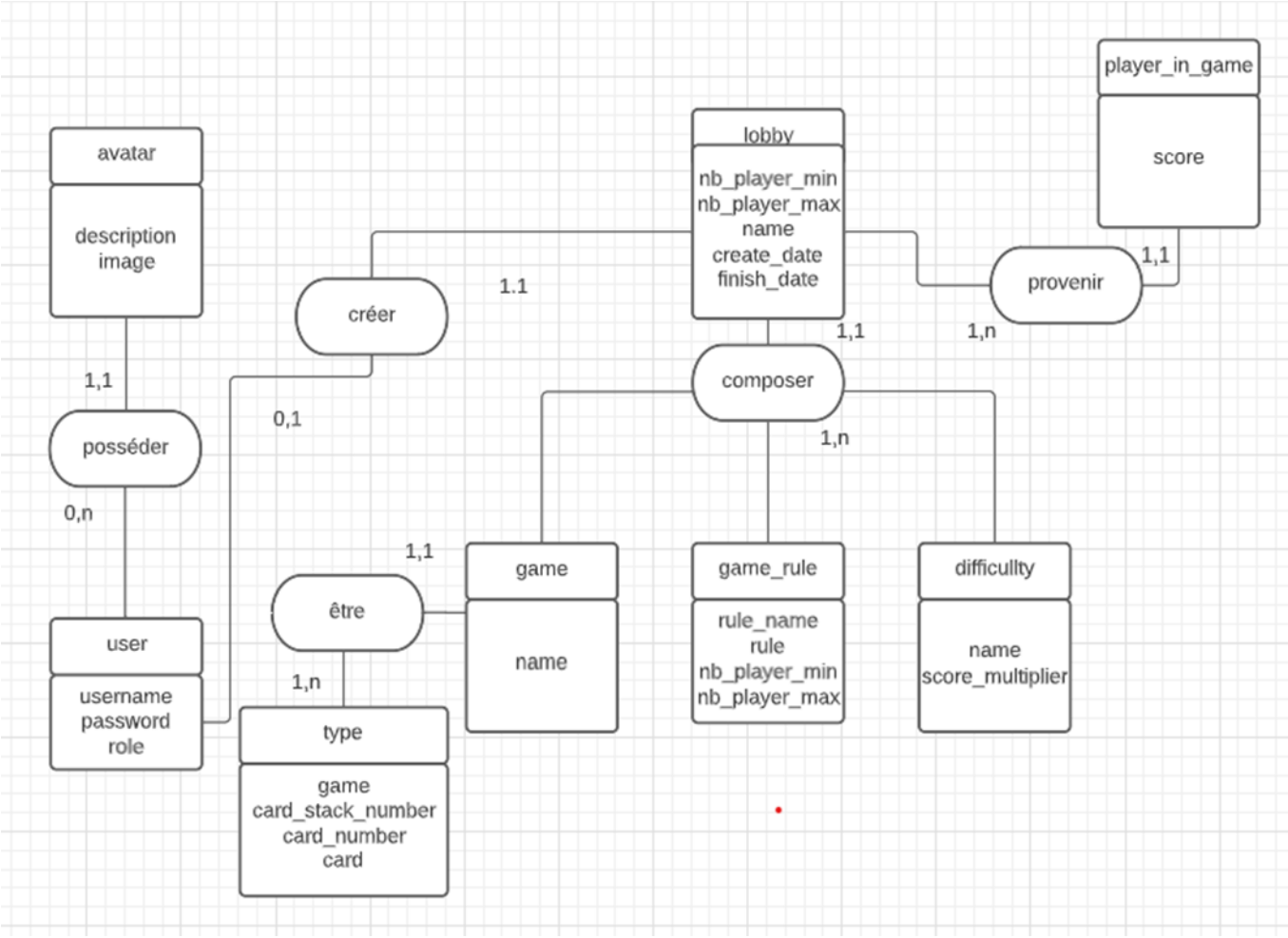
Nous avons utilisé le framework React Native et le langage Javascript pour développer l'application.
En utilisant la structure de la maquette.

Back-end

Base de données

Dans le cadre de notre projet d'application de jeux de carte mobile, nous avons imaginé un MCD et un MLD qui ont évolué au fil du développement de notre projet. Cette base de données possède une dizaine de tables, la plupart d'entre elles sont contraintes par les cascades via leurs clés étrangères et sont toutes accessibles et testables via notre environnement Swagger.

Avant de pouvoir développer la base de données nous nous sommes concertés avec un papier et un stylo pour imaginer à quoi elle devrait ressembler. En avançant dans le développement nous avons fait évoluer notre base de données pour répondre au nouveau besoin que nous avons rencontré et auxquels il fallait répondre.



Si create_date != null alors ce n'est plus un lobby mais une partie.

Si finish_date != nulll alors la partie est terminée.

Si * == null et que le serveur socket ne détecte aucun socket dans la room du lobby alors le lobby se supprimera automatiquement via le serveur socket pour annuler le lobby créer.

Nous avons utilisé PhpMyAdmin, sql, InnoDB, MariaDB, Swagger, LucidChart.

Composants métiers

Un utilisateur se connecte, émet naturellement un événement de connexion sur le serveur socket qui l'authentifie et l'utilisateur reçoit un token qui lui permet d'accéder à l'ensemble des espaces de l'application.

La création et l'accessibilité des lobbies :

Il a donc accès au bouton de création de lobby. Celui-ci appelle un composant général CreateLobby dans lequel un composant CreateLobbyServices permet de fetch l'ensemble des jeux disponibles. Une fois le jeu sélectionné, le composant GameRule permet d'afficher les règles et les difficultés de jeu associé au jeu sélectionné.

On nomme alors le lobby et on appuie sur le bouton créer. Une redirection s'effectue, les données sont envoyées au serveur en POST via le component CreateLobbyServices et l'on est dirigé directement dans le composant général Lobby que l'on vient de créer.

A ce moment-là un emit (envoie d'un signal socket comportant parfois une data) est effectué vers le serveur socket pour le notifier de notre présence. Le socket de l'utilisateur est alors directement associé à une room qui portera le nom du Lobby. Un fois ces signaux effectués, le Lobby figurera dans la liste des Lobby disponibles.

Un autre utilisateur se connecte et souhaite accéder au Lobby que l'on vient de créer. Il appuie sur le bouton liste des Lobbies et accède au composant général LobbyList. Ce composant affiche l'ensemble des lobbies existant qui sont tous répertoriés par leur nom de lobby qui est aussi le nom de la room associé en temps réel. L'utilisateur clique, rejoint ainsi le lobby et répète les signaux qu'a effectués le créateur du lobby pour se joindre à la room en y insérant son socket d'utilisateur.

Un troisième joueur souhaite se connecter. Mais le lobby est plein. Il recevra alors une réponse du serveur socket lors de sa tentative de connexion qui lui expliquera que le serveur est déjà rempli.

Si un utilisateur quitte alors il pourra rejoindre et si tous les utilisateurs quitte ou si la partie est lancée et terminée alors le lobby se supprimera par lui-même en socket avec l'événement disconnect. Lors de la déconnection en socket du lobby, si nodejs constate que le lobby est vide ou n'existe plus alors il va envoyer une requête de suppression du lobby en base de donnée via notre API utilisé avec axios dans le serveur socket.

Ainsi, la liste des lobby ne restera jamais rempli de serveur indisponible et sera toujours mis à jour avec soit les lobby disponible soit les lobby plein ou déjà en cours de jeu.

.Application organisée en couches

Notre application de jeu de carte mobile présente plusieurs couches afin de fonctionner. Tout d'abord un serveur (distribution Debian 11) nous permet d'héberger notre back-end. Celui-ci a été développé sur le FrameWork NestJS, en NodeJS et en TypeScript.

Un système de routeur intégré au framework nous a permis de construire notre API grâce à l'appel de fonction suivant les routes demandées. Ces routes exécutent des fonctions ayant des requêtes sql permettant de satisfaire les besoins de l'utilisateur de l'API vis-à-vis de la base de données et ce qu'elle contient.

Pour chaque entité de notre back-end, NestJS va posséder un controller un model d'entité, une interface, des DTO , des services, un fichier .spec dans lequel on peut tester nos fonctions et un module qui va permettre d'assembler la logique entre chaque fichier, puis de tout rassembler dans le main.module qui est le fichier module racine du projet par lequel toutes les entités qui sont imbriquées au lancement de NestJS vont être appelées.

```
@Module({

 imports: [TypeOrmModule.forFeature([User])],

 exports: [TypeOrmModule],

 providers: [UsersService],

 controllers: [UsersController],

}

)
```

Notre base de données est accessible via notre système de gestion de base de données MariaDB. Il utilise InnoDB qui est un moteur de stockage pour nous fournir des relations entre les tables. Notre base de données fonctionne à l'aide du serveur web Nginx et est relié à notre API via Un mapping objet-relationnel (en anglais object-relational mapping ou ORM, dans notre cas TypeOrm) qui est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet. Ce programme définit des correspondances entre les schémas de la base de données et les classes du programme applicatif.

Ainsi lorsqu'une requête s'exécute elle doit être validé par TypeOrm et la configuration qu'on lui a attribuée.

Notre API est disponible via un URL et le port 3001.

Celle-ci va être appelée par notre Front et une autre partie de notre back-end qui est le serveur socket.

Développé a l'aide de NodeJS et socket.io notre serveur socket est lui aussi hébergé sur le même serveur, sur le port 3002 et écoute en permanence les événements qu'il reçoit en provenance du front. Dans ses réponses, il appel parfois l'API pour donner des informations à l'utilisateur.

Le serveur socket nous permet d'avoir une gestion des événements javascript en temps réel entre tous les utilisateurs.

Ainsi nos composant métier réagissent en temps réel aux cliques de chacun (lobby, jeu de carte), il nous permet également de mettre en place tout ce qui va permettre des interactions sociales entre les utilisateurs comme l'ajout d'amis, l'envoie et la réception de message, les notifications et encore d'autre fonctionnalités à venir…

Notre front-end est développé en React-native et construit à l'aide d'Expo.

React et react-native sont deux langages très proches, pour ne pas dire que ce sont les mêmes.

Cette proximité dans la compatibilité des deux langages nous permet d'émuler notre application mobile sur navigateur (notamment pour tester rapidement l'avancée de notre application) et sur mobile à l'aide d'un QR code à scanner. Afin de pouvoir accéder à l'API le front-end utilise la librairie AXIOS et utilise le SecureStore pour pouvoir utiliser le localStorage du téléphone afin de stocker des token ou des cookies.

Au début nous nous étions lancés dans un design pattern atomique. Notre code est donc divisé en organisme (ensemble d'une page), molécule (un composant appelé dans un organisme) et d'un atome (un tout petit composant appelé dans une molécule). Ainsi avec un ensemble de molécules et d'atomes nous sommes capable de générer une page modulaire (un organisme). Mais à terme nous n'avons pas utilisé cette architecture car elle nous demandait de trop refactoriser le code.

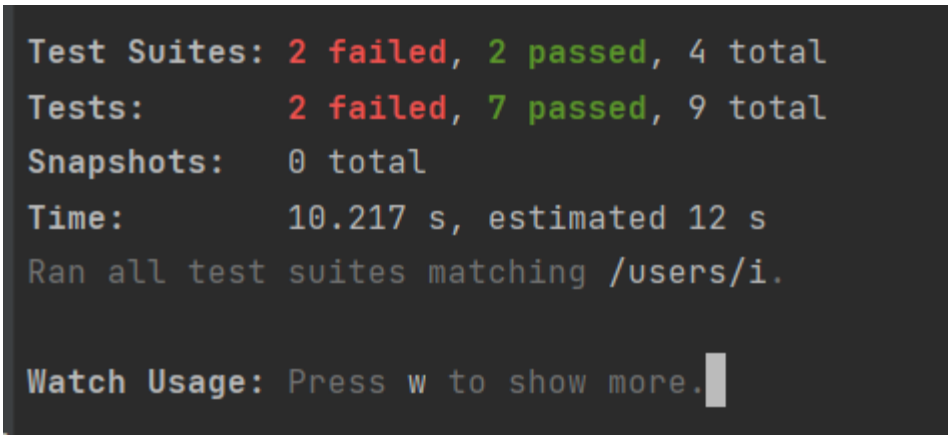
Nous sommes donc restés sur une imbrication assez classique de nos composants dans une navigation Stack (react-navigation) dans laquelle on appelle un composant vue qui sera constitué de plusieurs composants qui effectueront des actions plus ou moins indépendantes du composant parent.

Nous faisons passer nos states dans notre stackNavigation qui alimente l'ensemble de nos pages. Parmi les states les plus partagés on a notamment le Token et le Socket de l'utilisateur. Ainsi notre utilisateur est identifié à la fois sur l'API et sur le serveur socket une fois qu'il a réussi sa connexion à l'aide de son compte utilisateur.

Plans tests de l’application

Afin de prévoir au mieux le déploiement de toute l’interface de programmation d'application ou application programming interface (API) que nous avons choisi de développer en utilisant Javascript et plus précisément le Framework NestJS, nous avons effectué une batterie de test unitaires ainsi qu’un test dit « end-to-end » sur l’intégralité de notre API.

Les test end-to-end sont des tests globaux réalisés sur l'intégralité d’un bout à l'autre de l’application et non plus sur chacune des fonctions de chacun des composants. Concrètement, lors d’un test dit end to end on recrée l’environnement de développement et d’utilisation de notre app et on test l’ensemble des fonctionnalité avec plusieurs types de données et plusieurs cas de figure afin de pouvoir s’asssurer que notr application est bien sécurisé et marche comme on attend qu’elle marche



Grâce à NestJS et la création automatique des fichiers de test utilisant le Framework de testing de javascript Jest, la création de test est facilitée. En effet, avec l'utilisation de Jest, la création de fausses données est facilitée pour vérifier que la fonction fonctionne correctement et renvoie exactement ce que nous attendions qu'elle renvoie. Il a fallu aussi effectuer des tests sur l'ensemble des fonctions de base c'est-à-dire l'ensemble des opérations possibles sur chacun de nos composants aussi bien sur la partie Controller que Service des modules de notre application.

Déploiement

Dans le cadre du déploiement de notre API et de notre serveur socket nous avons utilisé un serveur distant. Dans un premier temps nous avons créé une Virtual Machine (VM) pour accéder au terminal SSH de notre serveur. Étant sur Window, nous ne possédons pas de terminal SSH, nous aurions pu en installer un léger mais nous avons préféré tester cela sur une VM. Une fois l'environnement mis en place nous avons pu accéder à notre serveur via les identifiants utilisateurs qui nous ont été fournis.

Avant de pouvoir procéder au déploiement de notre API il a fallu installer diverses technologies. On a tout d'abord installé NodeJS pour pouvoir utiliser NPM (gestion des paquets), NestJS (api sous nodejs). Puis apache2 même si par la suite on est passé sur Nginx. Et MariaDB pour gérer nos bases de données en SQL.

A l'aide d'un gestionnaire de port UFW (debian) nous avons ouvert les ports 3001 (API) et 3002 (socket). Puis est venu le temps de la migration sftp (Secure file transfert program) que l'on a effectué à l'aide de FileZilla.

Il nous a suffi de transférer nos fichiers sur un répertoire de linux configuré pour recevoir les transfert sftp, puis de déplacer les dossiers reçus dans le répertoire de notre utilisateur. Enfin nous avons pu lancer les npm install pour recevoir tous les modules nécessaires au lancement de nos deux serveurs et les tester.

Nous avons ainsi accès à notre serveur API via l'url http://51.75.241.128:3001 et a notre serveur Socket via l'url http://51.75.241.128:3002

Spécifications techniques

Choix Techniques et environnement de travail

Technologies utilisées pour la partie back-end:

- Le site sera réalisé avec les langage javascript (nodeJS , TypeScript)
- Une base de données SQL

Technologies utilisées pour la partie front-end:

- Le projet sera réalisé avec du ReactNative
- NativeBase
- ReactNavigation

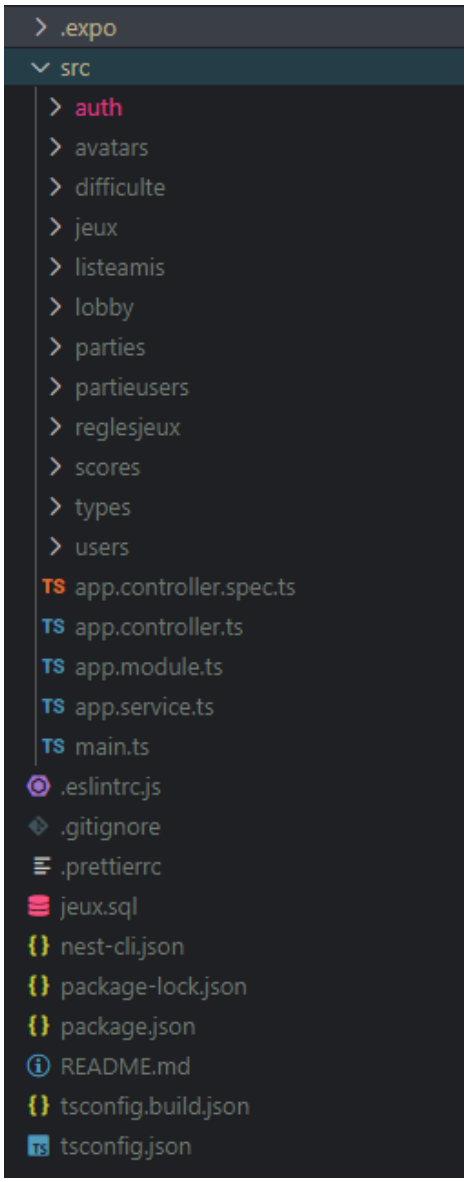
L'environnement de développement est le suivant:

- Editeur de code: Visual studio code
- Outil de versionning: GIT, Github
- Maquettage: adobe xd

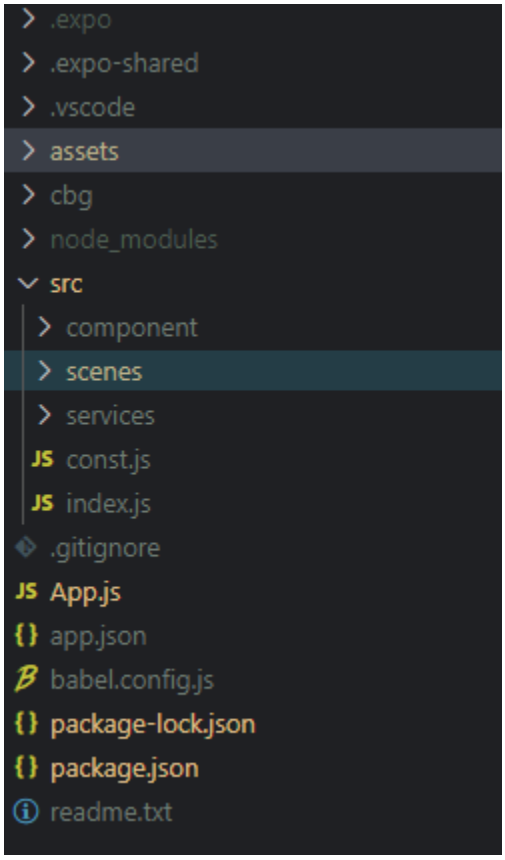
_____Architecture

Pour ce qui est de l'architecture du projet nous avons choisi une architecture classique. Avec des dossiers organisés. Les dossier s'organisent comme ci dessous:

Pour le back :


.expo
src
auth
avatars
difficulte
jeux
listeamis
lobby
parties
partieusers
reglesjeux
scores
types
users
app.controller.spec.ts
app.controller.ts
app.module.ts
app.service.ts
main.ts
.eslintrc.js
.gitignore
.prettierrc
jeux.sql
nest-cli.json
package-lock.json
package.json
README.md
tsconfig.build.json
tsconfig.json

Pour le front :



Sécurité mise en oeuvre

Ensuite le problème de la sécurité de notre application s'est imposé. Pour répondre à cela, nous avons mis en place un certain nombre de moyens afin d'assurer une application sécurisée.

Les injections SQL et les failles XSS sont les premiers points de sécurité que nous avons abordés.L'injection SQL est généralement utilisée dans les formulaires présents sur votre site. Le pirate inclura une chaîne de caractère lui permettant de détourner votre requête SQL et ainsi de récupérer les informations de vos utilisateurs et bien d'autres choses que permet d'effectuer le SQL. La faille XSS consiste à injecter un script arbitraire dans une page pour provoquer une action bien définie. Les autres utilisateurs exécutent le script sans s'en rendre compte dès l'ouverture de la page.Cross veut également dire traverser, car l'un des buts de la faille est d'exécuter un script permettant de transmettre des données depuis un site vers un autre. Pour nous protéger de ces deux failles de sécurité nous avons utilisé DTO (data transfert object).Son but est de simplifier les transferts de données entre les sous-systèmes d'une application logicielle. DTO nous a permis de simplifier et de sécuriser le transfert de données.

```
create(jeux: CreateJeuxDto): Promise<JeuxInterface> {
  return this.jeuxRepository.save(jeux);
}

findAll(): Promise<Jeux[]> {
  return this.jeuxRepository.find();
}

update(id: number, jeux: UpdateJeuxDto): Promise<any> {
  return this.jeuxRepository.update(id, jeux);
}

remove(id: number): Promise<any> {
  return this.jeuxRepository.delete(id);
}
```

Dans ce screenshot on peut apercevoir la déclaration de plusieurs fonctions. Celles-ci sont contraintes par typeScript et par l'usage des DTO. Si TypeScript ne rencontre pas l'objet demandé, il enverra une erreur. Si l'objet reçu ne correspond pas aux données l'interface alors la fonction ne pourra pas être exécutée.
Si les données ne correspondent pas aux contraintes contenues dans les DTO alors la fonction ne s'exécutera toujours pas.

Par exemple :
Ici, pour poster un nouveau jeu, il faut que les champs soit rempli et du bon type. Il faudra aussi que le nom des champs correspondent à ceux dans l'interface. Une fois toutes ces contraintes respectées, le nouveau jeu pourra être posté.

api_back > src > jeux > dto > TS create-jeux.dto.ts > ...

```ts
import { IsNotEmpty, IsNumber, IsString } from 'class-validator';
import { ApiProperty, PartialType } from '@nestjs/swagger';
import { Type } from 'class-transformer';
import { GetJeuxDto } from './get-jeux.dto';

export class CreateJeuxDto extends PartialType(GetJeuxDto) {

  @IsNotEmpty()
  @ApiProperty()
  @Type(() => String)
  nom: string;

  @IsNotEmpty()
  @ApiProperty()
  @Type(() => Number)
  idtype: number;
}
```

(a noter que ApiProperty sert à configurer Swagger (testeur d'api))

Ensuite nous nous sommes concentrés sur la faille CSRF (Cross-site request forgery). Il s'agit d'effectuer une action visant un site ou une page précise en utilisant l'utilisateur comme déclencheur, sans qu'il en ait conscience.On va deviner un lien qu'un utilisateur obtient habituellement, et tout simplement faire en sorte qu'il clique lui-même sur ce lien. Pour pallier ce problème un système de token a été mis en place. Grâce au JWT (Json Web Token). Il permet l'échange sécurisé de jetons (token) entre plusieurs parties. Cette sécurité se traduit par la vérification de l'intégrité et de l'authenticité des données. Un JWT se structure de la façons suivante :

- Un en-tête (header) : utilisé pour décrire le jeton (objet json).
- Une charge utile (payload) : représente les informations embarquées dans le jeton (objet json).
- Une signature numérique : générée à partir du payload et d'un algorithme.

```
const payload = {

  sub: user.id,
  username: user.username,
  idavatar: user.idavatar,
  role: user.role,
  expiresIn: ''

};
return {
  access_token: this.jwtService.sign(payload),
};
```

Ici nous définissons le payload il est propre à ce que désire le développeur. Dans notre projet nous avons décidé de définir la payload avec l'id utilisateur, l'username, son avatar et son rôle. Toutes ces informations sont prises en base de données.Puis nous définissons la durée de validité du token (expiresIn). Puis grâce à jwtservice nous formons le token final qui comportera les 3 grandes parties évoquées au-dessus (header, payload et signature). La fonction sign nous permet de créer la signature a partir ici du payload.

Pour finir nous nous sommes concentrés sur la faille upload puisque l'utilisateur aura la possibilité d'uploader son image de profil. Le principe de l'attaque est très simple. Le pirate essaie d'uploader un fichier qui contient du code malveillant ou un code PHP de sa création. Si la faille est là alors le fichier finira pas atterrir sur le serveur. Il suffit ensuite au pirate d'appeler son fichier pour que celui-ci s'exécute. Pour éviter cela, nous avons mis un filtre. C'est-à- dire que l'utilisateur ne pourra uploader d'autres formats que les suivants : png, jpg, jpeg. De plus, nous avons limité la taille des fichiers à uploader.

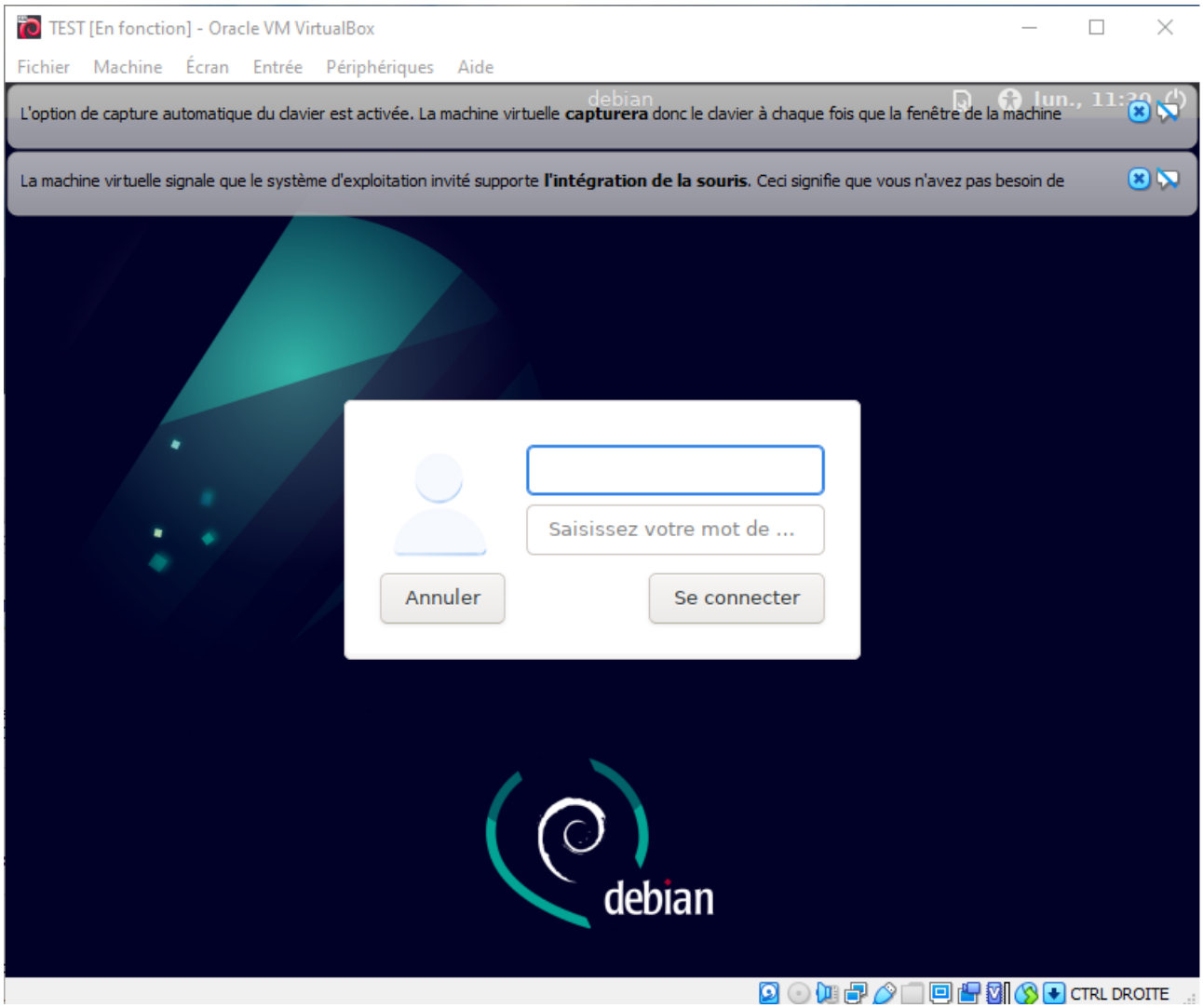
Mes sources d’aides

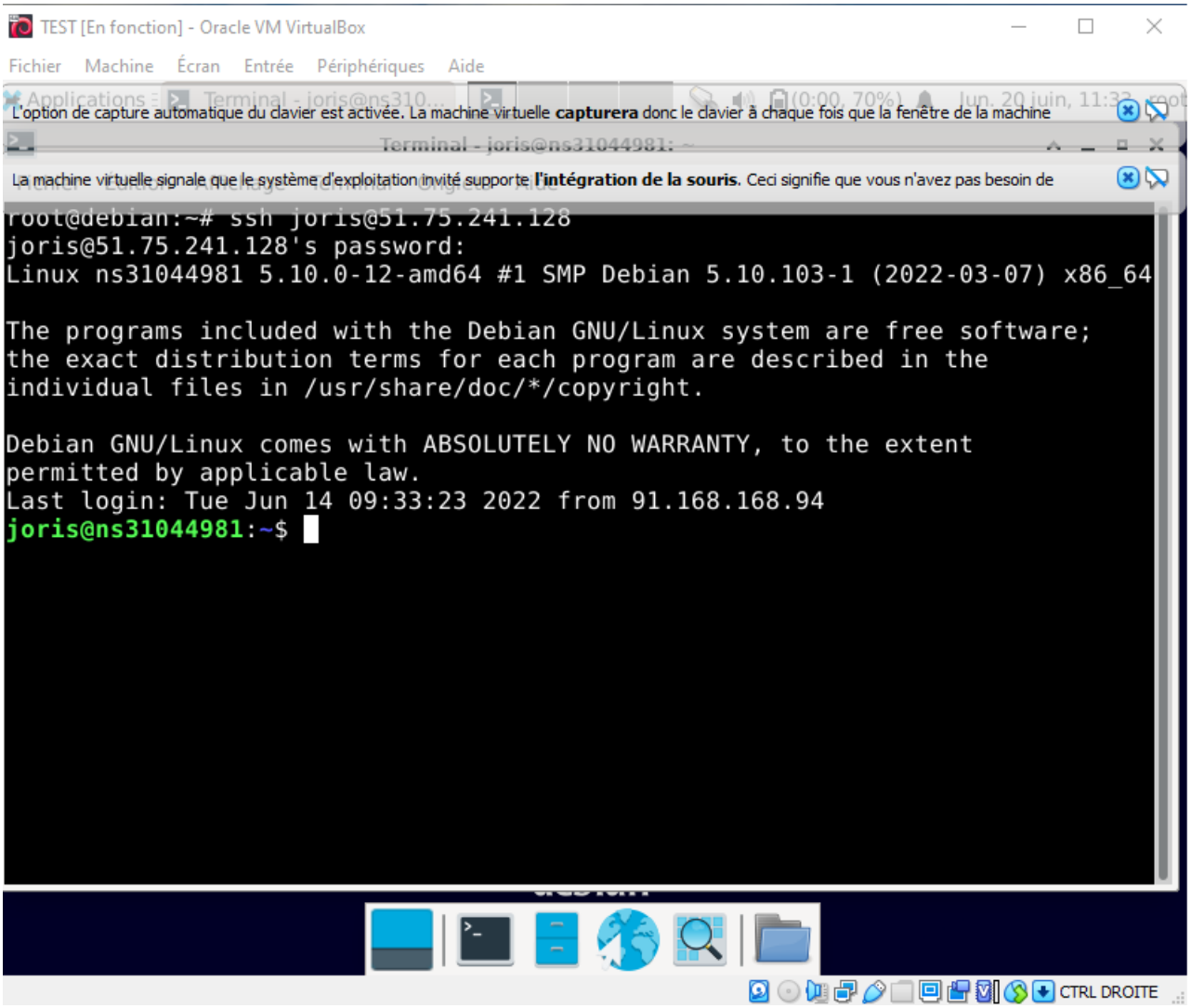
Pour ce projet je me suis aidé des sources suivantes:

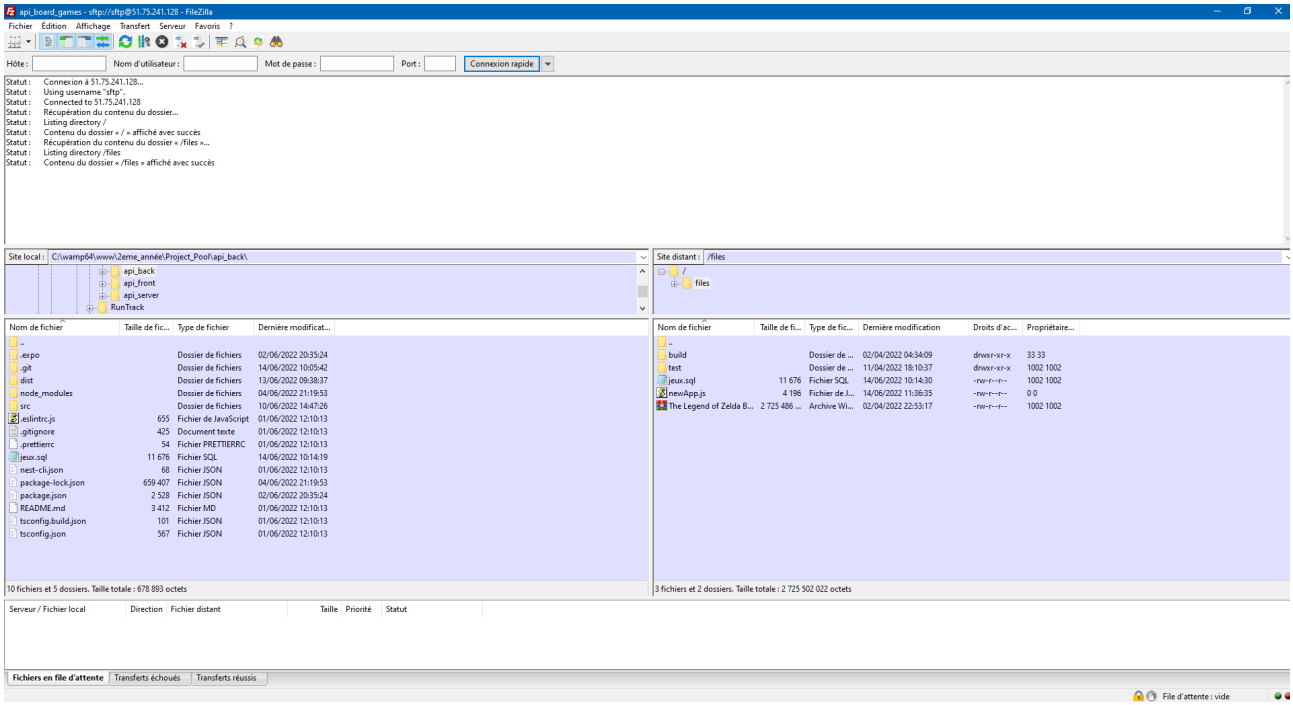
- https://reactnative.dev/
- https://stackoverflow.com/
- https://developer.mozilla.org/fr/docs/Web/JavaScript
- D’autres forum et sites

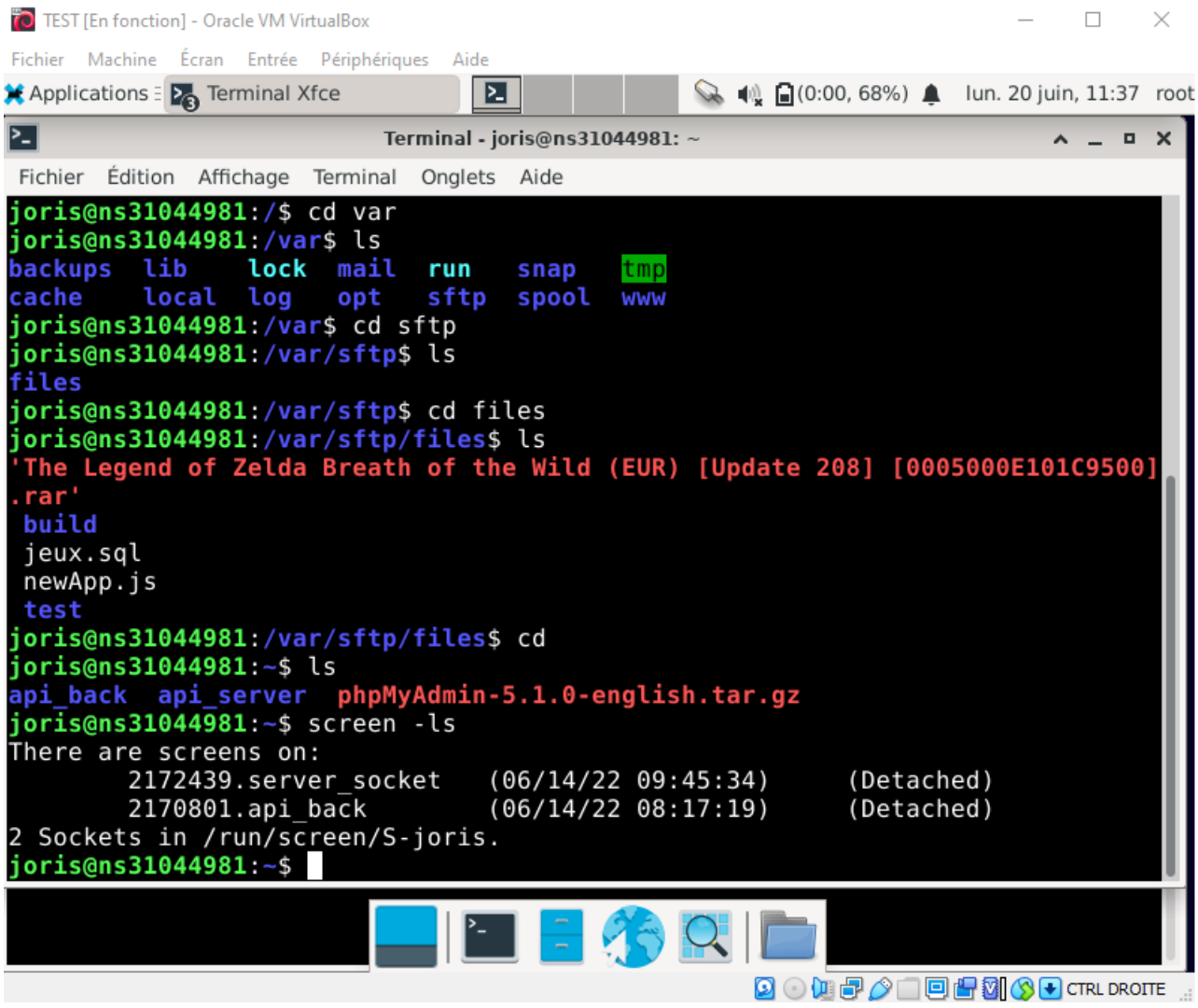
Annexe

Déploiement :

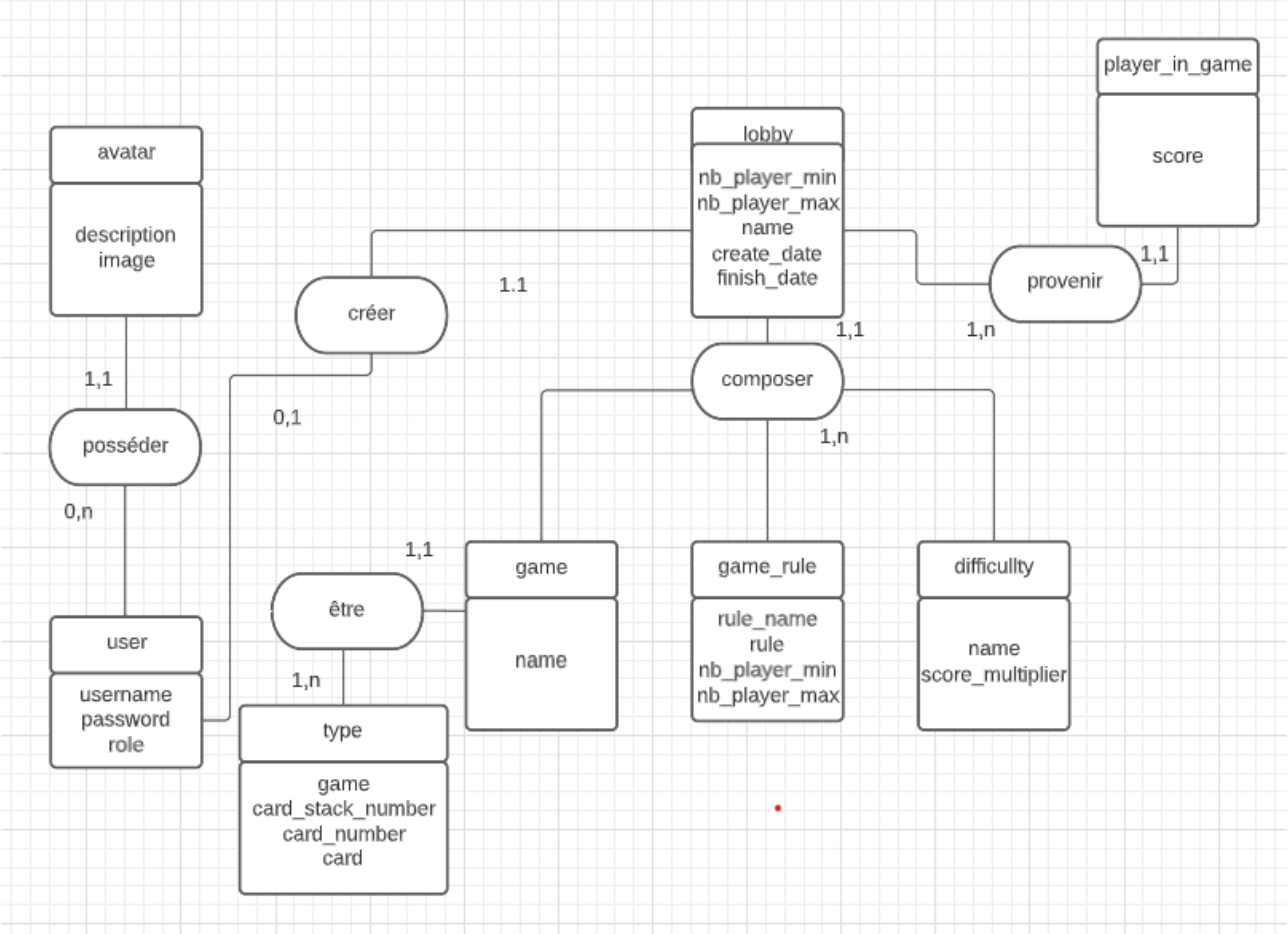



TEST [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
L'option de capture automatique du clavier est activée. La machine virtuelle capturera donc le clavier à chaque fois que la fenêtre de la machine
La machine virtuelle signale que le système d'exploitation invité supporte l'intégration de la souris. Ceci signifie que vous n'avez pas besoin de
root@debian:~# ssh joris@51.75.241.128
joris@51.75.241.128's password:
Linux ns31044981 5.10.0-12-amd64 #1 SMP Debian 5.10.103-1 (2022-03-07) x86_64
The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jun 14 09:33:23 2022 from 91.168.168.94
joris@ns31044981:~$
CTRL DROITE

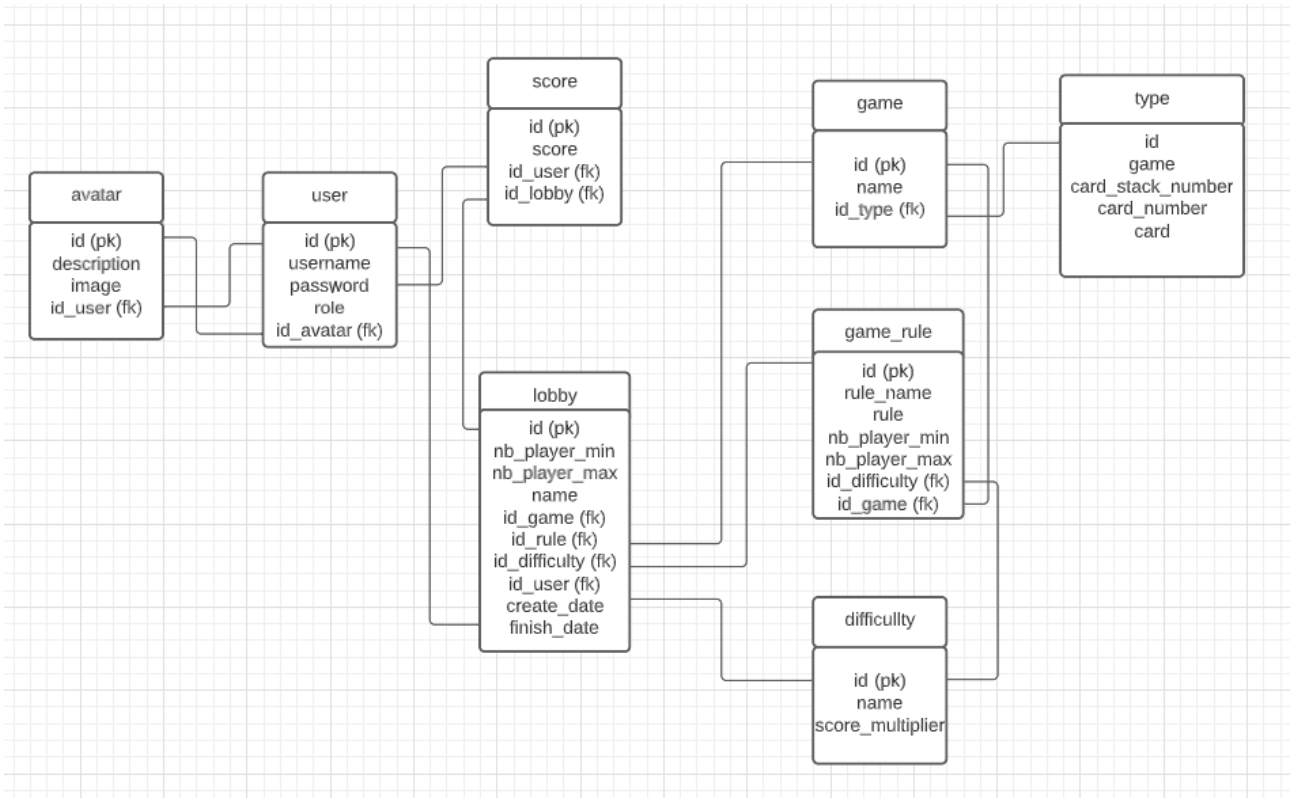

api_board_games - sftp://sftp@51.75.241.128 - FileZilla
Fichier Édition Affichage Transfert Serveur Favoris ?
Hôte : Nom d'utilisateur : Mot de passe : Port : Connexion rapide
Statut : Connexion à 51.75.241.128...
Statut : Using username "sftp".
Statut : Connected to 51.75.241.128
Statut : Récupération du contenu du dossier...
Statut : Listing directory /
Statut : Contenu du dossier « / » affiché avec succès
Statut : Récupération du contenu du dossier « /files »...
Statut : Listing directory /files
Statut : Contenu du dossier « /files » affiché avec succès
Site local : C:\wamp64\www\2eme_année\Project_Pool\api_back\
api_back
api_front
api_server
RunTrack
Site distant : /files
files
Nom de fichier Taille de fic... Type de fichier Dernière modificat...
.expo Dossier de fichiers 02/06/2022 20:35:24
.git Dossier de fichiers 14/06/2022 10:05:42
dist Dossier de fichiers 13/06/2022 09:38:37
node_modules Dossier de fichiers 04/06/2022 21:19:53
src Dossier de fichiers 10/06/2022 14:47:26
.eslintrc.js 655 Fichier de JavaScript 01/06/2022 12:10:13
.gitignore 425 Document texte 01/06/2022 12:10:13
.prettierrc 54 Fichier PRETTIERRC 01/06/2022 12:10:13
jeux.sql 11 676 Fichier SQL 14/06/2022 10:14:19
nest-cli.json 68 Fichier JSON 01/06/2022 12:10:13
package-lock.json 659 407 Fichier JSON 04/06/2022 21:19:53
package.json 2 528 Fichier JSON 02/06/2022 20:35:24
README.md 3 412 Fichier MD 01/06/2022 12:10:13
tsconfig.build.json 101 Fichier JSON 01/06/2022 12:10:13
tsconfig.json 567 Fichier JSON 01/06/2022 12:10:13
10 fichiers et 5 dossiers. Taille totale : 678 893 octets
Nom de fichier Taille de fi... Type de fic... Dernière modification Droits d'ac... Propriétaire...
build Dossier de ... 02/04/2022 04:34:09 drwxr-xr-x 33 33
test Dossier de ... 11/04/2022 18:10:37 drwxr-xr-x 1002 1002
jeux.sql 11 676 Fichier SQL 14/06/2022 10:14:30 -rw-r--r-- 1002 1002
newApp.js 4 196 Fichier J... 14/06/2022 11:36:35 -rw-r--r-- 0 0
The Legend of Zelda B... 2 725 486 ... Archive Wi... 02/04/2022 22:53:17 -rw-r--r-- 1002 1002
3 fichiers et 2 dossiers. Taille totale : 2 725 502 022 octets
Serveur / Fichier local Direction Fichier distant Taille Priorité Statut
Fichiers en file d'attente Transferts échoués Transferts réussis
File d'attente : vide


TEST [En fonction] - Oracle VM VirtualBox
Fichier Machine Écran Entrée Périphériques Aide
Applications Terminal Xfce
(0:00, 68%) lun. 20 juin, 11:37 root
Terminal - joris@ns31044981: ~
Fichier Édition Affichage Terminal Onglets Aide
joris@ns31044981:/$ cd var
joris@ns31044981:/var$ ls
backups lib lock mail run snap tmp
cache local log opt sftp spool www
joris@ns31044981:/var$ cd sftp
joris@ns31044981:/var/sftp$ ls
files
joris@ns31044981:/var/sftp$ cd files
joris@ns31044981:/var/sftp/files$ ls
'The Legend of Zelda Breath of the Wild (EUR) [Update 208] [0005000E101C9500]
.rar'
build
jeux.sql
newApp.js
test
joris@ns31044981:/var/sftp/files$ cd
joris@ns31044981:~$ ls
api_back api_server phpMyAdmin-5.1.0-english.tar.gz
joris@ns31044981:~$ screen -ls
There are screens on:
2172439.server_socket (06/14/22 09:45:34) (Detached)
2170801.api_back (06/14/22 08:17:19) (Detached)
2 Sockets in /run/screen/S-joris.
joris@ns31044981:~$
CTRL DROITE

Concevoir une base de donnée:

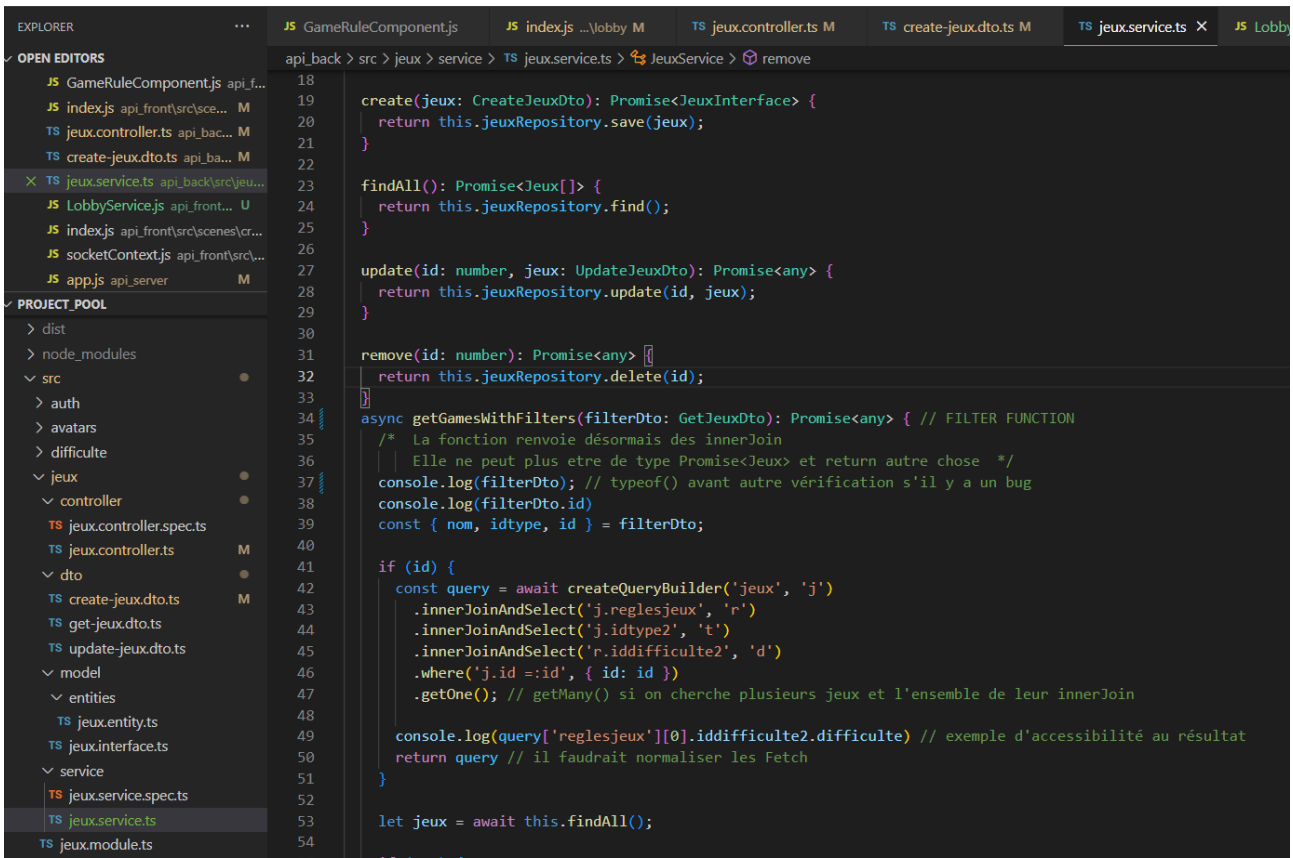


MCD

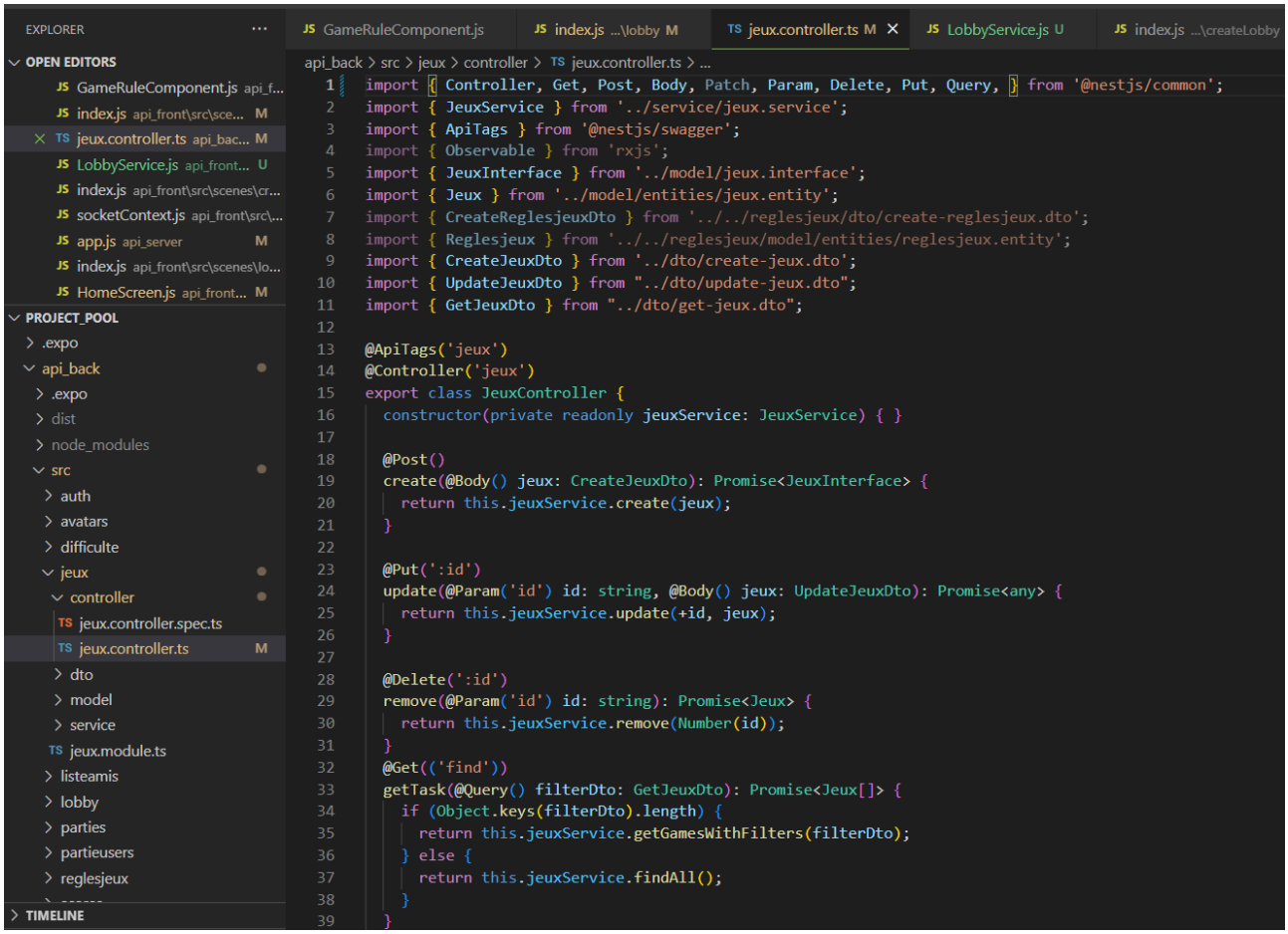


MLD

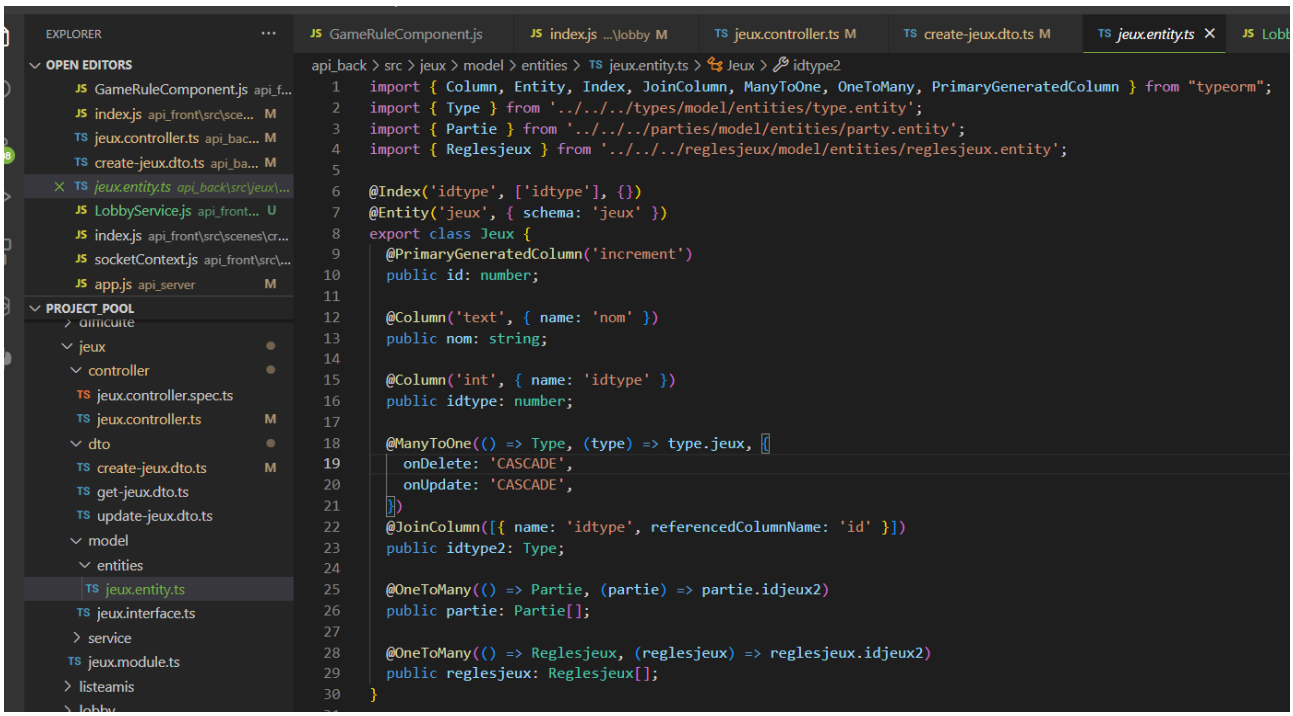
Développer les composant d'accès aux données :



services - controller

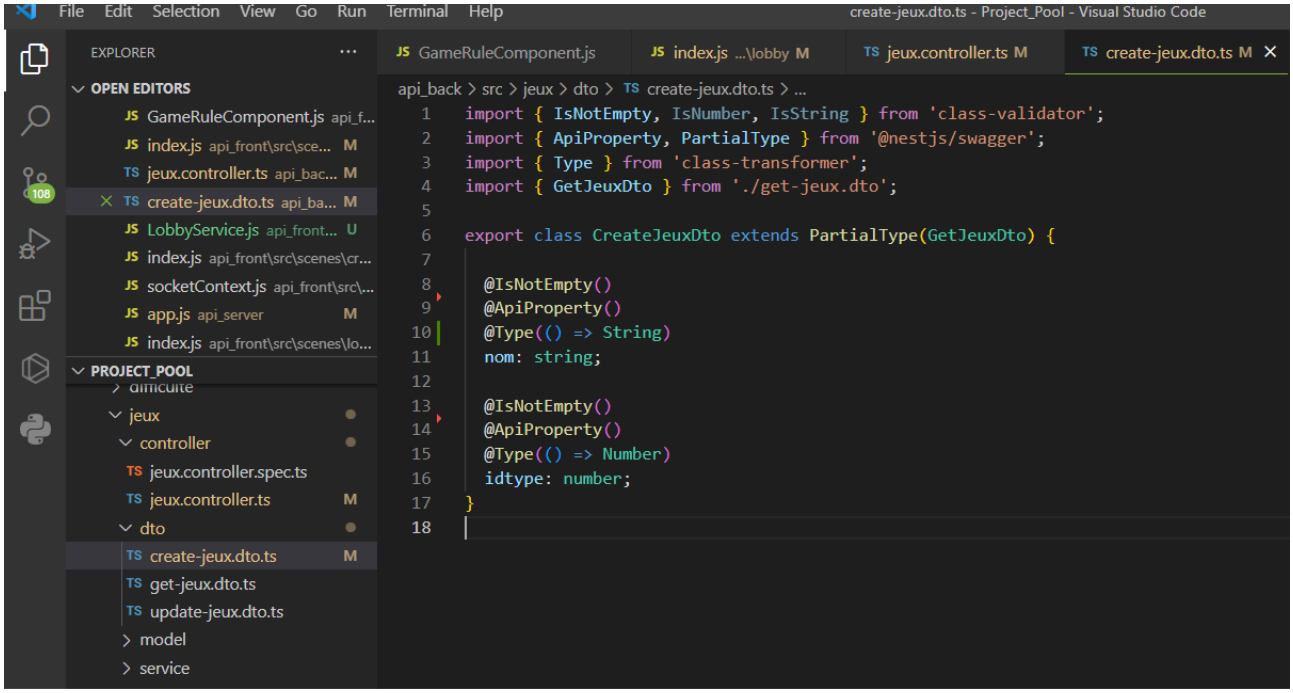


```
import { Controller, Get, Post, Body, Patch, Param, Delete, Put, Query, } from '@nestjs/common';
import { JeuxService } from '../service/jeux.service';
import { ApiTags } from '@nestjs/swagger';
import { Observable } from 'rxjs';
import { JeuxInterface } from '../model/jeux.interface';
import { Jeux } from '../model/entities/jeux.entity';
import { CreateReglesjeuxDto } from '../../reglesjeux/dto/create-reglesjeux.dto';
import { Reglesjeux } from '../../reglesjeux/model/entities/reglesjeux.entity';
import { CreateJeuxDto } from '../dto/create-jeux.dto';
import { UpdateJeuxDto } from "../dto/update-jeux.dto";
import { GetJeuxDto } from "../dto/get-jeux.dto";

@ApiTags('jeux')
@Controller('jeux')
export class JeuxController {
  constructor(private readonly jeuxService: JeuxService) { }

  @Post()
  create(@Body() jeux: CreateJeuxDto): Promise<JeuxInterface> {
    return this.jeuxService.create(jeux);
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() jeux: UpdateJeuxDto): Promise<any> {
    return this.jeuxService.update(+id, jeux);
  }

  @Delete(':id')
  remove(@Param('id') id: string): Promise<Jeux> {
    return this.jeuxService.remove(Number(id));
  }
  @Get(('find'))
  getTask(@Query() filterDto: GetJeuxDto): Promise<Jeux[]> {
    if (Object.keys(filterDto).length) {
      return this.jeuxService.getGamesWithFilters(filterDto);
    } else {
      return this.jeuxService.findAll();
    }
  }
```

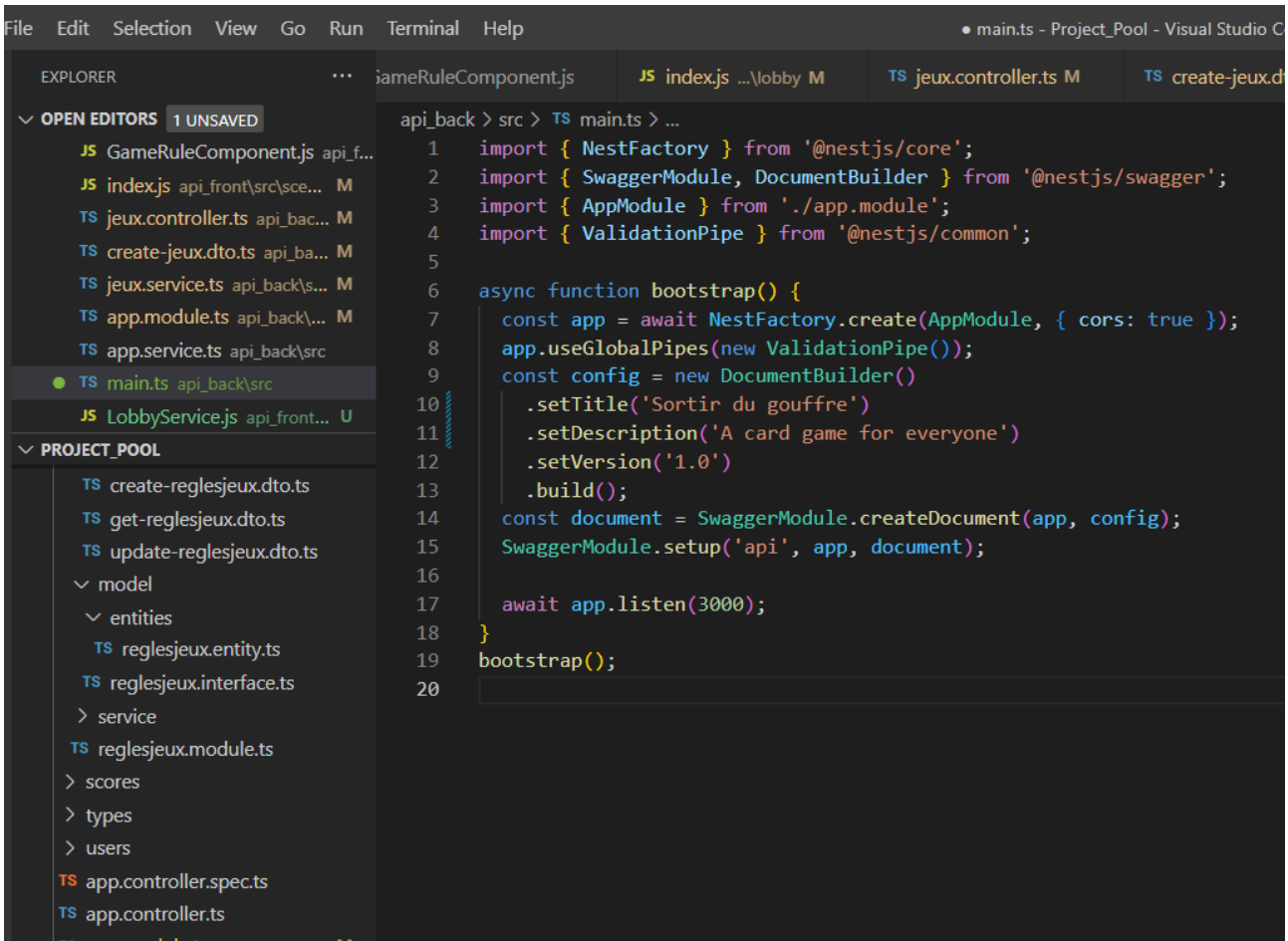


```
import { Column, Entity, Index, JoinColumn, ManyToOne, OneToMany, PrimaryGeneratedColumn } from "typeorm";
import { Type } from '../../../types/model/entities/type.entity';
import { Partie } from '../../../parties/model/entities/party.entity';
import { Reglesjeux } from '../../../reglesjeux/model/entities/reglesjeux.entity';

@Index('idtype', ['idtype'], {})
@Entity('jeux', { schema: 'jeux' })
export class Jeux {
  @PrimaryGeneratedColumn('increment')
  public id: number;

  @Column('text', { name: 'nom' })
  public nom: string;

  @Column('int', { name: 'idtype' })
  public idtype: number;

  @ManyToOne(() => Type, (type) => type.jeux, {
    onDelete: 'CASCADE',
    onUpdate: 'CASCADE',
  })
  @JoinColumn([{ name: 'idtype', referencedColumnName: 'id' }])
  public idtype2: Type;

  @OneToMany(() => Partie, (partie) => partie.idjeux2)
  public partie: Partie[];

  @OneToMany(() => Reglesjeux, (reglesjeux) => reglesjeux.idjeux2)
  public reglesjeux: Reglesjeux[];
}
```

entity ORM

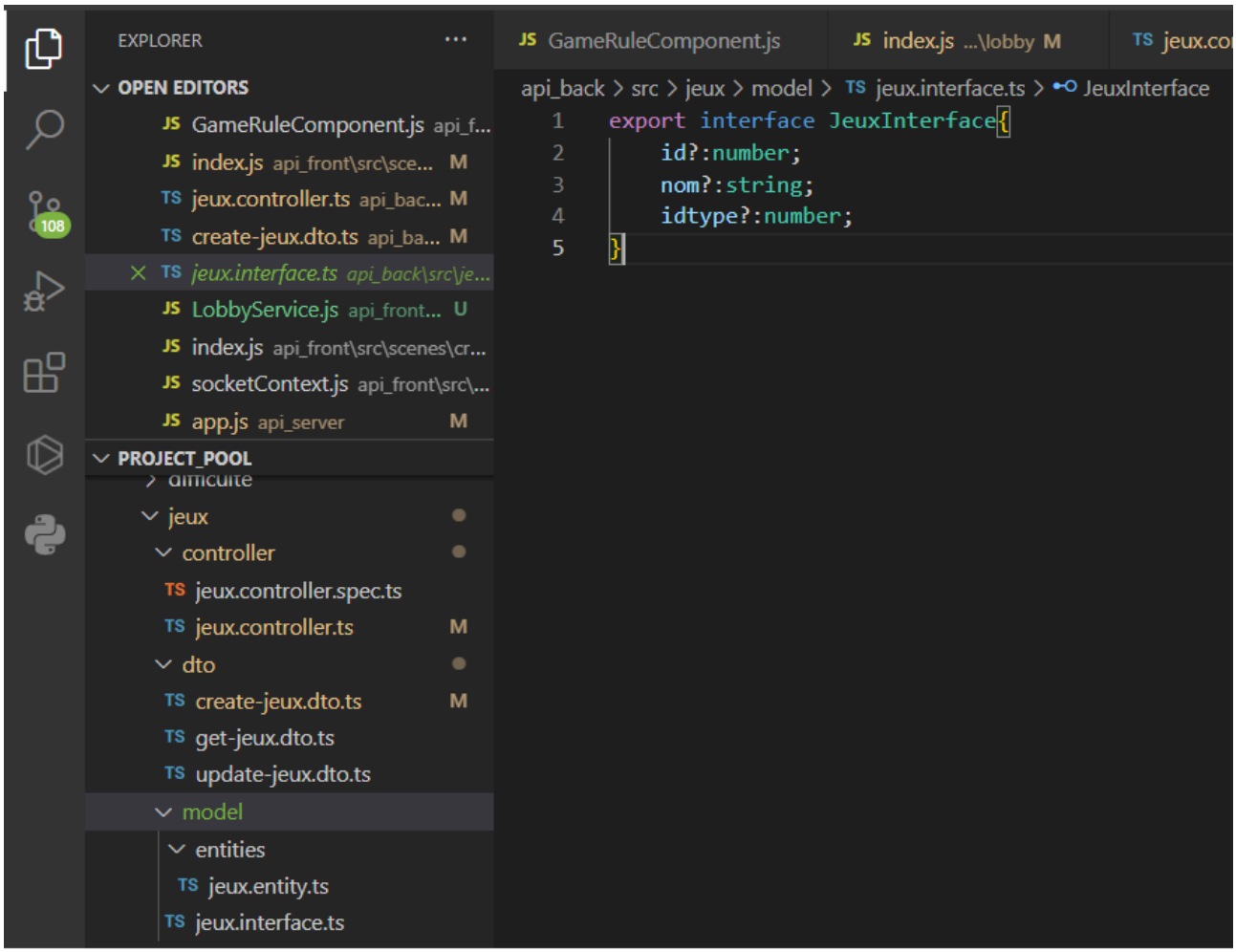


DTO ORM

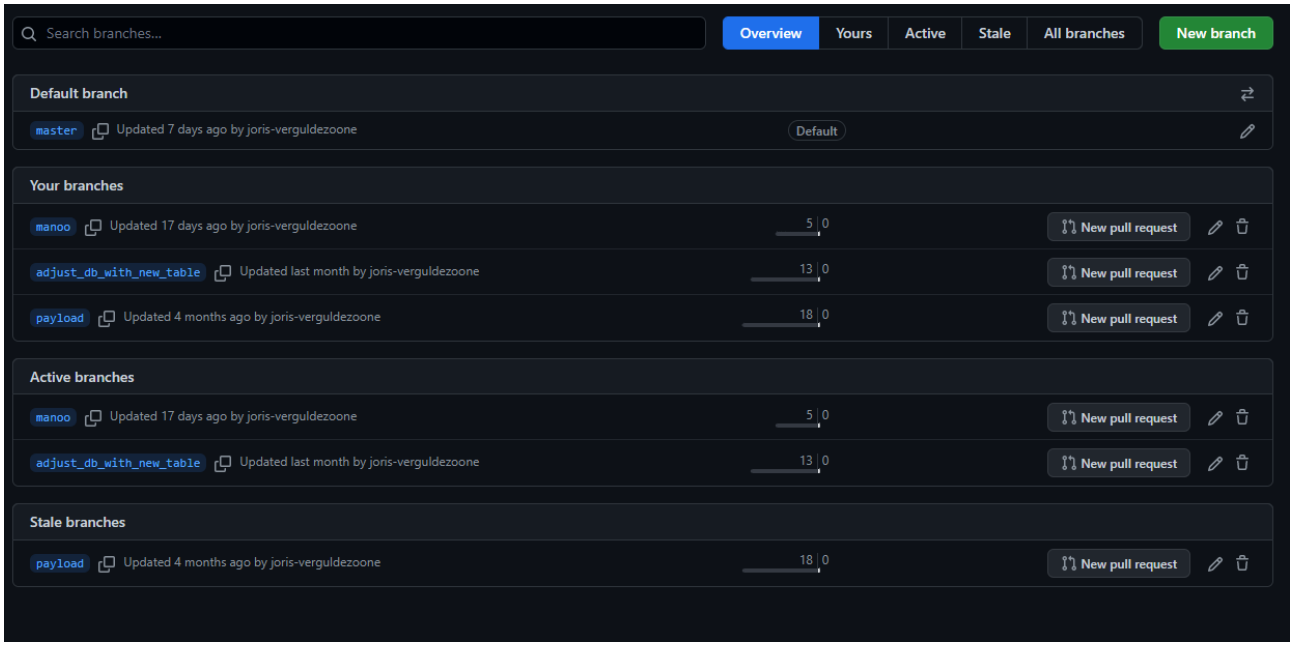


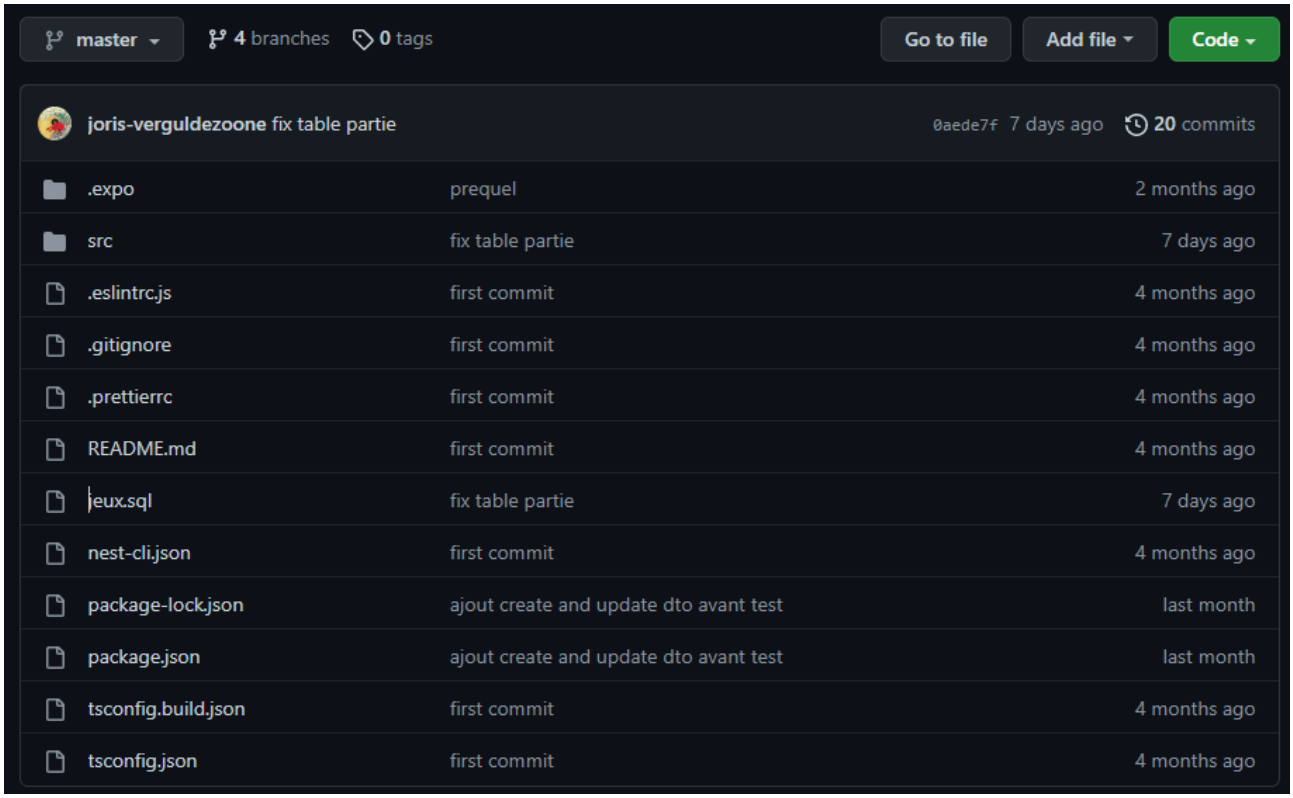
main config

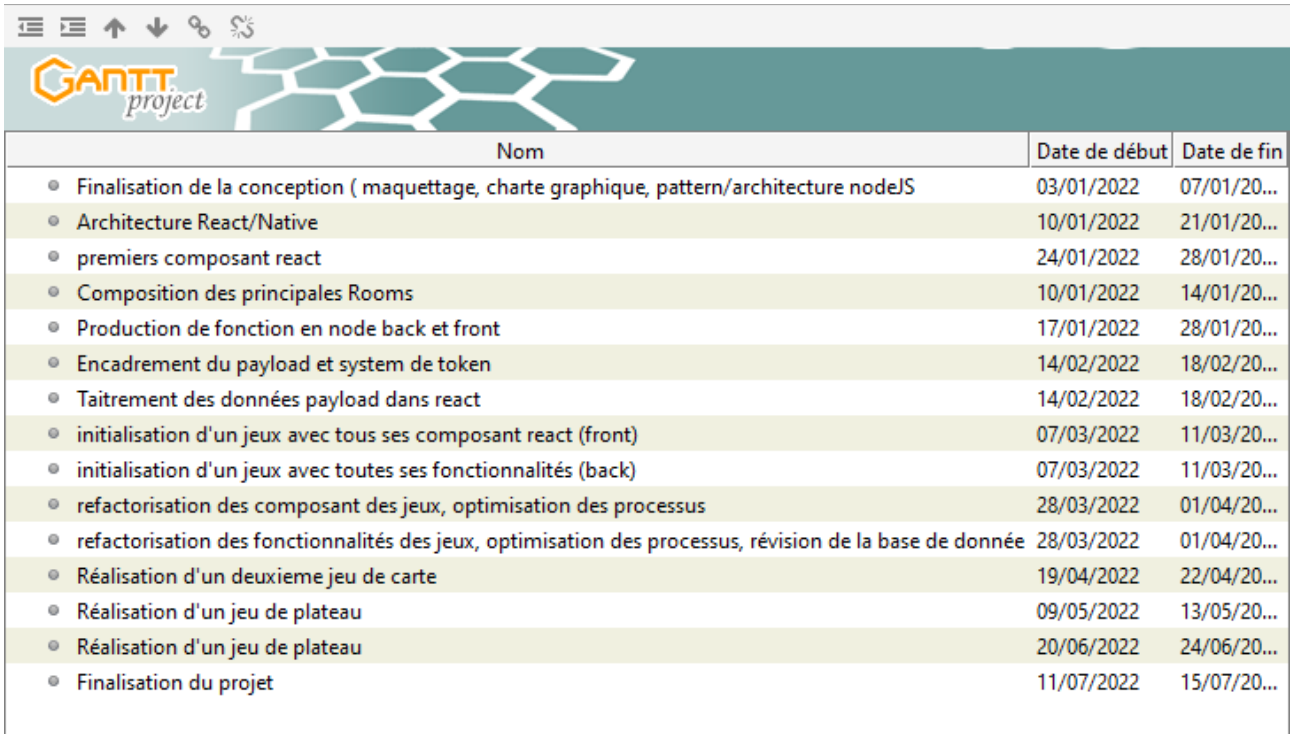
interface ORM



travail collaboratif


Search branches...
Overview
Yours
Active
Stale
All branches
New branch
Default branch
master Updated 7 days ago by joris-verguldezoone
Default
Your branches
manoo Updated 17 days ago by joris-verguldezoone 5 0 New pull request
adjust_db_with_new_table Updated last month by joris-verguldezoone 13 0 New pull request
payload Updated 4 months ago by joris-verguldezoone 18 0 New pull request
Active branches
manoo Updated 17 days ago by joris-verguldezoone 5 0 New pull request
adjust_db_with_new_table Updated last month by joris-verguldezoone 13 0 New pull request
Stale branches
payload Updated 4 months ago by joris-verguldezoone 18 0 New pull request


master
4 branches
0 tags
Go to file
Add file
Code
joris-verguldezoone fix table partie
0aede7f 7 days ago
20 commits
.expo prequel 2 months ago
src fix table partie 7 days ago
.eslintrc.js first commit 4 months ago
.gitignore first commit 4 months ago
.prettierrc first commit 4 months ago
README.md first commit 4 months ago
jeux.sql fix table partie 7 days ago
nest-cli.json first commit 4 months ago
package-lock.json ajout create and update dto avant test last month
package.json ajout create and update dto avant test last month
tsconfig.build.json first commit 4 months ago
tsconfig.json first commit 4 months ago



Nom	Date de début	Date de fin
Finalisation de la conception (maquettage, charte graphique, pattern/architecture nodeJS	03/01/2022	07/01/20...
Architecture React/Native	10/01/2022	21/01/20...
premiers composant react	24/01/2022	28/01/20...
Composition des principales Rooms	10/01/2022	14/01/20...
Production de fonction en node back et front	17/01/2022	28/01/20...
Encadrement du payload et system de token	14/02/2022	18/02/20...
Taitrement des données payload dans react	14/02/2022	18/02/20...
initialisation d'un jeux avec tous ses composant react (front)	07/03/2022	11/03/20...
initialisation d'un jeux avec toutes ses fonctionnalités (back)	07/03/2022	11/03/20...
refactorisation des composant des jeux, optimisation des processus	28/03/2022	01/04/20...
refactorisation des fonctionnalités des jeux, optimisation des processus, révision de la base de donnée	28/03/2022	01/04/20...
Réalisation d'un deuxieme jeu de carte	19/04/2022	22/04/20...
Réalisation d'un jeu de plateau	09/05/2022	13/05/20...
Réalisation d'un jeu de plateau	20/06/2022	24/06/20...
Finalisation du projet	11/07/2022	15/07/20...

Test Unitaires sur la base de données Users:

```
describe( name: 'UsersController', fn: () => {
  let controller: UsersController;
  let service: UsersService;
  const mockUsersService = {
    create: jest.fn( implementation: (dto) => {
      return {
        id: Date.now(),
        ...dto,
      };
    }),
    update: jest.fn( implementation: (id, dto) => ({
      id,
      ...dto,
    })),
    getTask: jest
      .fn()
      .mockImplementation( fn: (user :any ) =>
        Promise.resolve( value: { id: Date.now(), ...user }),
      ),
    getUsersWithFilters: jest
      .fn()
      .mockImplementation( fn: (user :any ) => Promise.resolve( value: { ...user })),
    remove: jest.fn().mockResolvedValue( value: 1),
  };
```

```
beforeEach( fn: async () => {
  const module: TestingModule = await Test.createTestingModule( metadata: {
    controllers: [UsersController],
    providers: [UsersService, User],
  }) TestingModuleBuilder
    .overrideProvider(UsersService) OverrideBy
    .useValue(mockUsersService) TestingModuleBuilder
    .compile();
  service = module.get<UsersService>(UsersService);
  controller = module.get<UsersController>(UsersController);
});
```

```
it( name: 'should be defined', fn: () => {
  expect(controller).toBeDefined();
});
it( name: 'should create a user', fn: () => {
  const dto = {
    username: 'termti',
    password: 'termti',
    idavatar: 1,
    role: 0,
  };
  expect(controller.create(dto)).toEqual( expected: {
    id: expect.any(Number),
    username: 'termti',
    password: 'termti',
    idavatar: 1,
    role: 0,
  });
});
it( name: 'should update a user', fn: () => {
  const dto = {
    id: 1,
    username: 'termta',
    password: 'termta',
    idavatar: 1,
    role: 0,
  };
  expect(controller.update( id: '1', dto)).toEqual( expected: {
    id: 1,
    ...dto,
```