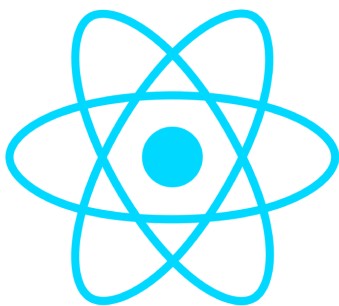




CBG Games



React Native



nest



socket.io

1. Spécifications fonctionnelles.....	3
1.1. Périmètre du projet.....	3
1.2. Arborescence du site.....	3
1.3. Description des fonctionnalités.....	3
1.3.1 – Inscription.....	3
1.3.2 – Connexion.....	3
1.3.3 – Profil	3
1.3.4 – Création de lobby	4
1.3.5 – Liste des lobbies	4
1.3.6 - Lobby	4
1.4. Jeu d’essai.....	4
2 – Spécification technique (travail collaboratif Bdd Architecture).....	6
2.1 – Travail collaboratif.....	6
2.2 BDD.....	9
2.2.1 - MCD.....	10
2.2.2 - MLD	11
2.3 - Les composants d’accès aux données :.....	11
2.4 - Développer une application n-tier :	13
2.4.1 – Back-end	13
2.4.2 - Server socket.....	15
2.4.3 – Front-end.....	16
2.5 - Charte graphique :.....	17
2.6 - Préparer et exécuter le déploiement d’une application.....	19
2.7. Sécurité.....	20
2.8 – Test unitaire.....	23
3. - Annexe.....	25
3.1 – BDD	25
3.2 – Les composants d’accès aux données	26
3.3 – Préparer et exécuter le déploiement d’une application	30

1. Spécifications fonctionnelles

1.1. Périmètre du projet

Cette application mobile a été réalisée en français et est pratiquement compatible au web grâce à expo qui nous permet d'émuler notre react native en react sur navigateur.

L'application est destinée aux personnes voulant jouer aux cartes en ligne et en temps réel avec d'autres personnes sur leur téléphone mobile.

1.2. Arborescence du site

L'arborescence du site se décline comme suit :

- Page d'accueil
- Page inscription
- Page connexion
- Page mon profil
- Création d'un lobby
- Liste des lobby
- Lobby

1.3. Description des fonctionnalités

1.3.1 – Inscription

L'utilisateur peut s'inscrire directement depuis le téléphone à travers un formulaire d'inscription en React Native.

Le mot de passe sera encrypté dans notre back-end.

1.3.2 – Connexion

Connexion classique qui délivre un token à l'utilisateur si elle est réussite.

1.3.3 – Profil

Contient un formulaire pour changer les champs de l'utilisateur.
Affiche également l'ensemble des scores obtenu par l'utilisateur et les parties auxquels il a participé

1.3.4 – Création de lobby

Permet à l'utilisateur de créer une partie en ligne en choisissant le type de jeu, les règles, la difficulté adéquat (qui multipliera le score), la nomination du jeu en elle-même ainsi que la nomination du lobby de création de partie

1.3.5 – Liste des lobbies

Affiche l'ensemble des lobbies disponibles en temps réel. La liste des lobbies ne se charge qu'une fois mais en base de données tout est actualisé en temps réel. Si un lobby n'existe plus ou s'il est fini alors il ne s'affichera pas dans la liste des lobbies à moins que l'on y reste dans la page liste des lobbies sans la rafraichir.

En cliquant sur un lobby l'utilisateur pourra le rejoindre sauf si le server repère le lobby comme étant plein ou déjà en cours de jeu.

1.3.6 - Lobby

Est un lieu dans lequel les utilisateurs se rassemblent pour jouer. Si le lobby possède suffisamment de joueurs et qu'ils sont prêts alors la partie sera valide et lancée. A ce moment une `starting_date` est mutée dans la base de données à partir de la ligne qui correspond au lobby. Cette mise à jour permet au serveur de distinguer si un lobby n'est plus accessible car il a commencé. C'est également une étape importante avant qu'un quelconque algorithme de jeu ne se lance chez les utilisateurs.

1.4. Jeu d'essai

Un utilisateur se connecte, il émet naturellement un événement de connexion sur le serveur socket qui l'authentifie (`socket_id`) et l'utilisateur reçoit un token du back-end qui lui permet d'accéder à l'ensemble des espaces de l'application.

La création et l'accessibilité des lobbies :

L'utilisateur, authentifié, a 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 associées au jeu sélectionné.

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

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 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, et rejoint ainsi le lobby et répète les signaux qu'a effectué 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 le troisième joueur pourra rejoindre le lobby. 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'événement natif 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 données via notre API utilisé avec axios dans le serveur socket.

Ainsi, la liste des lobby ne restera jamais remplie de serveur indisponible et sera toujours mise à jour avec soit les lobby disponibles soit les lobby pleins ou déjà en cours de jeu.

2 – Spécification technique (travail collaboratif Bdd Architecture)

2.1 – Travail collaboratif

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.

Au début de ce projet il y avait beaucoup de technologies que nous ne maîtrisions 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.

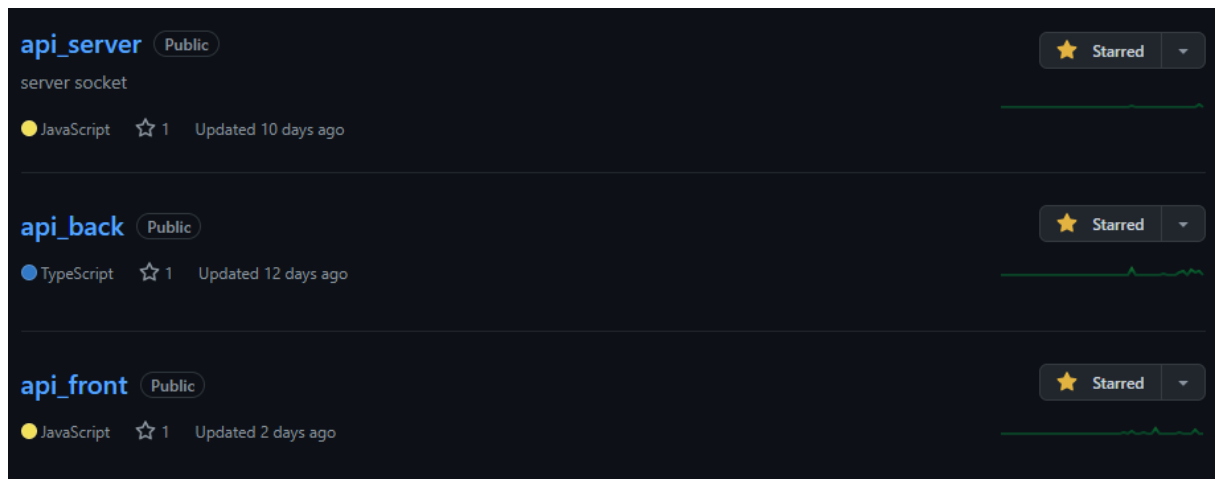
Pour connaître les tâches à long termes que nous aurions à réaliser nous avons produit un diagramme de **Gantt**. Celui-ci nous permet de visualiser toutes les tâches à effectuer sur une sorte de planning annuel. Ainsi à chaque fois que l'on rentrait d'alternance nous savions sur quoi nous concentrer et quelle serait la prochaine étape à remplir avant de pouvoir s'intéresser à une autre technologie ou fonctionnalité.

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...

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 puissent 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 créer. 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.



A 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 mette 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.

A screenshot of the Swagger API interface. At the top, there's a 'type' label and a list of endpoints: POST /types, PUT /types/{id}, DELETE /types/{id}, and GET /types/find. The GET /types/find endpoint is selected and expanded. Below it, there's a 'Parameters' section with a table of query parameters. At the bottom, there are 'Execute' and 'Clear' buttons.

Name	Description
id number (query)	id
typedejoux string (query)	cartes
nbdejoux number (query)	nbdejoux
nbcartes number (query)	nbcartes
typedecarte string (query)	typedecarte

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.

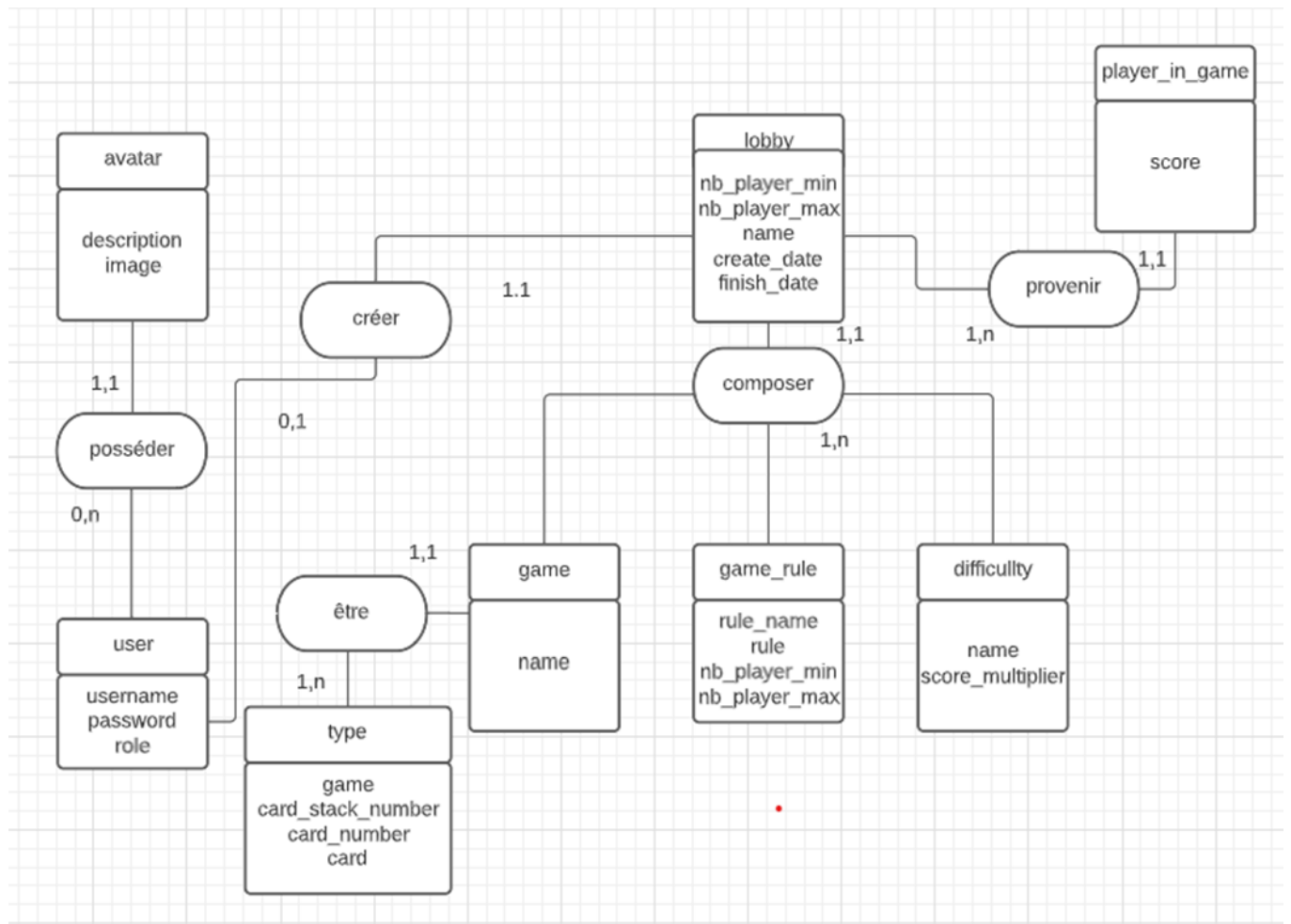
Le serveur socket nécessite également une série de test dans laquelle on ouvre plusieurs sessions sur un même navigateur pour accumuler des utilisateurs connectés. Cependant cette façon de tester fonctionne uniquement lorsque l'on désactive le système de Token qui est directement dépendant du LocalStorage 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.

2.2 BDD

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.

2.2.1 - MCD



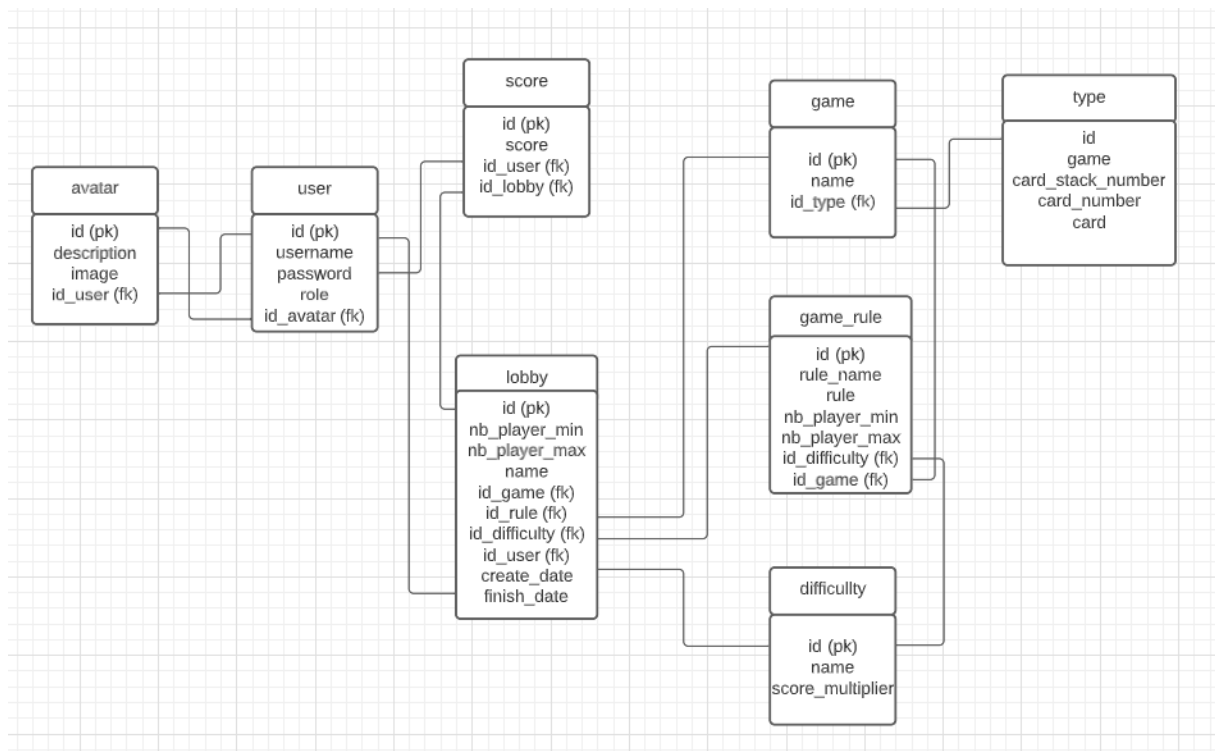
Pour lobby, petite explication de son état :

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

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 sur son propre événement disconnect (car il ne contient aucun socket utilisateur) via le serveur socket pour annuler le lobby créer.

2.2.2 - MLD



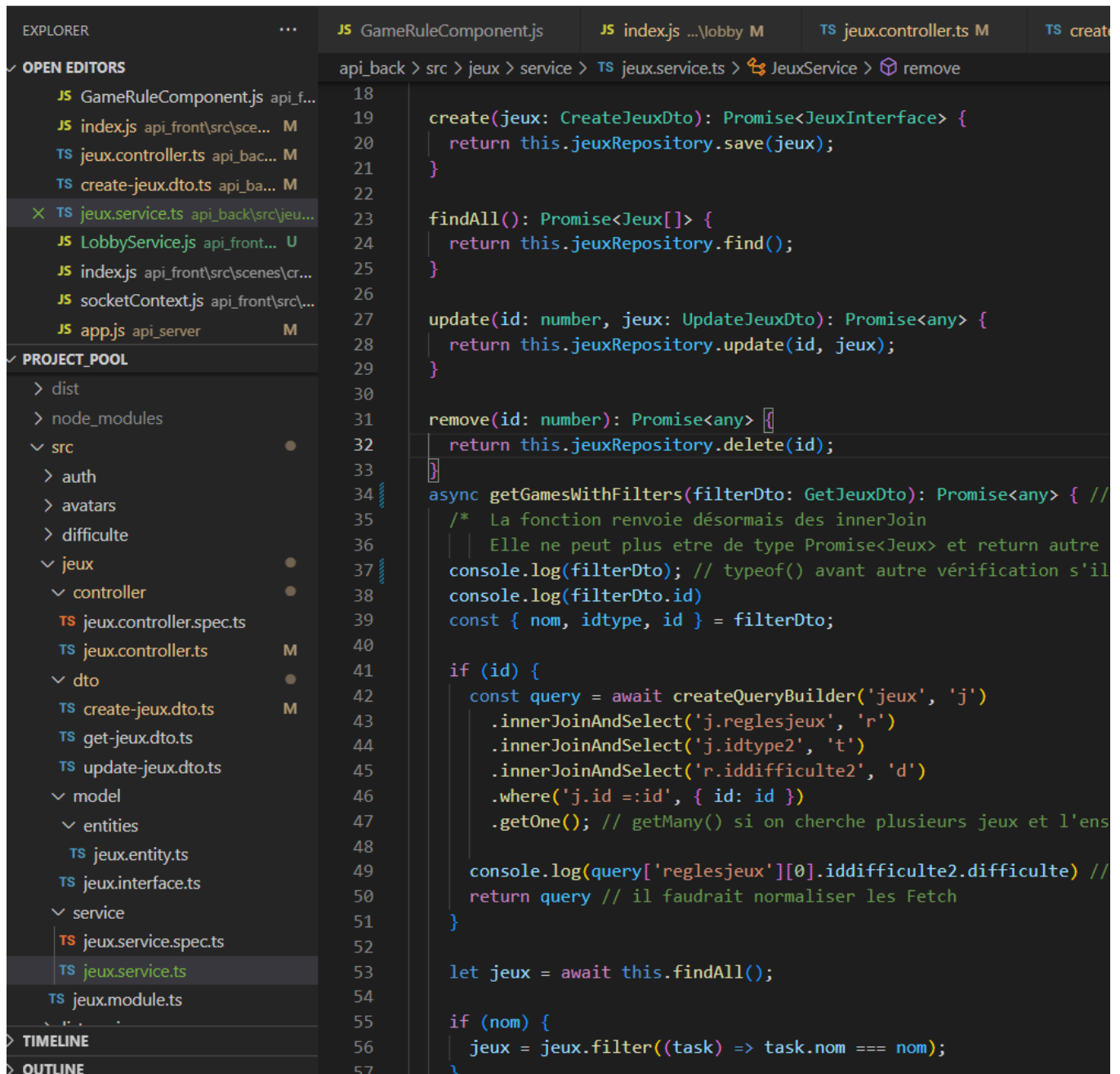
Toutes nos relations sont configurées en cascade ce qu'il fait que si une entrée d'une entité est update ou delete alors elle fera subir les mêmes actions à l'entrée de la table qui lui liée et qui lui correspond.

2.3 - Les composants d'accès aux données :

Dans le cadre du développement de notre **API** nous avons décidé d'utiliser **NestJS** (nodejs). A l'aide de ligne 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 attendu.

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 jointures qui sont effectuées en fonction de certains paramètres.



```
api_back > src > jeux > service > TS jeux.service.ts > JeuxService > remove

18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57

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);
}

async getGamesWithFilters(filterDto: GetJeuxDto): Promise<any> { //
  /* La fonction renvoie désormais des innerJoin
  Elle ne peut plus être de type Promise<Jeux> et return autre
  console.log(filterDto); // typeof() avant autre vérification s'il
  console.log(filterDto.id)
  const { nom, idtype, id } = filterDto;

  if (id) {
    const query = await createQueryBuilder('jeux', 'j')
      .innerJoinAndSelect('j.reglesjeux', 'r')
      .innerJoinAndSelect('j.idtype2', 't')
      .innerJoinAndSelect('r.iddifficulte2', 'd')
      .where('j.id =:id', { id: id })
      .getOne(); // getMany() si on cherche plusieurs jeux et l'ens

    console.log(query['reglesjeux'][0].iddifficulte2.difficulte) //
    return query // il faudrait normaliser les Fetch
  }

  let jeux = await this.findAll();

  if (nom) {
    jeux = jeux.filter((task) => task.nom === nom);
  }
}
```

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 hermétiques aux actions indésirables.

Dans un autre domaine nous avons créé un **server socket.io** en nodejs qui nous permet de communiquer avec la partie front de notre application dans le cadre **d'échange en temps réel 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 préférés effectuer cette partie à 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 aux différents événements lui permettant d'émettre et de recevoir des données en temps réel.

2. 4 - Développer une application n-tier :

2.4.1 – Back-end

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 validé par TypeOrm et la configuration qu'on lui a attribué.

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.

2.4.2 - Server 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 au front, il appelle parfois l'API pour donner des informations supplémentaires 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 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'envoi et la réception de message, les notifications et encore d'autres fonctionnalités à venir...

```
app.module.ts M package.json TS app.service.ts TS main.ts JS LobbySer
r > JS app.js > ...

socket.on("disconnecting", async () => {
  console.log('disconnecting')
  console.log(socket.rooms, "the Set contains at least the socket ID"); // the
  rooms = io.of("/").adapter.rooms;
  console.log(rooms, 'stop rooms disconnection')
});

socket.on("disconnect", () => {
  console.log(LobbyObj.nomLobby, "LobbyObj.nomLobby")
  let nomLobby = LobbyObj.getNomLobby();
  var clients = io.sockets.adapter.rooms.get(LobbyObj.nomLobby);
  console.log(clients)
  if (clients == undefined) {
    console.log("client on est passé dedans y'a plus qu'à delete")
    getAny_DeleteByName('lobby/find?nomLobby=' + nomLobby)
  }
  else {
    console.log("else ", clients, 'il doit rester des socket dans le lobby')
  }

  console.log("Client disconnected");
  clearInterval(interval);
});

const LobbyObj = new Lobby(socket, 'toDefine');
socket.on("join_lobby", value => {

  let result = LobbyObj.createLobby(value)
  console.log(result)
})

socket.on("join_lobby_validate", data => { // si a partie a été créer
  let result = LobbyObj.joinLobbyAccess(data)
  LobbyObj.setNomLobby(data)

  console.log(result)
})
});
```

2.4.3 – Front-end

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 refactorisation du code.

Nous sommes donc restés sur une imbrication assez classique de nos composant dans une **navigation Stack (react-navigation)** dans laquelle on appelle 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 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.


```

import ClientComponent from './src/component/ClientComponent';
import { SOCKET_URL } from './src/const';
import { SocketContext, socketIo } from './src/scenes/socketContext';

const Stack = createNativeStackNavigator();

export default function App() {
  return (
    <>
      <NavigationContainer>
        <Stack.Navigator>
          <Stack.Screen name="HomeScreen" component={HomeScreen} options={{ title: "Accueil" }} />
          <Stack.Screen name="RegisterScreen" component={RegisterScreen} options={{ title: 'Register' }} />
          <Stack.Screen name="LoginScreen" component={LoginScreen} options={{ title: 'Connexion' }} />
          <Stack.Screen name="LobbyScreen" component={LobbyScreen} options={({ route, navigation }) => ({ title: 'Lobby' }} />
          <Stack.Screen name="ProfilScreen" component={ProfilScreen} options={{ title: 'Profil' }} />
          <Stack.Screen name="CreateLobbyScreen" component={CreateLobbyScreen} options={{ title: 'CreateLobby' }} />
          <Stack.Screen name="LobbyList" component={LobbyList} options={{ title: 'LobbyList' }} />
        </Stack.Navigator>
      </NavigationContainer>
    </>
  );
}

```

2.5 - Charte graphique :

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 sélectionnés les couleurs suivantes:



- Nous avons créé un logo pour l'application :



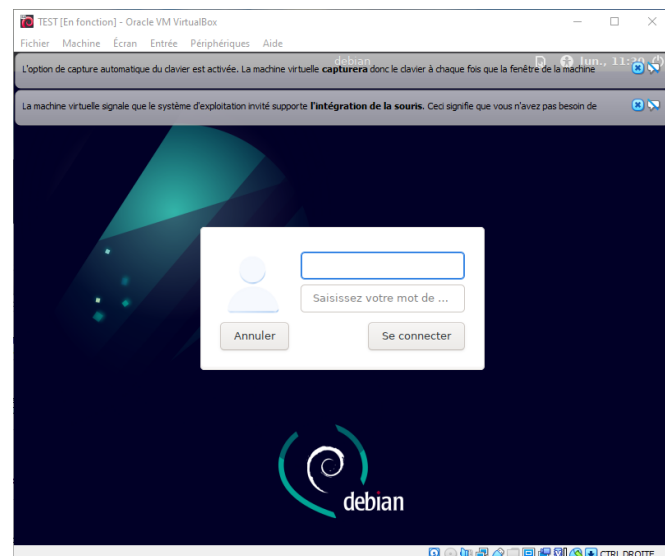
- Tous les composants comme les boutons pour naviguer, les boutons pour annuler/valider, les entrées utilisateur, auront un style fixe et défini à l'avance.

CONNEXION

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.

2.6 - Préparer et exécuter le déploiement d'une application

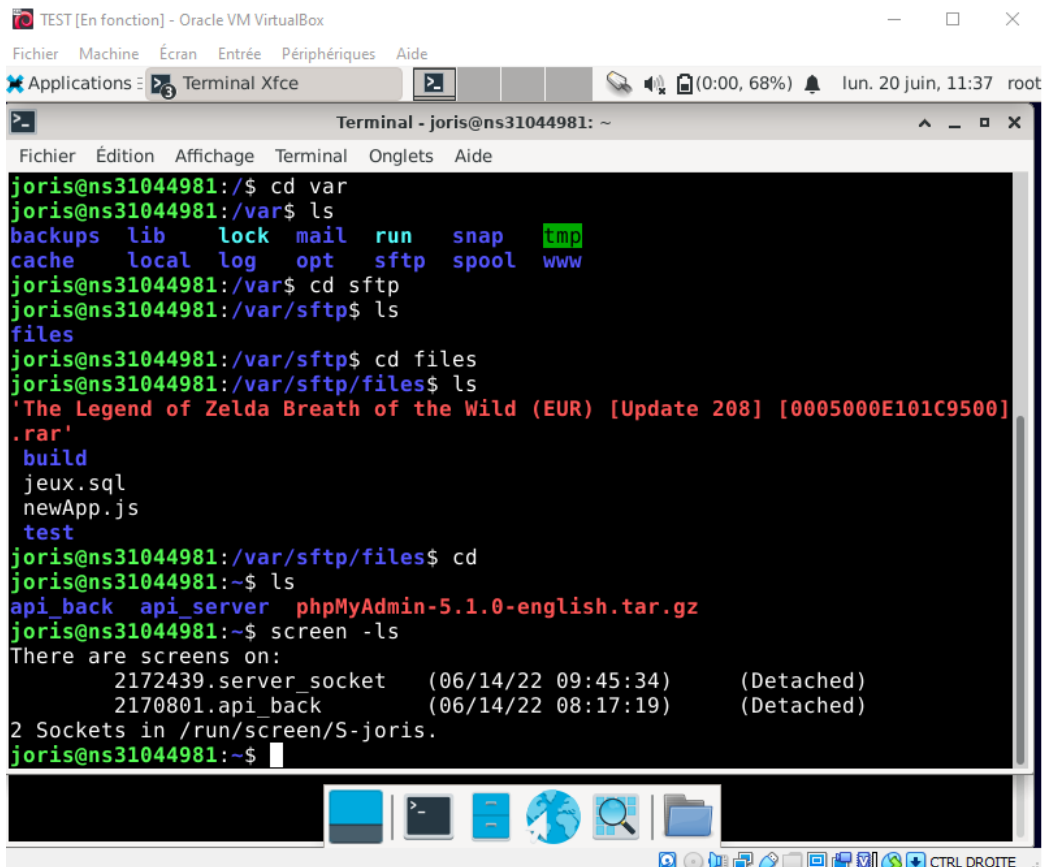
Dans le cadre du déploiement de notre API et de notre server socket nous avons utilisé un serveur distant. Dans un premier temps nous avons créé une **Virtual Machine** (VM) pour accéder au terminal **SSH** de notre server. Etant sur Window, nous ne possédons pas de terminal SSH, nous aurions pu en installer 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.



```
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:~$
```

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

2.7. Sécurité

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. 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 > ...
1  import { IsNotEmpty, IsNumber, IsString } from 'class-validator';
2  import { ApiProperty, PartialType } from '@nestjs/swagger';
3  import { Type } from 'class-transformer';
4  import { GetJeuxDto } from './get-jeux.dto';
5
6  export class CreateJeuxDto extends PartialType(GetJeuxDto) {
7
8      @IsNotEmpty()
9      @ApiProperty()
10     @Type(() => String)
11     nom: string;
12
13     @IsNotEmpty()
14     @ApiProperty()
15     @Type(() => Number)
16     idtype: number;
17 }
18
```

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

Une autre faille à laquelle nous sommes protégé est 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 façon 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 à partir ici du payload.

Grâce au Token, on peut mettre facilement en place un système de rôles, empêchant les hackers de pouvoir cibler facilement un admin possédant tous les droits sur les api par exemple. On peut également grandement limiter l'accessibilité des données de tous les autres utilisateurs. Mais un passage obligatoire pour contrer cette faille est de mettre des demandes de confirmation lors de suppression par exemple pour limiter au maximum l'erreur humaine.

2.8 – Test unitaire

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 fonctionnalités avec plusieurs types de données et plusieurs cas de figure afin de pouvoir s'assurer que notre application est bien sécurisé 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.

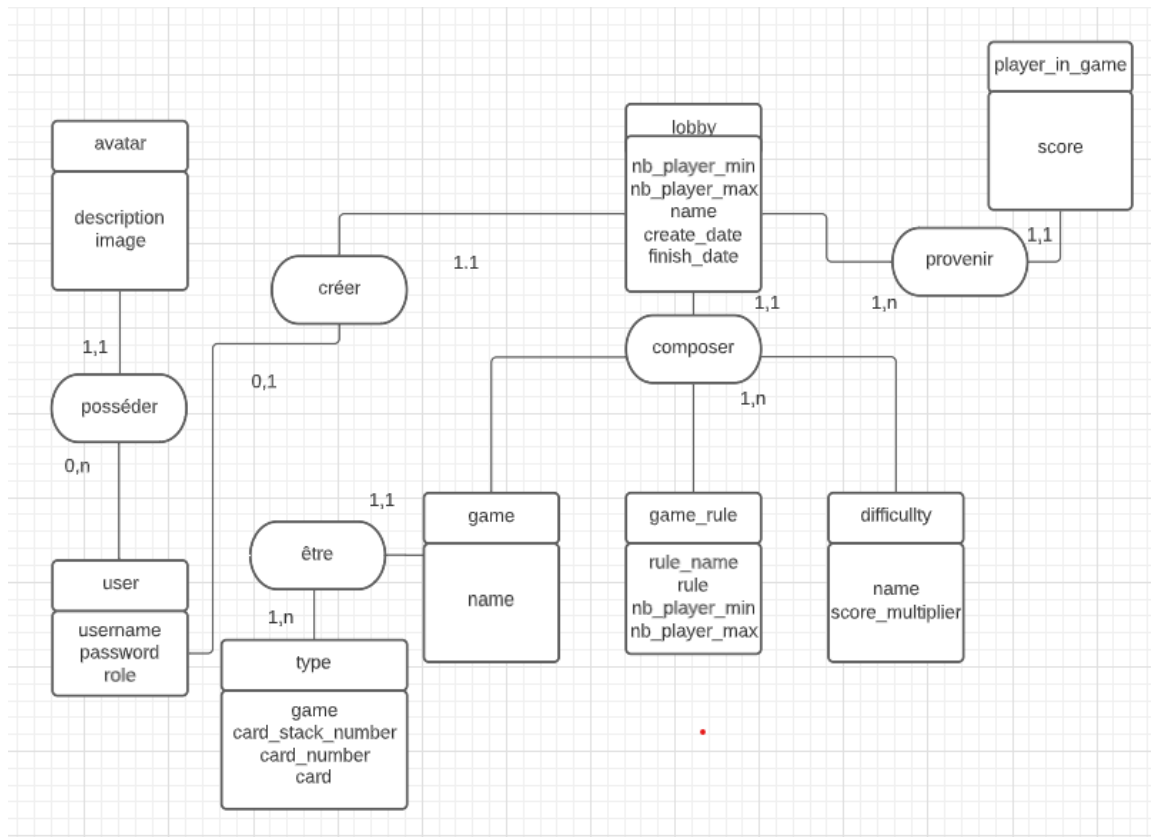
Watch Usage: Press w to show more.
```

Tout cela a été possible 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, l'utilisation de Jest permet la création de fausses données pour vérifier que la fonction que nous testons 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.

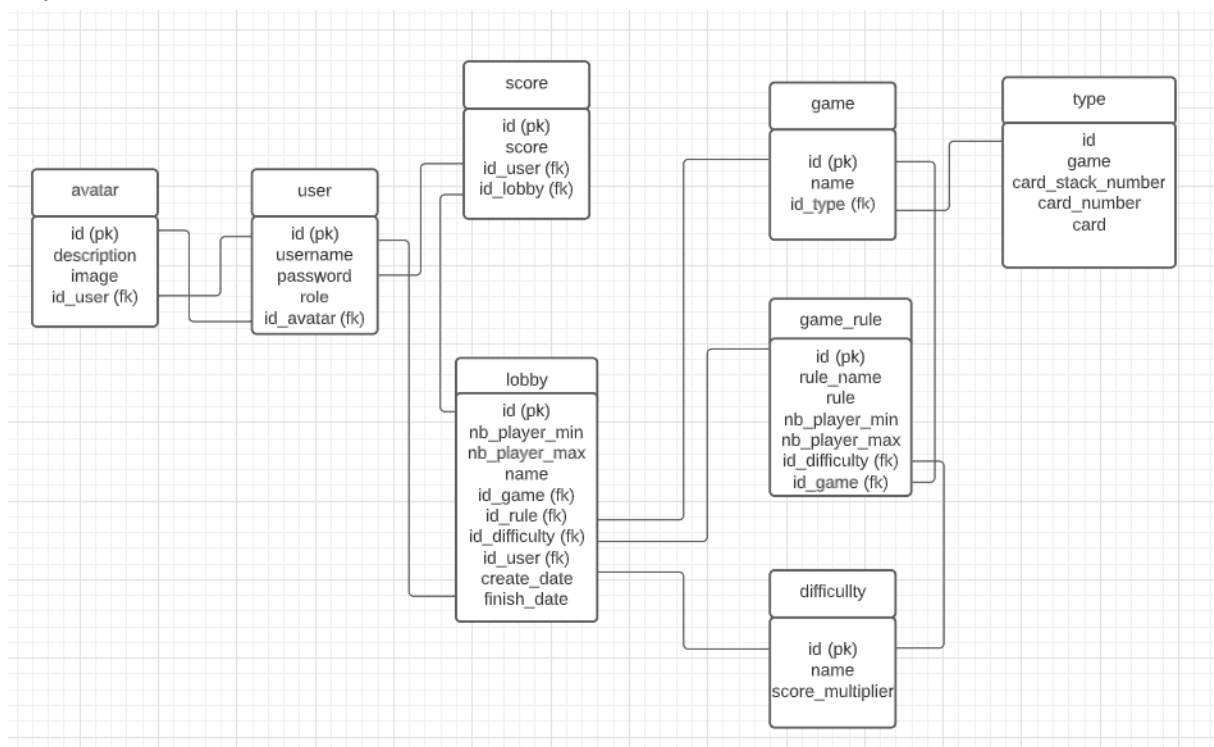
3. - Annexe

3.1 – BDD

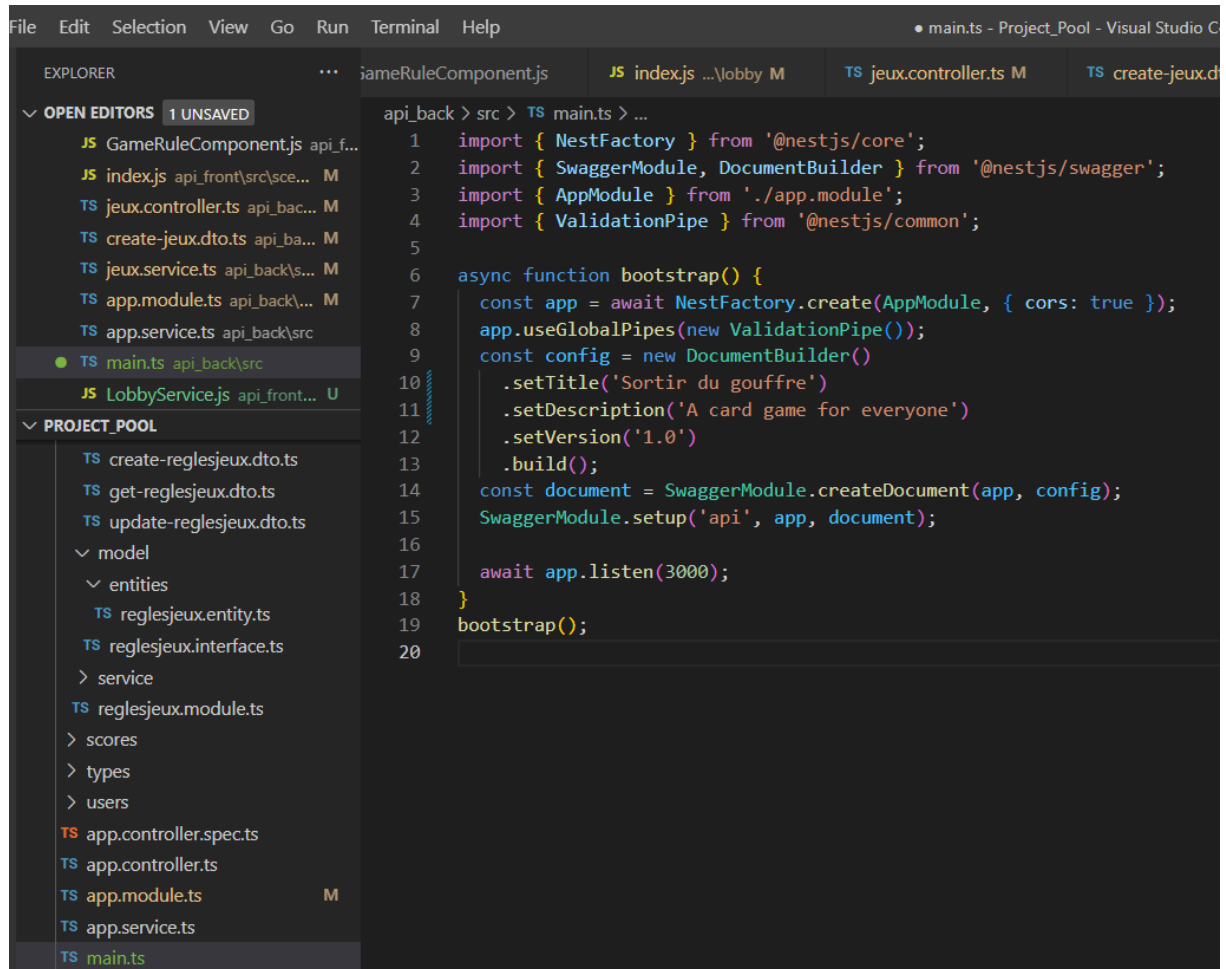
mcd



mld

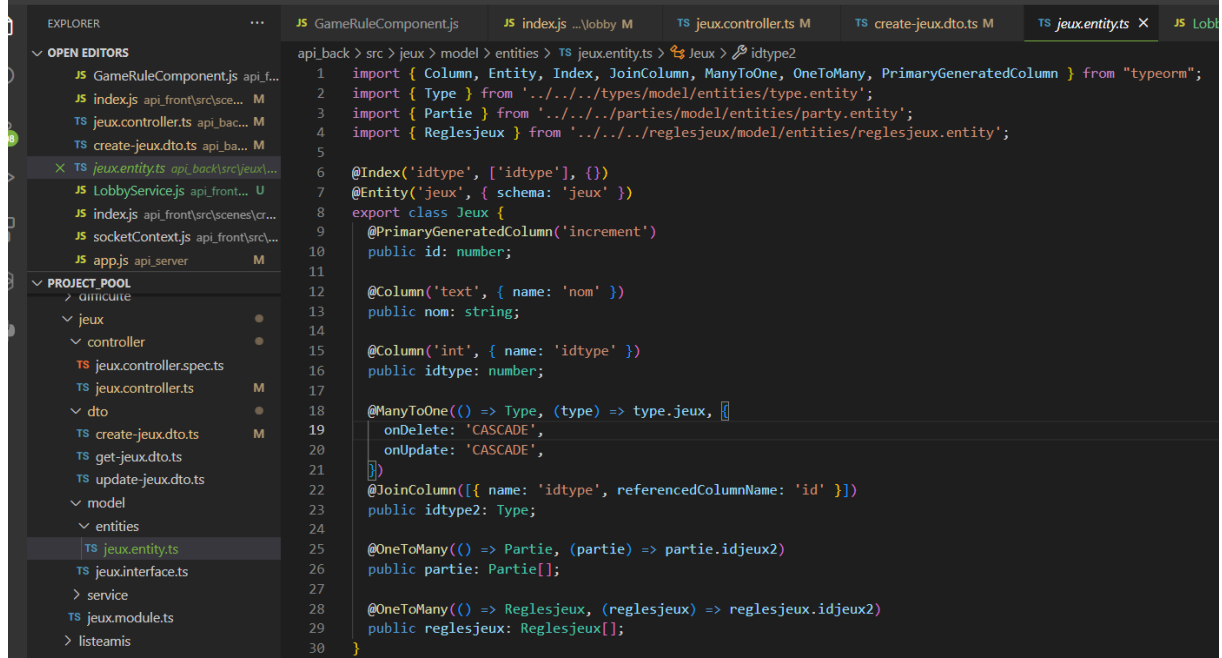
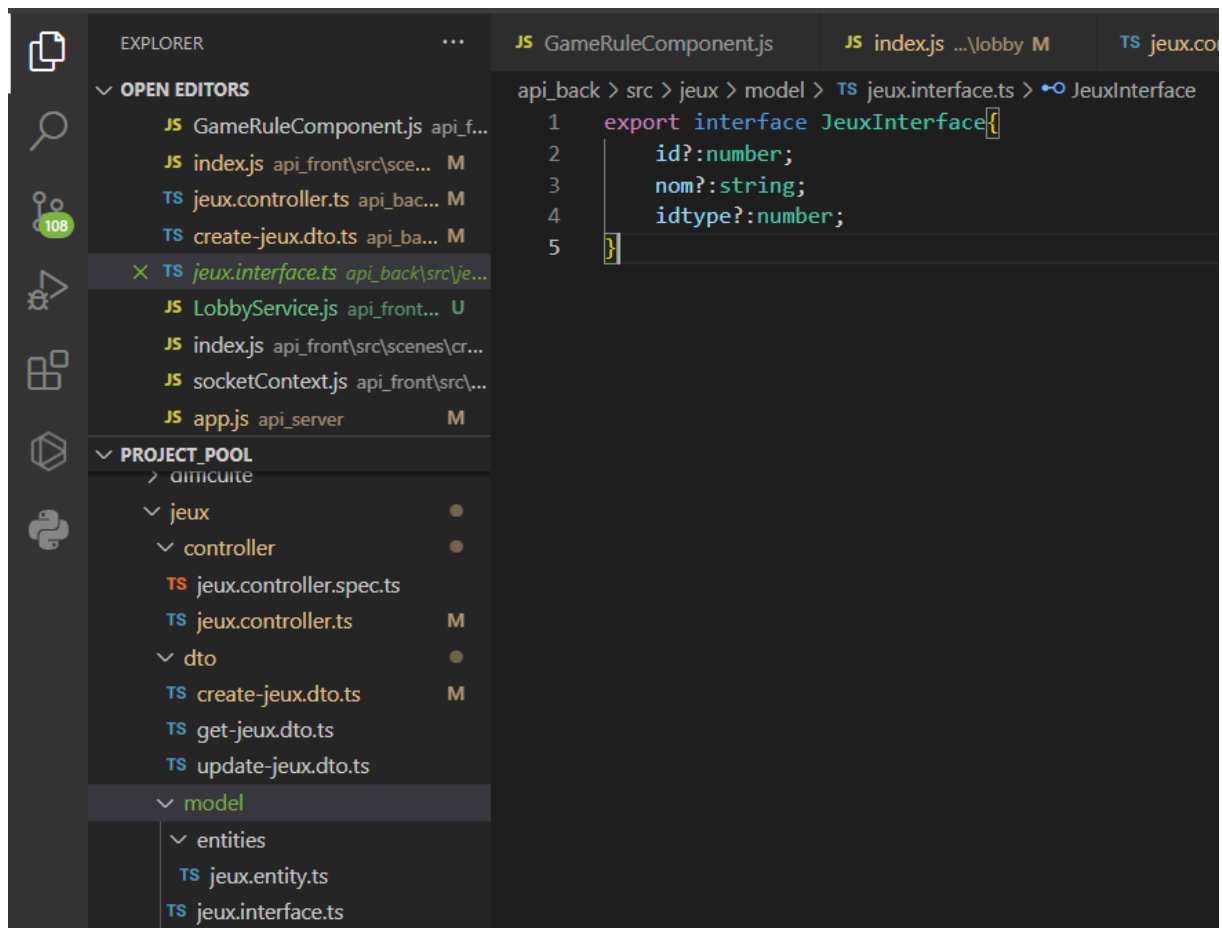


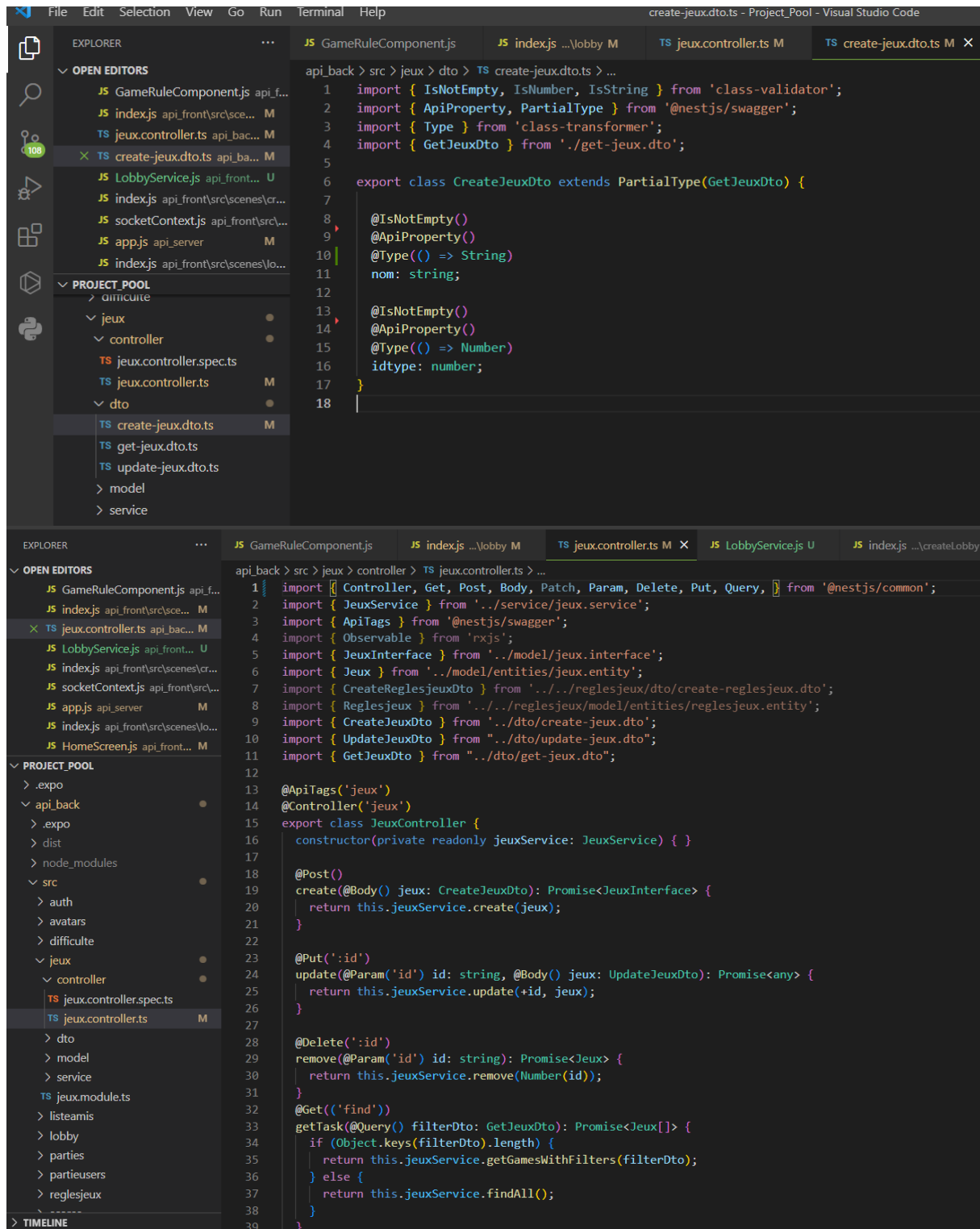
3.2 – Les composants d'accès aux données



The screenshot shows the Visual Studio Code interface with a project named 'Project_Pool'. The Explorer sidebar on the left displays the file structure, including 'OPEN EDITORS' and 'PROJECT_POOL'. The main editor area shows the 'main.ts' file with the following code:

```
api_back > src > TS main.ts > ...
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
```





```
GameRuleComponent.js  index.js ...lobby M  TS jeux.controller.ts M  TS create-jeux.dto.ts M  TS jeux.service.ts X  JS Lobby

OPEN EDITORS
api_back > src > jeux > service > TS jeux.service.ts > JeuxService > remove

JS GameRuleComponent.js api_f... 18
JS index.js api_front/src/sce... M 19
TS jeux.controller.ts api_bac... M 20
TS create-jeux.dto.ts api_ba... M 21
X TS jeux.service.ts api_back/src/jeu... 22
JS LobbyService.js api_front... U 23
JS index.js api_front/src/scenes/cr... 24
JS socketContext.js api_front/src/v... 25
JS apps api_server M 26
PROJECT_POOL
> dist 27
> node_modules 28
> src 29
> auth 30
> avatars 31
> difficulte 32
> jeux 33
  > controller 34
    TS jeux.controller.spec.ts 35
    TS jeux.controller.ts 36
  > dto 37
    TS create-jeux.dto.ts M 38
    TS get-jeux.dto.ts 39
    TS update-jeux.dto.ts 40
  > model 41
    > entities 42
      TS jeux.entity.ts 43
      TS jeux.interface.ts 44
    > service 45
      TS jeux.service.spec.ts 46
      TS jeux.service.ts 47
      TS jeux.module.ts 48
  > timeline 49
  > outline 50
  51
  52
  53
  54
  55
  56
  57

EXPLORER
api_back > src > TS app.modules > AppModule
JS GameRuleComponent.js api_f... 28
JS index.js api_front/src/sce... M 29
TS jeux.controller.ts api_bac... M 30
TS create-jeux.dto.ts api_ba... M 31
TS jeux.service.ts api_back/s... M 32
X TS app.modules.ts api_back/src 33
JS LobbyService.js api_front... U 34
JS index.js api_front/src/scenes/cr... 35
JS socketContext.js api_front/src/v... 36
PROJECT_POOL
TS create-reglesjeux.dto.ts 37
TS get-reglesjeux.dto.ts 38
TS update-reglesjeux.dto.ts 39
> model 40
  > entities 41
    TS reglesjeux.entity.ts 42
    TS reglesjeux.interface.ts 43
  > service 44
    TS reglesjeux.module.ts 45
  > scores 46
  > types 47
  > users 48
  TS app.controller.spec.ts 49
  TS app.controller.ts 50
  TS app.modules.ts 51
  TS app.service.ts 52
  TS main.ts 53
  .eslintrc.js 54
  .gitignore 55
  .prettierrc 56
  jeux.sql M 57
  nest-cli.json 58

TS app.modules > AppModule
import { AuthModule } from './auth/auth.module';
import { AuthService } from './auth/auth.service';
import { LobbyModule } from './lobby/lobby.module';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: '',
      database: 'jeux',
      entities: [
        User,
        Avatar,
        Difficulte,
        Jeux,
        Listeamis,
        Partie,
        Partieuser,
        Reglesjeux,
        Score,
        Type,
      ],
      synchronize: true,
      autoLoadEntities: true,
    }),
    AuthModule, UsersModule, AvatarsModule, DifficulteModule, JeuxModule, ListeamisModule,
    PartiesModule, PartieusersModule, ReglesjeuxModule, ScoresModule, TypesModule, LobbyModule,
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {
  constructor(private connection: Connection) {}
}
```

3.3 – Préparer et exécuter le déploiement d'une application

