

CardGamesApp

Dossier de Projet Professionnel

Mohamed Marwane Bellagha

BELLAGHA Mohamed
24/06/2022

Table des matières

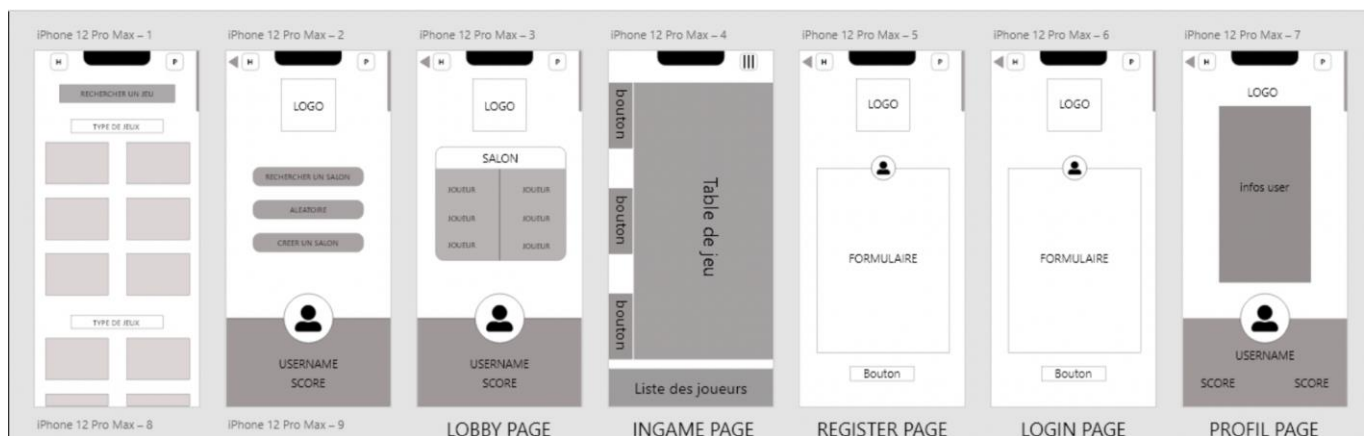
| | | |
|-------|--------------------------------------------------------|----|
| I. | Maquettage | 2 |
| II. | Composants d'accès aux données | 4 |
| III. | Concevoir et Mettre en place une base de données | 6 |
| IV. | Collaboration | 7 |
| V. | Composants métiers..... | 9 |
| VI. | Couches de l'application..... | 11 |
| VII. | Tests | 14 |
| VIII. | Deploiement | 15 |

I. 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 maquette de l'application,

Chaque page a été maquettée afin d'avoir une vision d'ensemble sur la structure de l'application :



Nous avons également réalisé une charte graphique afin de définir l'identité visuelle de notre application :

- Nous avons sélectionné les couleurs suivantes :



- Nous avons créé un logo pour l'application :



- Tous les composants comme les boutons pour naviguer, les boutons pour annuler/valider, les entrées utilisateur, auront un style fixe et défini à l'avance.



Nous avons également pensé à utiliser un Framework React Native pour styliser notre application,

'Native Base', qui est un Framework semblable à Bootstrap pour styliser un site web.

II. Composants d'accès aux données

Dans le cadre du développement de notre API nous avons décidé d'utiliser NestJS (nodejs). A l'aide de lignes de commande nous avons pu générer nos premiers fichiers pour créer tout ce qui servira au routage et aux services. Avec le temps nous avons réussi à nous approprier la logique et le fonctionnement du Framework. Avec la compréhension des DTO (Data Transfert Object) nous avons pu configurer des pipes basiques pour contenir et vérifier nos données avant de les passer à des Controller. Cela nous a aussi permis de configurer correctement SWAGGER pour pouvoir tester convenablement notre API et plus rapidement qu'avec POSTMAN. A l'aide de TypeOrm nous pouvons gérer nos entités, configurer les champs attendus, lier les entités et personnaliser leurs requêtes en fonction des comportements attendus.

Nous avons réussi à maîtriser comme il se doit toute la partie des services de NestJS. Cela nous a permis de personnaliser complètement les appels de notre API. Elle peut ainsi être appelée par n'importe quel champ sur n'importe quelle table. Certains champs en appellent d'autres grâce à des jointure qui sont effectuées en fonction de certains paramètres.

Notre API décode également les tokens qu'elle reçoit pour vérifier leur conformité. Dans une version future, l'API pourra posséder un système de rôle qui permettrait aux utilisateurs d'accéder aux services (routes) de l'API en fonction du rôle qui est contenu dans leur token. Actuellement n'importe qui peut accéder et utiliser l'API. Avec un système de rôle cela la rendrait complètement hermétique aux actions indésirables.

Dans un autre domaine nous avons créé un serveur socket.io en nodejs qui nous permet de communiquer avec la partie front de notre application dans le cadre d'échange dynamique de signaux et de data légères. Pour implémenter socket.io sur un serveur nodejs il nous a fallu installer CORS, axios (accéder à notre API), express (partitionner notre serveur en route) et socket.io (recevoir et émettre des événements). Nous aurions pu directement l'intégrer à notre server NestJS mais dans un souci de compréhension nous avons préféré effectuer cette partie a part.

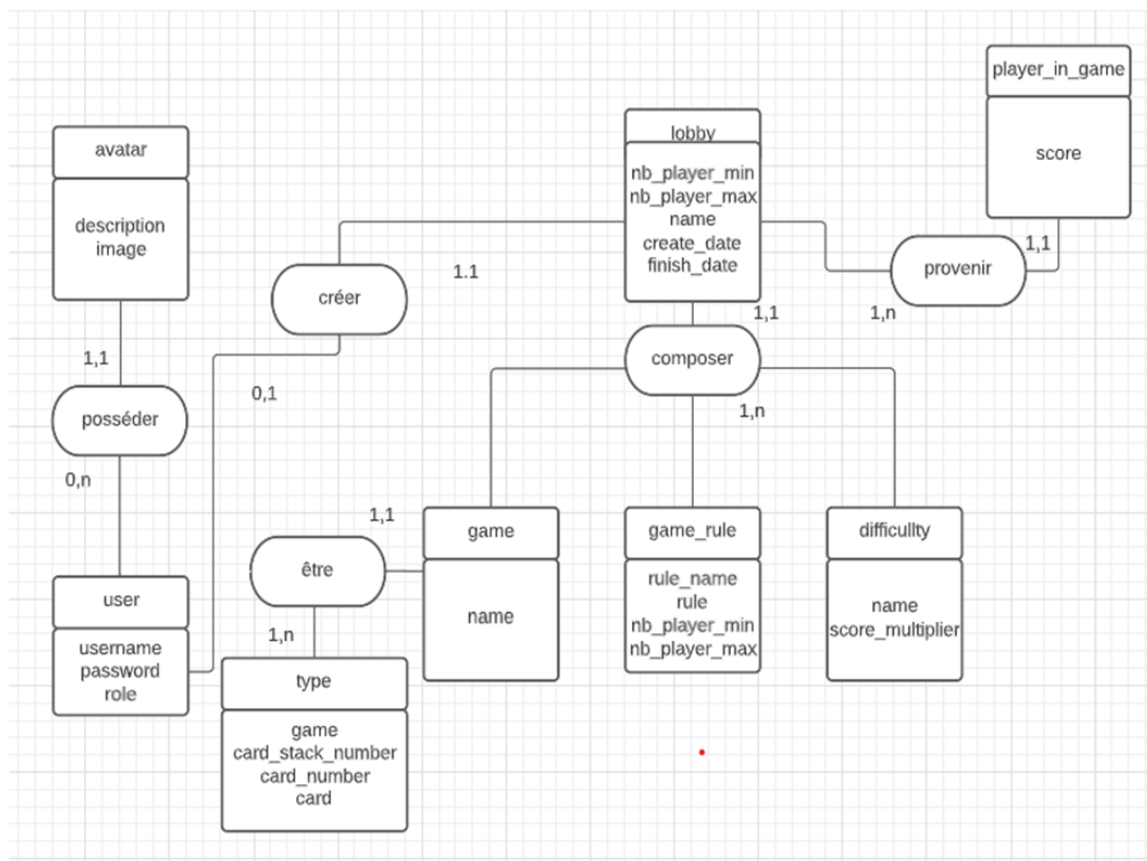
Pour accéder au server socket il suffit de l'URL du serveur sur laquelle est hébergée l'API et du numéro de port correspondant au serveur socket. A partir de là, l'utilisateur sera authentifiable par le serveur et pourra accéder au

différent événement lui permettant d'émettre et de recevoir des données en temps réel.

III. Concevoir et Mettre en place une 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 table, 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 != null 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.

IV. Collaboration

Lors du développement de ce projet nous étions quatre. Il a fallu organiser les tâches en fonction de notre planning d'alternant et en dehors des cours que l'on recevait. Nous avons tout d'abord brainstormé sur les différents sujets qui nous intéresseraient 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 personnes 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 fixé. 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. À 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.

Nous possédons 3 répertoires GitHub afin de versionner notre Front-end, Back-end, et le serveur socket(back). React Native, NestJS et les socket (NodeJS, Socket.io) ont donc tous été développés indépendamment.

À chaque fois qu'une personne devait développer une nouvelle fonctionnalité il lui a été donné de créer une nouvelle branche sur GitHub.

Une fois que les deux membres de la même sous équipe se mettent d'accord ils

peuvent merge sur le master ou l'un après l'autre si le travail a été bien intégré.

Concernant le Back-end, des tests sont effectués à chaque mise à jour de l'API à l'aide de SWAGGER que nous avons configuré.

Quand il s'agit du Front-end nous testons l'application sur téléphone et sur navigateur. La simulation sur navigateur nous permet de développer plus rapidement car nous n'avons pas besoin de télécharger les mises à jour sur téléphone qui sont longues. Cependant les compatibilités avec le simulateur du navigateur sont limitées et nous oblige à tester régulièrement avec le téléphone.

Le style dépend beaucoup du modèle de téléphone utilisé ce qui nous a obligé à faire attention lors des tests à ce que ce notre code soit d'autant plus compatible avec les autres supports.

Le serveur socket nécessite également une série de test dans laquelle on ouvre plusieurs sessions sur un même navigateur pour accumuler des utilisateurs connectés. Cependant cette façon de tester fonctionne uniquement lorsque l'on désactive le système de Token qui est directement dépendant du LocalStorage du téléphone, non-compatible avec le stockage local du navigateur. Encore une chose qui nous oblige à rallonger nos tests en aillant plusieurs téléphones connectés à l'application. On doit ainsi désactiver une partie de notre application ou prendre beaucoup de temps avec plusieurs téléphones pour pouvoir mettre à jour notre serveur socket.

V. Composants métiers

Un utilisateur se connecte, émet naturellement un évènement 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 créé. Une redirection s'effectue, les données sont envoyées au serveur en POST via le composant `CreateLobbyServices` et l'on est dirigé directement dans le composant général `Lobby` que l'on vient de créer.

A ce moment-là un emit (envoi 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. Une fois ces signaux effectués, le Lobby figurera dans la liste des Lobbies 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 existants qui sont tous répertoriés par leur nom de lobby qui est aussi le nom de la room associée en temps réel. L'utilisateur clique, et 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 quittent ou si la partie est lancée et terminée alors le lobby se supprimera par lui-même en socket avec l'évènement `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 requete 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.

VI. Couches de l'application

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 aillant 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 validée 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 à 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 appelle 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 composants métier réagissent en temps réel aux clics 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'envoi et la réception de message, les notifications et encore d'autres 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 tokens 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 a 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 composant dans une navigation Stack (react-navigation) dans laquelle on appel un composant vue qui sera constitué de plusieurs composant qui effectueront des actions plus ou moins indépendante 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 a 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.

VII. Tests

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 effectuée 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 fonctionnalités avec plusieurs types de données et plusieurs cas de figure afin de pouvoir s'assurer que notre application est bien sécurisée et marche comme on attend qu'elle marche

```
Test Suites: 2 failed, 2 passed, 4 total
Tests:      2 failed, 7 passed, 9 total
Snapshots:  0 total
Time:       10.217 s, estimated 12 s
Ran all test suites matching /users/i.

Watch Usage: Press w to show more.
```

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.

VIII. Deploiement

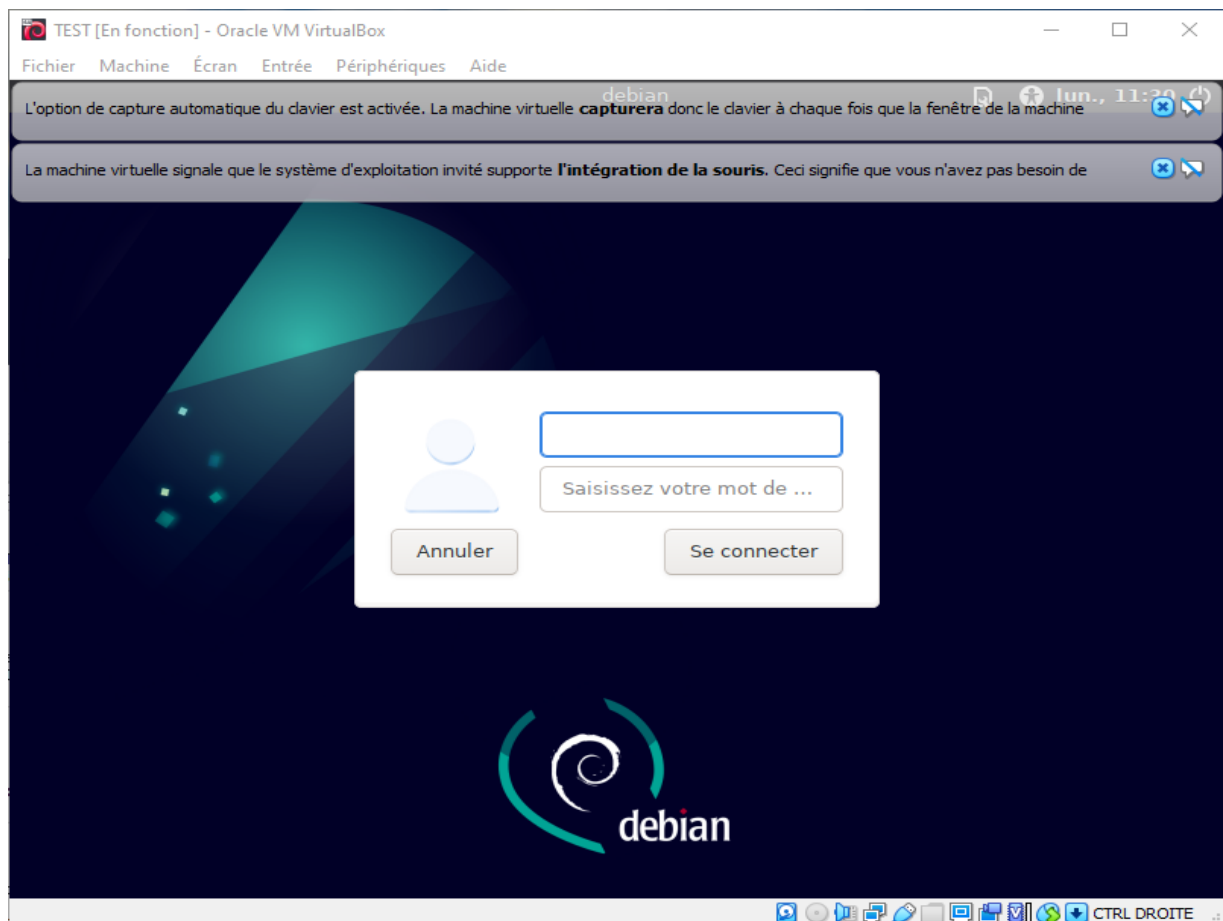
Dans le cadre du déploiement de notre API et de notre server socket nous avons utilisé un serveur distant. Dans un premier temps nous avons créé une Virtual Machine (VM) pour accéder au terminal SSH de notre server. Etant sur Window, nous ne possédons pas de terminal SSH, nous aurions pu en installer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

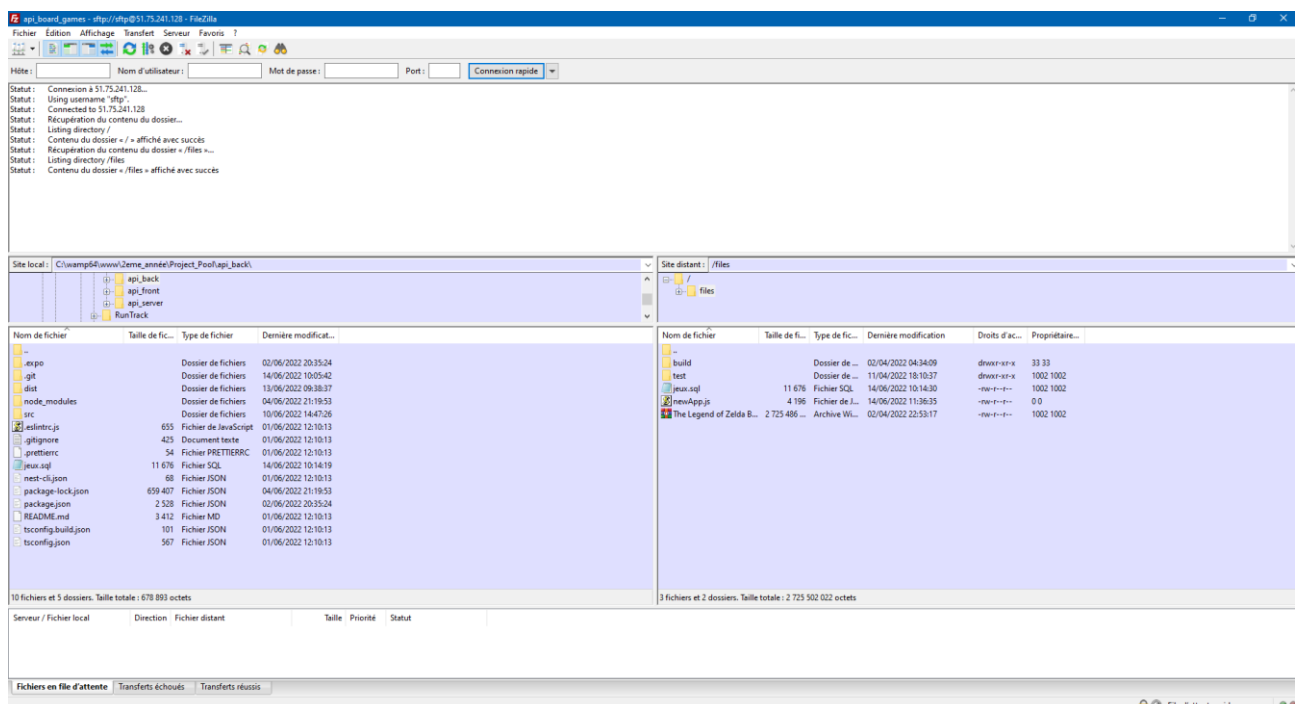
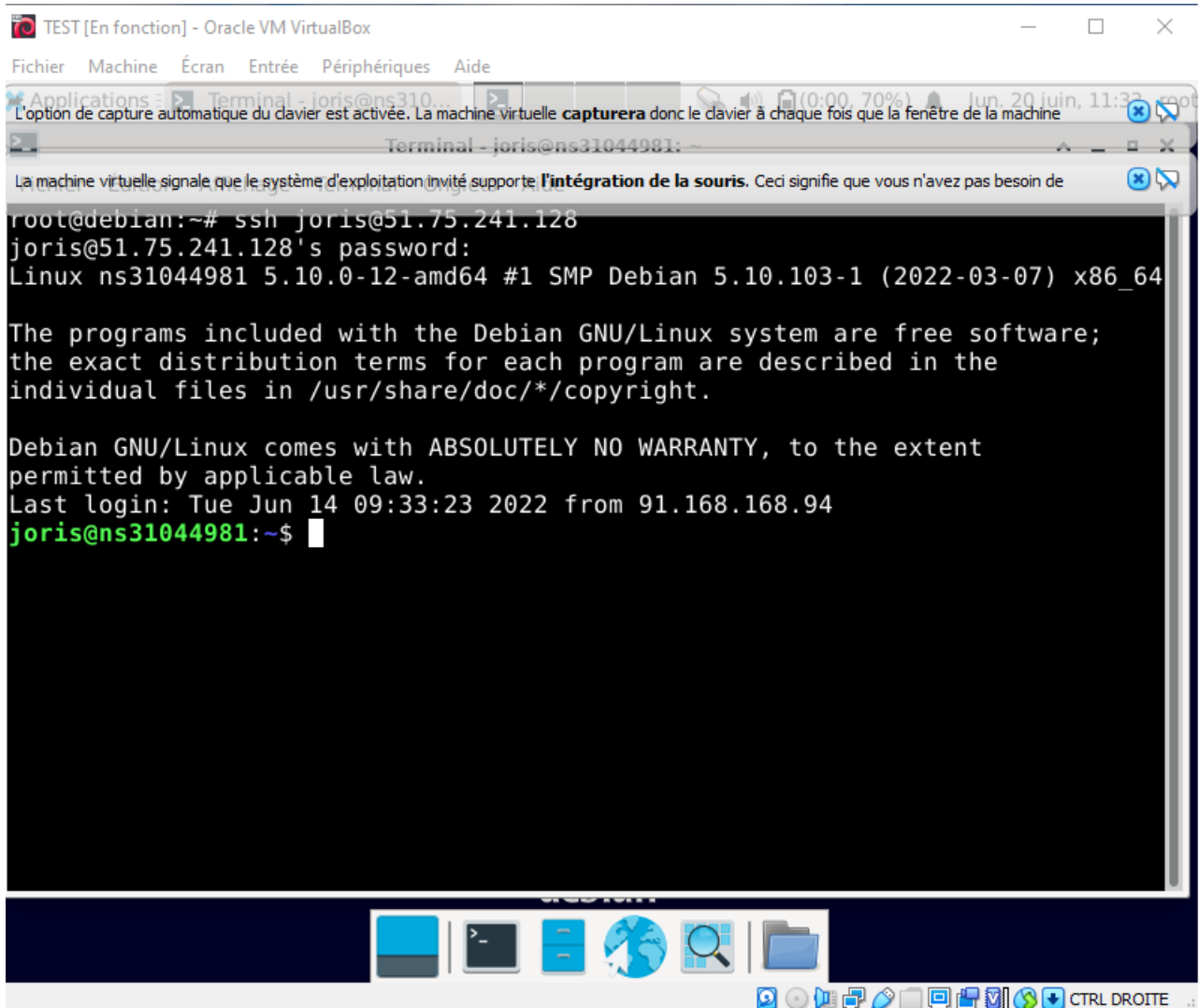
Avant de pouvoir procéder au déploiement de notre API il a fallu installer diverses technologies. On a tout d'abord installé NodeJS pour pouvoir utiliser NPM (gestion des paquets), NestJS (api sous nodejs). Puis apache2 même si par la suite on est passé sur Nginx. Et MariaDB pour gérer nos bases de données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





TEST [En fonction] - Oracle VM VirtualBox

Fichier Machine Écran Entrée Périphériques Aide

Applications: Terminal Xfce (0:00, 68%) lun. 20 juin, 11:37 root

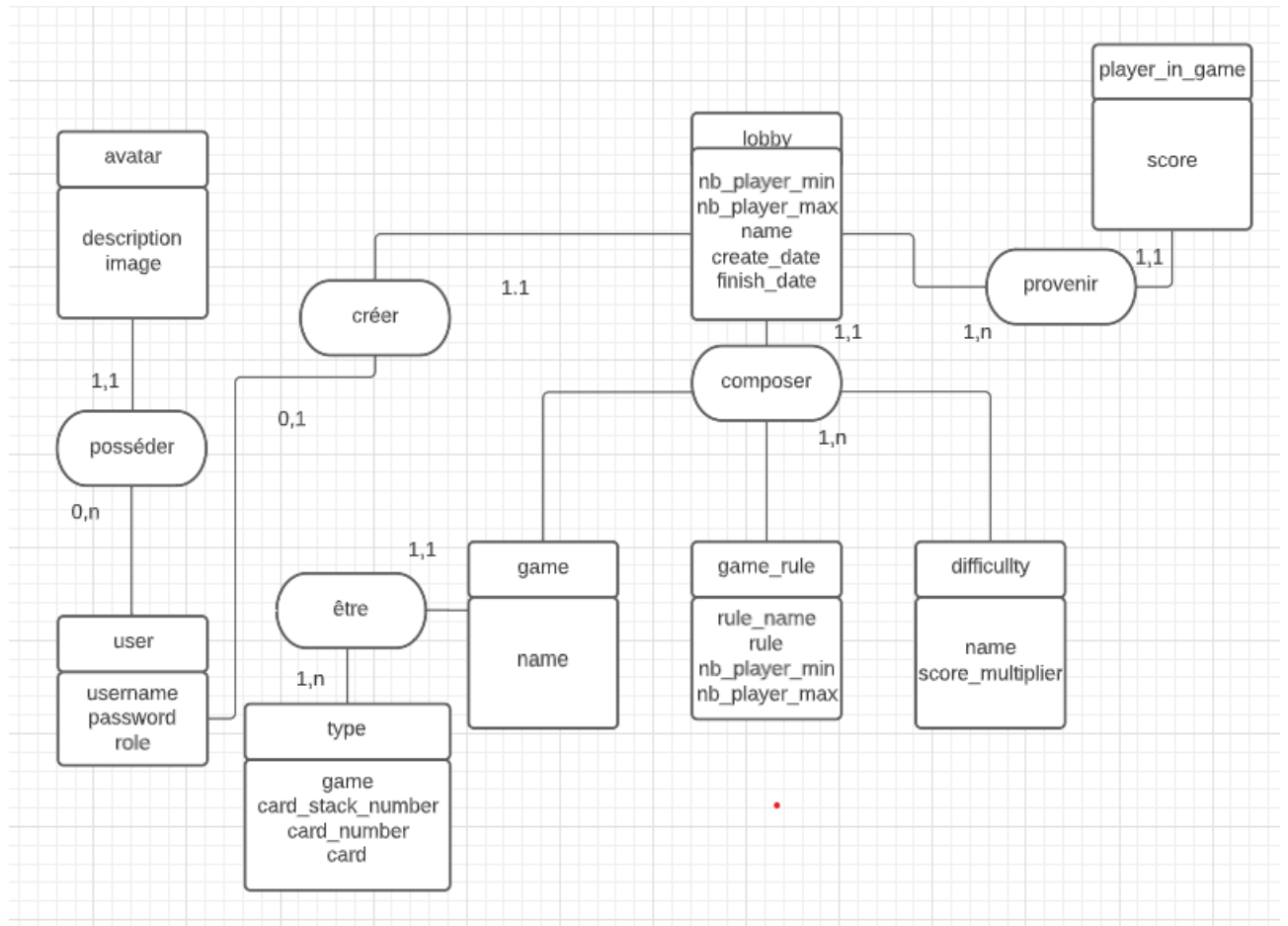
Terminal - joris@ns31044981: ~

Fichier Édition Affichage Terminal Onglets Aide

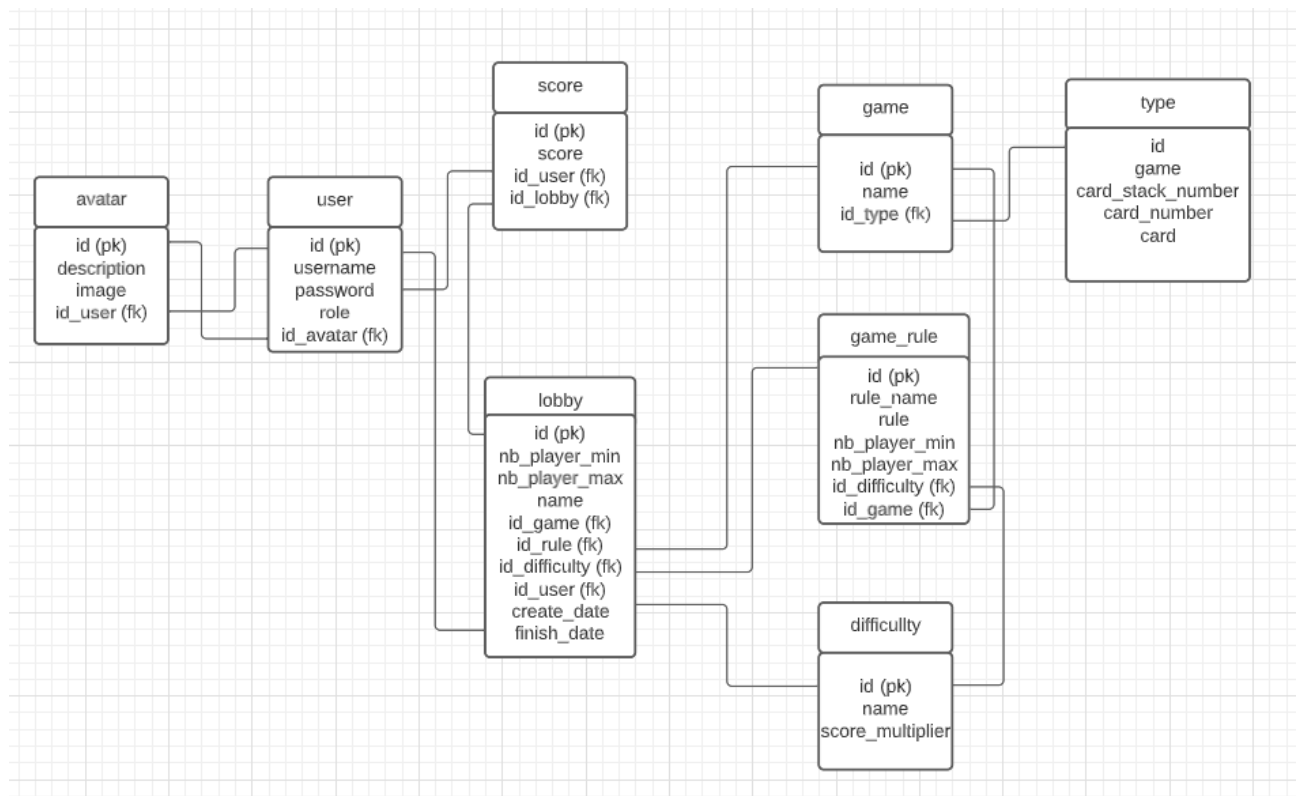
```
joris@ns31044981:/$ cd var
joris@ns31044981:/var$ ls
backups  lib      lock  mail  run  snap  tmp
cache    local   log   opt   sftp spool  www
joris@ns31044981:/var$ cd sftp
joris@ns31044981:/var/sftp$ ls
files
joris@ns31044981:/var/sftp$ cd files
joris@ns31044981:/var/sftp/files$ ls
'The Legend 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



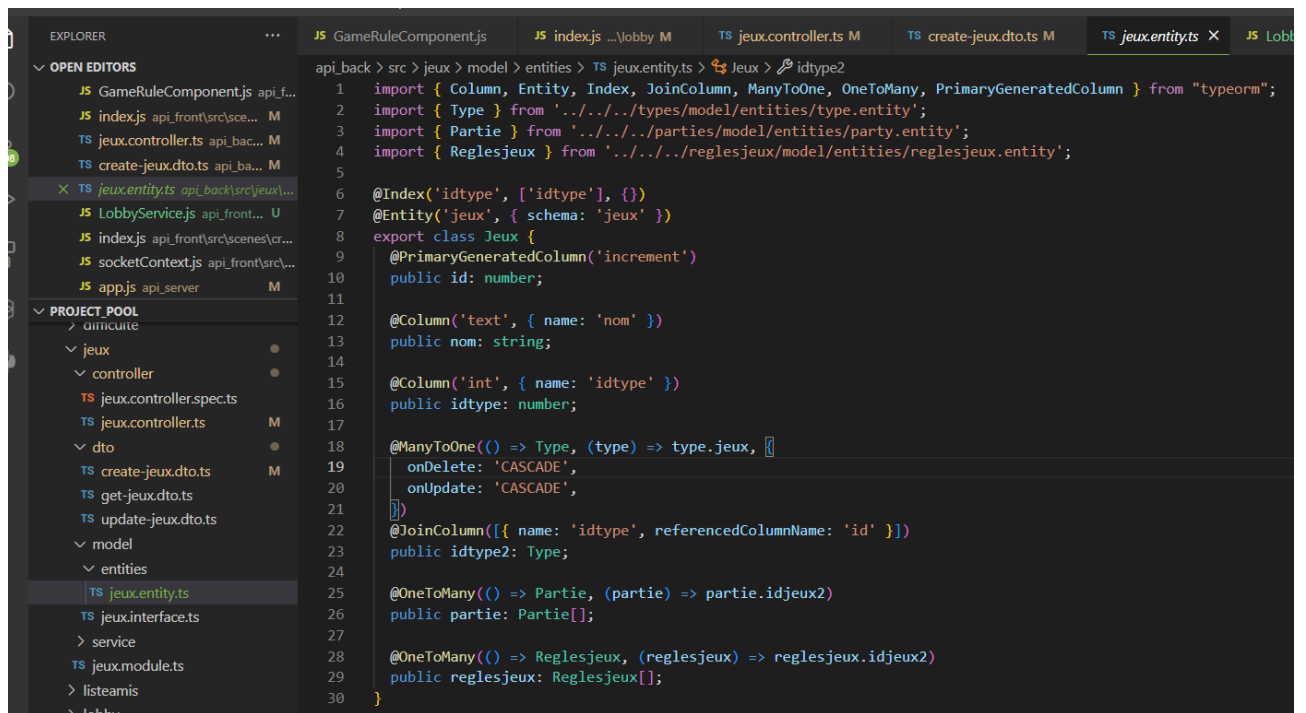
MLD

Développer les composant d'accès aux données :

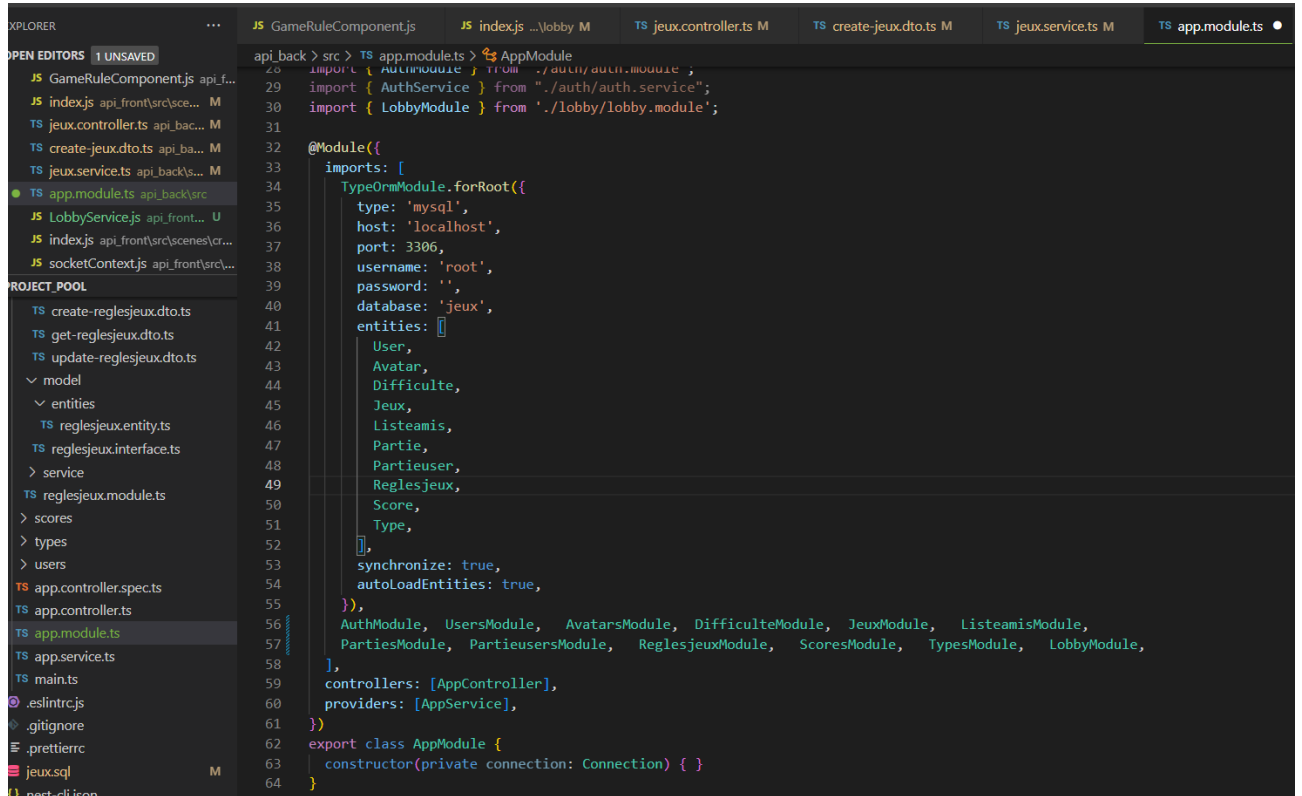
```
18
19
20 create(jeux: CreateJeuxDto): Promise<JeuxInterface> {
21   return this.jeuxRepository.save(jeux);
22 }
23
24 findAll(): Promise<Jeux[]> {
25   return this.jeuxRepository.find();
26 }
27
28 update(id: number, jeux: UpdateJeuxDto): Promise<any> {
29   return this.jeuxRepository.update(id, jeux);
30 }
31
32 remove(id: number): Promise<any> {
33   return this.jeuxRepository.delete(id);
34 }
35
36 async getGamesWithFilters(filterDto: GetJeuxDto): Promise<any> { // FILTER FUNCTION
37   /* La fonction renvoie désormais des innerJoin
38   Elle ne peut plus etre de type Promise<Jeux> et return autre chose */
39   console.log(filterDto); // typeof() avant autre vérification s'il y a un bug
40   console.log(filterDto.id)
41   const { nom, idtype, id } = filterDto;
42
43   if (id) {
44     const query = await createQueryBuilder('jeux', 'j')
45       .innerJoinAndSelect('j.reglesjeux', 'r')
46       .innerJoinAndSelect('j.idtype2', 't')
47       .innerJoinAndSelect('r.iddifficulte2', 'd')
48       .where('j.id =:id', { id: id })
49       .getOne(); // getMany() si on cherche plusieurs jeux et l'ensemble de leur innerJoin
50
51     console.log(query['reglesjeux'][0].iddifficulte2.difficulte) // exemple d'accessibilité au résultat
52     return query // il faudrait normaliser les Fetch
53   }
54
55   let jeux = await this.findAll();
56
57   if (nom) {
58     jeux = jeux.filter((task) => task.nom === nom);
59   }
60 }
```

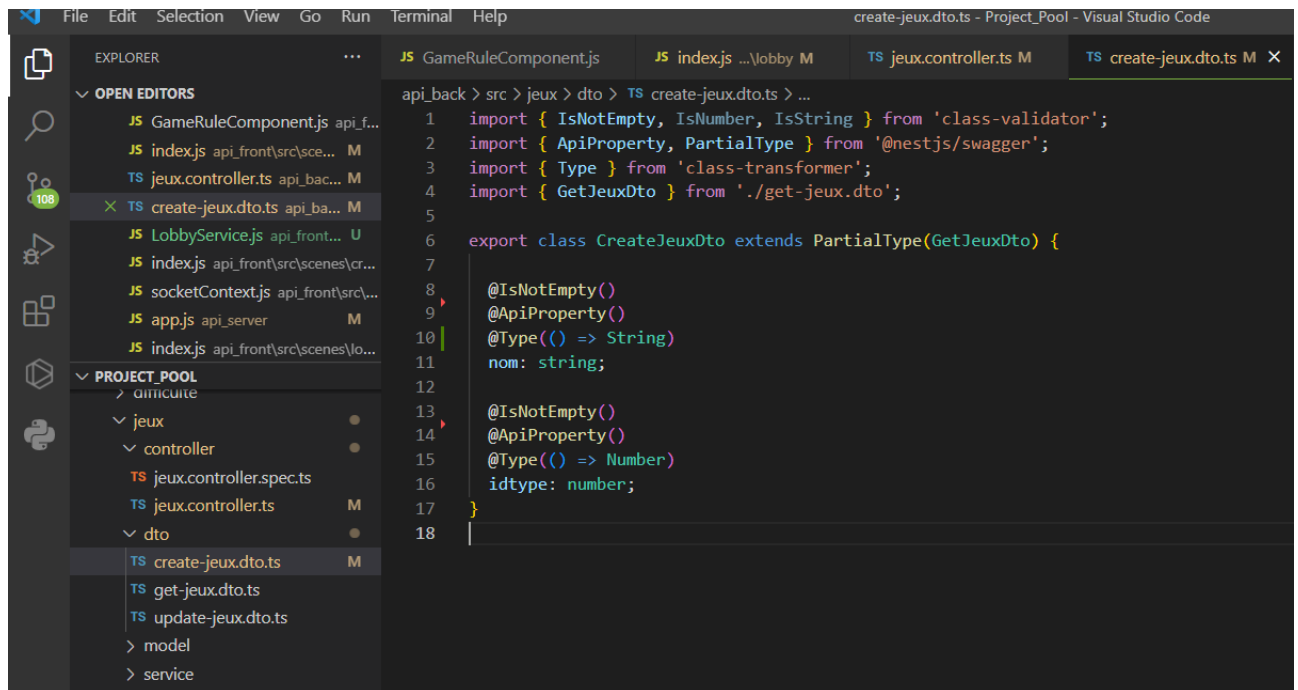
services - controller

```
1 import { Controller, Get, Post, Body, Patch, Param, Delete, Put, Query, } from '@nestjs/common';
2 import { JeuxService } from '../service/jeux.service';
3 import { ApiTags } from '@nestjs/swagger';
4 import { Observable } from 'rxjs';
5 import { JeuxInterface } from '../model/jeux.interface';
6 import { Jeux } from '../model/entities/jeux.entity';
7 import { CreateReglesjeuxDto } from '../reglesjeux/dto/create-reglesjeux.dto';
8 import { Reglesjeux } from '../reglesjeux/model/entities/reglesjeux.entity';
9 import { CreateJeuxDto } from '../dto/create-jeux.dto';
10 import { UpdateJeuxDto } from '../dto/update-jeux.dto';
11 import { GetJeuxDto } from '../dto/get-jeux.dto';
12
13 @ApiTags('jeux')
14 @Controller('jeux')
15 export class JeuxController {
16   constructor(private readonly jeuxService: JeuxService) { }
17
18   @Post()
19   create(@Body() jeux: CreateJeuxDto): Promise<JeuxInterface> {
20     return this.jeuxService.create(jeux);
21   }
22
23   @Put('/:id')
24   update(@Param('id') id: string, @Body() jeux: UpdateJeuxDto): Promise<any> {
25     return this.jeuxService.update(+id, jeux);
26   }
27
28   @Delete('/:id')
29   remove(@Param('id') id: string): Promise<Jeux> {
30     return this.jeuxService.remove(Number(id));
31   }
32
33   @Get('/:find')
34   getTask(@Query() filterDto: GetJeuxDto): Promise<Jeux[]> {
35     if (Object.keys(filterDto).length) {
36       return this.jeuxService.getGamesWithFilters(filterDto);
37     } else {
38       return this.jeuxService.findAll();
39     }
40   }
41 }
```

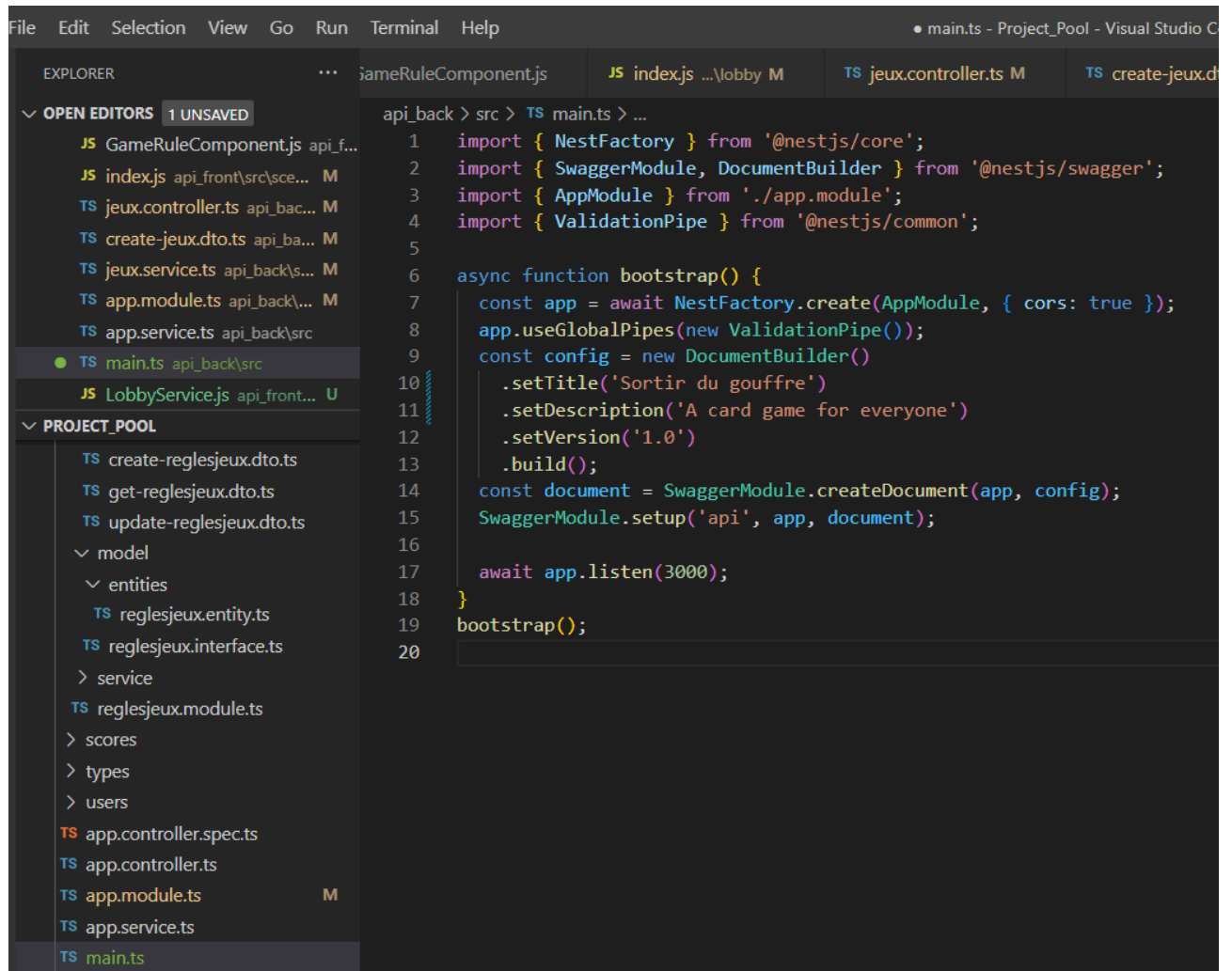


entity ORM





dto ORM



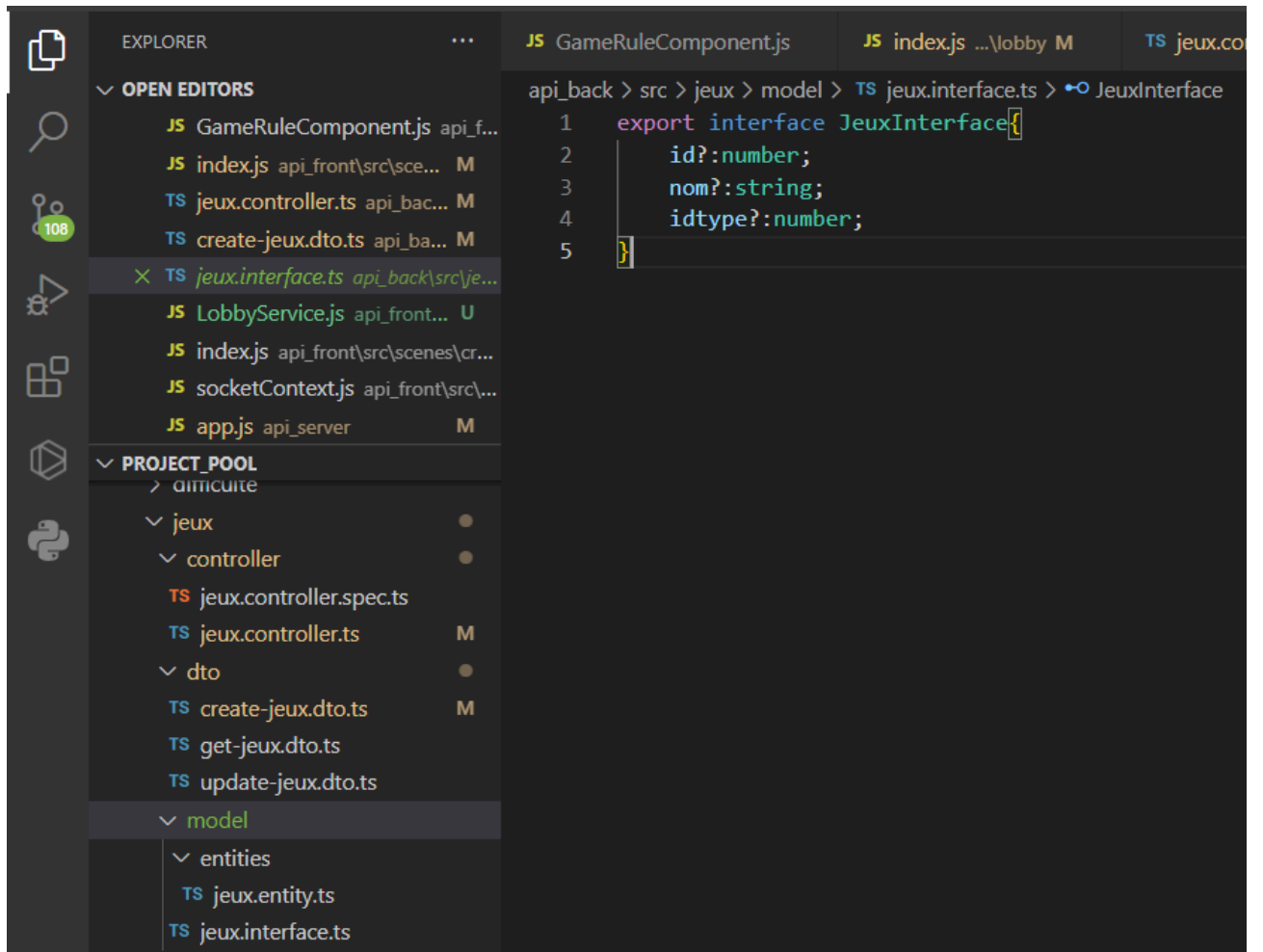
The image shows a Visual Studio Code editor window with the following components:

- Explorer Panel:** Displays the file structure of the project. The **PROJECT_POOL** folder is expanded, showing subfolders like **model**, **entities**, **service**, **scores**, **types**, and **users**. Files include **GameRuleComponent.js**, **index.js**, **jeux.controller.ts**, **create-jeux.dto.ts**, **jeux.service.ts**, **app.module.ts**, **app.service.ts**, **main.ts** (selected), and **LobbyService.js**.
- Editor Panel:** Shows the content of **main.ts**. The code is as follows:

```
1 import { NestFactory } from '@nestjs/core';
2 import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
3 import { AppModule } from './app.module';
4 import { ValidationPipe } from '@nestjs/common';
5
6 async function bootstrap() {
7   const app = await NestFactory.create(AppModule, { cors: true });
8   app.useGlobalPipes(new ValidationPipe());
9   const config = new DocumentBuilder()
10     .setTitle('Sortir du gouffre')
11     .setDescription('A card game for everyone')
12     .setVersion('1.0')
13     .build();
14   const document = SwaggerModule.createDocument(app, config);
15   SwaggerModule.setup('api', app, document);
16
17   await app.listen(3000);
18 }
19 bootstrap();
20
```

main config

interface ORM



Travail collaboratif :

Q Search branches...

OverviewYoursActiveStaleAll branchesNew branch

Default branch

master

Updated 7 days ago by joris-verguldezoone

Default

Your branches

manoo

Updated 17 days ago by joris-verguldezoone

50

New pull request

adjust_db_with_new_table

Updated last month by joris-verguldezoone

130

New pull request

payload

Updated 4 months ago by joris-verguldezoone

180

New pull request

Active branches

manoo

Updated 17 days ago by joris-verguldezoone

50

New pull request

adjust_db_with_new_table

Updated last month by joris-verguldezoone

130

New pull request

Stale branches

payload

Updated 4 months ago by joris-verguldezoone

180

New pull request

master

4 branches

0 tags

Go to file

Add file

Code

joris-verguldezoone

fix table partie

0aede7f7 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 |

| GANTT project | | |
|-----------------------------------------------------------------------------------------------------------|---------------|-------------|
| 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... |
| • Traitement 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,
  });
});

```