



Game Night

Christophe Midelet

Formation à la Wild Code School - Promotion mars 2022

Titre professionnel de développeur web et web mobile

Sommaire

1. Compétences couvertes par le projet
 - a. Développer la partie back-end d'une application web et mobile
 - b. Développer la partie front-end d'une application web et mobile
2. Résumé du projet / Abstract
 - a. Contexte
 - b. Projet
3. Cahier des charges
 - a. Fonctionnalités visiteur
 - b. Fonctionnalités utilisateur
 - c. Fonctionnalités administrateur
4. Spécifications Techniques
 - a. Les compétences
 - b. Les outils
5. Réalisation du projet
 - a. Base de Donnée
 - b. Développement des composants d'accès aux données
 - Les modèles ou Manager
 - Les contrôleurs
 - Les routes
 - c. Développement des composants d'affichage des données.
 - Maquettage de du projet
 - Les « pages »
 - Les composants
6. Présentation d'un jeu d'essai de la fonctionnalité la plus représentative
7. Vielle technologique
8. Description d'une situation de travail ayant nécessité une recherche sur un site anglophone
9. Extrait du site de Board Game Atlas API

1. Compétences couvertes par le projet

a. Développer la partie back-end d'une application web et mobile

- Conception et réalisation de la base de données (BDD)
- Création des modèles qui contiennent les données ainsi que de la logique en rapport avec les données : validation, lecture et enregistrement
- Développement des Controller permettant de traiter les actions de l'utilisateur, et de modifier les données des modèles
- Réalisations des routes nécessaires à l'accès aux Controller.
- Gérer la sécurité par le biais de cookie et de argon2 (un moyen de cryptage des informations)

b. Développer la partie front-end d'une application web et mobile

- Maquetter le projet afin de déterminer le rendu visuel du site visé
- Réaliser une interface web dynamique et adaptable
- Utilisation d'une API extérieure avec traitement et affichage des données
- Gérer la sécurité des données sensibles (mot de passe) avec YUP
- Lier le back-end et le front-end pour que le site soit fonctionnel.

2. Résumé du projet

a. Contexte

Moi-même possesseur de plusieurs centaines de jeux de société, je me retrouve souvent à chercher quel jeu sortir. N'ayant jamais réussi à trouver un site ou une application suffisamment satisfaisante d'utilisation, je me suis donc décidé à créer le mien.

b. Projet

Ce projet a pour but de créer une ludothèque personnelle alimentant par ce biais une base de données de jeux de société avec la possibilité d'ajouter ces propres photos « in-Game » et de les rendre disponibles pour les autres utilisateurs.

L'objectif réel derrière ce projet est donc de permettre aux personnes possédant d'importantes ludothèques de ne pas passer des heures à fixer leurs étagères...

ABSTRACT

Context

Owner of several hundred board-games, I often find myself looking for which game to release. Having never managed to find a site or an application satisfactory enough to use, I therefore decided to create my own.

Project

The aim of this project is to create a personal toy library that feeds a board game database with the possibility of adding these own "in-game" photos and making them available to other users.

The real objective behind this project is therefore to allow people with large toy libraries not to spend hours fixing their shelves...

3. Cahier des charges

a. Fonctionnalités visiteur

- Un visiteur non enregistré doit pouvoir avoir accès à l'accueil du site qui affiche un jeu au hasard provenant de "Board Game Atlas", une API extérieure.
- Un visiteur non enregistré doit pouvoir accéder à la page de création de compte en cliquant sur "register" dans la barre de navigation.
- Un visiteur non enregistré doit pouvoir avoir accès à la page de recherche de jeux en cliquant sur "Search" dans la barre de navigation et ainsi recherché un jeu dans la BDD du site (alimentée par les utilisateurs enregistrés).
- Un visiteur non enregistré doit pouvoir accéder à l'affichage détaillé de chaque jeu disponible dans la BDD du site.
- Un visiteur non enregistré pourra accéder à la page de connexion en cliquant sur "Login" dans la barre de navigation mais ne sera pas en mesure d'aller plus loin sans s'être inscrit au préalable.

b. Fonctionnalités utilisateur

- Un utilisateur enregistré doit avoir les mêmes accès que ceux cités pour un visiteur non enregistré.
- Un utilisateur enregistré doit pouvoir se connecter en utilisant le module "Login" prévu à cet effet.
- Un utilisateur enregistré doit pouvoir accéder au détail d'un jeu et l'ajouter, ou le retirer de sa collection.
- Un utilisateur enregistré doit pouvoir ajouter un jeu dans sa collection et en même temps dans la BDD du site en renseignant lui-même les champs nécessaires à la création d'une nouvelle entrée.
- Un utilisateur enregistré doit pouvoir avoir accès à une page privée listant les jeux qu'il a ajouté dans sa collection.
- Un utilisateur doit pouvoir ajouter des photos supplémentaires pour chaque jeu possédé. (Feature à venir)
- Un utilisateur enregistré doit pouvoir accéder à ces informations dans une page "My account" et les modifier. (Feature à venir)

c. Fonctionnalités administrateur

- Un utilisateur ayant le statut d'administrateur ou Admin doit avoir les mêmes accès que cités précédemment.
- L'admin doit pouvoir avoir accès à la liste des utilisateurs et les supprimer si nécessaire.
- L'Admin doit pouvoir supprimer ou modifier n'importe quel élément (hors email et mot de passe pour la modification) de la BDD. (Feature à venir)

4. Spécifications techniques

a. Les Compétences

- Maquetter un projet
- Réaliser une interface utilisateur web
- Développer une interface web dynamique
- Créer une base donnée
- Développer des composants d'accès aux données
- Développer la partie back-end d'une application web.

b. Les outils

- Langages : HTML 5, CSS3, Javascript,
- Framework : Vite js
- Base de données : MySQL2
- API extérieure : fetch
- API interne : Axios
- Package front end : Prop-types, react, react-dom, react-router, react-router-dom
- Package front-end sécurité : yup
- Package back-end : cors, dotenv, express, Multer
- Package back-end sécurité : argon2, cookie-parser, joi, jsonwebtoken

5. Réalisation du projet

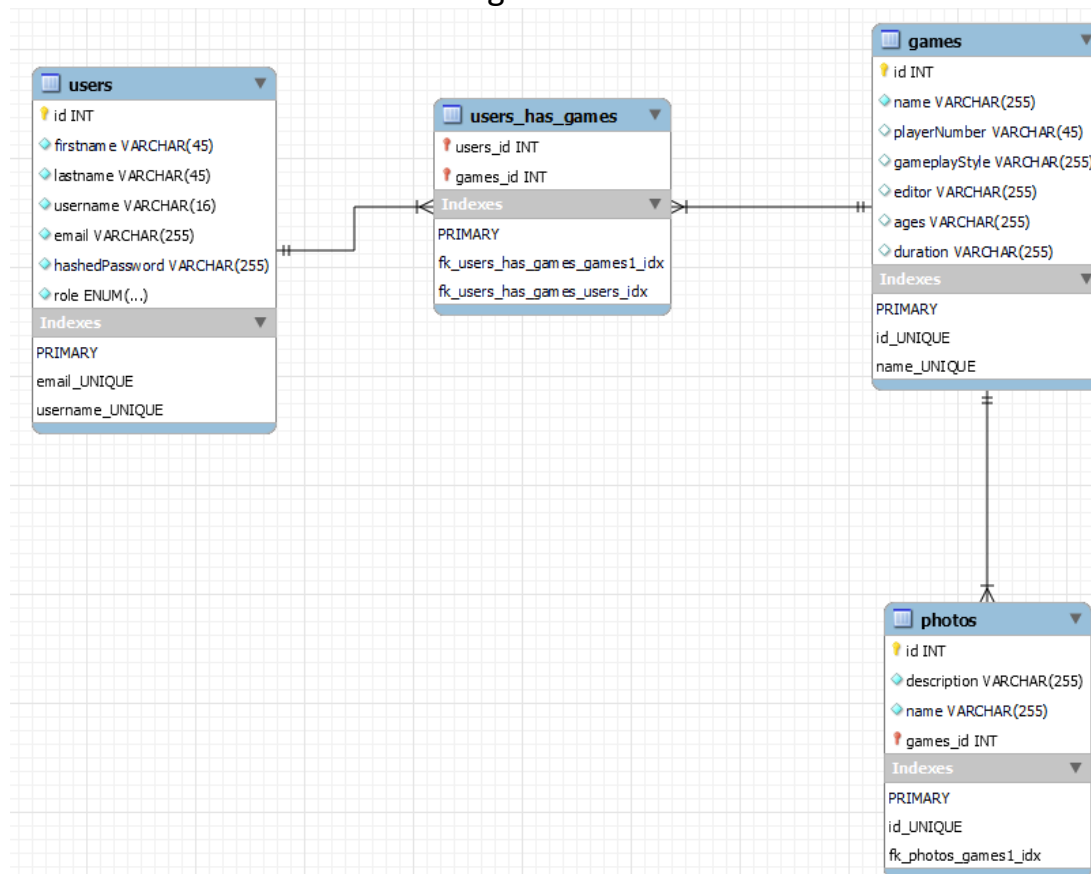
a. Base de données

Pour la réalisation de ce projet, il a fallu mettre en place une Base de données (BDD).

Cette BDD a pour nom "gamenight"

Afin de pouvoir gérer et enregistrer les informations nécessaires au bon fonctionnement du site, il a été décidé de créer 4 tables de données :

- ♦ "users" : permet de gérer les informations relatives aux utilisateurs ('firstname', 'lastname', 'username', 'email', 'hashedPasword', 'role')
- ♦ "games" : permet de gérer les informations relatives aux jeux de société ('name', 'playerNumber', 'gameplayStyle', 'editor', 'ages', 'duration')
- ♦ "photos" : permet de gérer les informations relatives aux photos de jeux de société ('description', 'name', 'games_id')
- ♦ "users_has_games" : permet de gérer quel utilisateur possède quel jeu ('users_id', 'games_id'). Cette table est une table de jointure liant la table "users" et la table "games".



Sur le schéma ci-avant, on constate qu’il existe des liaisons entre les différentes tables de données.

Comme dit précédemment, la table “users_has_games” est une table de jointure. Elle permet à un utilisateur de déclarer qu’il “possède” un ou plusieurs jeux de société, et ainsi chaque utilisateur pourra avoir sa collection personnelle.

La table “photos” est liée à la table “games” et permet ainsi d’ajouter plusieurs photos pour chaque jeu de société.

Exemple de Données se trouvant dans les tables :

◇ “users” :

	* id int	* firstname varchar(255)	* lastname varchar(255)	* username varchar(255)	* email varchar(255)	* hashedPassword varchar(255)	role enum(5)
1	1	christophe	midelet	bdf	bdf007@gmail.com	\$argon2id\$v=19\$m=4096,t=3,p=	ADMIN
2	3	ingrid	dudin	ingrid	ingrid@dudin.fr	\$argon2id\$v=19\$m=4096,t=3,p=	USER
3	9	cdhe	cdsklf	bdfuser	bdf@gmail.com	\$argon2id\$v=19\$m=4096,t=3,p=	USER

◇ “games” :

	* id int	* name varchar(255)	* playerNumber varchar(32)	gameplayStyle varchar(255)	editor varchar(255)	ages varchar(255)	duration varchar(255)
1		Monopoly	2 to 6	Auction	hasbro	from 8 years old	30min to 1h
2		Risk	2 to 6	wargame	hasbro	from 10 years old	1 to 2h
45		qui est-ce?	2	deduction	hasbro	from 6	15min
48		le dernier radot du titan	2 to 5	auuction	paille edition	form 10	60min
49		quinque	2	modualble	Mind fitness game	from 7 to 99	15min
50		Bordel temporel - le visi	1 to 9	cardsGame	paille edition	form 10	from 20 to 60min
51		6 qui prends	2 to 10	cardsgame	gigamic	from 10	45min
52		Fussoir un jeu à se rire d	3 to 10	cardsgame imp	original cup	from 16	from 30min
53		le jeu des séries TV	2 to 6	quizz	hugo image	from 2 to 6	45min
54		red hot silly dragon	2 to 5	auuction	tilsit studio	from 10	20min to 40min
55		Scroll	2 to 4	deckbuilding	epilogik	from 13	30 to 60min
57		el maestro	3 to 8	deduction	tiki edition	from 8	30min

◇ “photos” :

* id int	* description varchar(255)	games_id int	* name varchar(255)
15	"le derniercanotdutitan"	48	le derniercanot
16	"quinque.jpg"	49	quinque.jpg
17	"bordelTemporel.jpg"	50	bordelTemporel
18	"6quiprend.jpg"	51	6quiprend.jpg
19	"fussoir.jpg"	52	fussoir.jpg
20	"lejeudesseriectv.jpg"	53	lejeudesseriectv
21	"redhotsillydragon.jpg"	54	redhotsillydragon
22	"scroll.jpg"	55	scroll.jpg
24	"elMaestro.jpg"	57	elMaestro.jpg
25	"fistthemall.jpg"	58	fistthemall.jpg
26	"sillage.jpg"	59	sillage.jpg
28	"tshack.jpg"	61	tshack.jpg

- “usershasgames”

* users_id int	* games_id int
1	1
3	1
9	1
1	45
3	45
1	48
1	49
3	49
1	50
3	50
1	51
3	51

b. Développement des composants d'accès aux données

Afin d'accéder aux données, il est nécessaire de créer des composants, qui formeront le Back-end du site.

Ces composants peuvent se scinder en trois types principaux :

- Les modèles ou Manager :

Ce type de composant contient les requêtes SQL permettant de communiquer et modifier la BDD.

Il s'agit donc de la première couche d'interaction.

Dans notre cas, il y a 5 modèles :

- « ***AbstractManager.js*** » : contient les requêtes les plus simples, potentiellement utilisables par n'importe quel Controller.

```

backend > src > models > JS AbstractManager.js > AbstractManager
1  class AbstractManager {
2    constructor(connection, table) {
3      this.connection = connection;
4      this.table = table;
5    }
6
7    find(id) {
8      return this.connection.query(`select * from ${this.table} where id = ?`, [
9        id,
10       ]);
11    }
12
13    findAll() {
14      return this.connection.query(`select * from ${this.table}`);
15    }
16
17    delete(id) {
18      return this.connection.query(`delete * from ${this.table} where id = ?`, [
19        id,
20       ]);
21    }
22  }
23
24  module.exports = AbstractManager;
25

```

- « **UserManager.js** » : contient les requêtes permettant de gérer les informations relatives aux utilisateurs (email, mot de passe, prénom, nom, ...)
- « **GameManager.js** » : contient les requêtes permettant de gérer les informations concernant les jeux de société de notre BDD (nom, nombre de joueur, éditeur, style de jeu, tranche d'âge, ...)
- « **PhotoManager.js** » : permet de gérer les informations liées aux photos (nom, description) mais également de gérer le lien entre la table « photos » et la table « games », grâce à la valeur de « games_id » qui est lié à l'identificateur unique d'un jeu de société de la table « games ».
- « **UserhagameManager.js** » : ce dernier composant sert à faire le lien entre les tables « users » et « games ».

- Les Controller :

Ce type de composant est responsable de toute la logique C'est ici que l'on trouvera la plupart des algorithmes, calculs et autres manipulations de données, ainsi que les moyens de convertir les données pour quelles soient utilisables par la BDD d'un côté et compréhensible par l'utilisateur de l'autre côté

Pour ce projet, il a été créé 4 Controller :

- « **UserController.js** » : ce Controller permet de manipuler les données liées aux utilisateurs (« edit », « update », « delete », « register ») et de les renvoyer dans un format compréhensible pour être afficher du côté du client (« read », « browse »). Il permet également de gérer les autorisations liées aux cookies et au token JWT
- « **GameController.js** » : ce composant va gérer les demandes et le traitement des données liées aux jeux de société.
- « **PhotosController.js** » : dans ce composant, comme dans les précédents, les informations liées aux photos vont être traitées et manipulées. De plus, il y a également la gestion « physique » du fichier photo, par le biais de Multer (un package de gestion de fichiers) qui va permettre de créer une copie unique d'un fichier, par le biais d'un timestamp) et l'enregistrer au niveau du serveur afin qu'il soit accessible ensuite par tous les utilisateurs.
- « **UserhasgameController.js** » : Comme son nom l'indique, ce Controller va gérer le traitement des données liant la table « users » et la table « games ».

- Les Routes :

Ces composants permettent de déterminer quelles actions doivent être réalisée et dans quel ordre, afin de répondre aux demandes de l'utilisateur. Comme par exemple, si un utilisateur souhaite se connecter, il va rentrer ses informations qui seront ensuite redirigées vers le bon Controller par le biais de

la route adéquate. Les routes sont divisées en 5 fichiers pour faciliter la maintenance du code.

- « **index.js** » : Ce fichier est le point d'entrée des routes de notre site. Il contient les redirections vers les différents fichiers.
- « **users.routes.js** » : Ce fichier détermine dans quel sens doivent être effectués les actions liées à un utilisateur. Par exemple, si l'utilisateur souhaite se déconnecter, la route montre qu'il y a deux étapes à respecter, tout d'abord vérifier qu'il est effectivement connecté et qu'il a les autorisations nécessaires et ensuite, d'effectuer la déconnexion en supprimant le cookie maintenant la connexion.
- « **games.routes.js** » : permet de gérer l'ordre des actions liées aux jeux de société de la BDD. Par exemple, si on souhaite ajouter un jeu de société, la route nous indique que les données vont d'abord être enregistrées dans la base de données, puis les informations modifiées de la table « games » seront récupérées pour être renvoyées vers l'utilisateur.
- « **photos.routes.js** » : ce composant agit de la même façon que celui concernant les jeux de société, excepté qu'il concerne les informations liées aux photos.
- « **userhasgames.routes.js** » : ce composant agit de la même façon que « **games.routes.js** » en y ajoutant le traitement de la table « usershasgames »

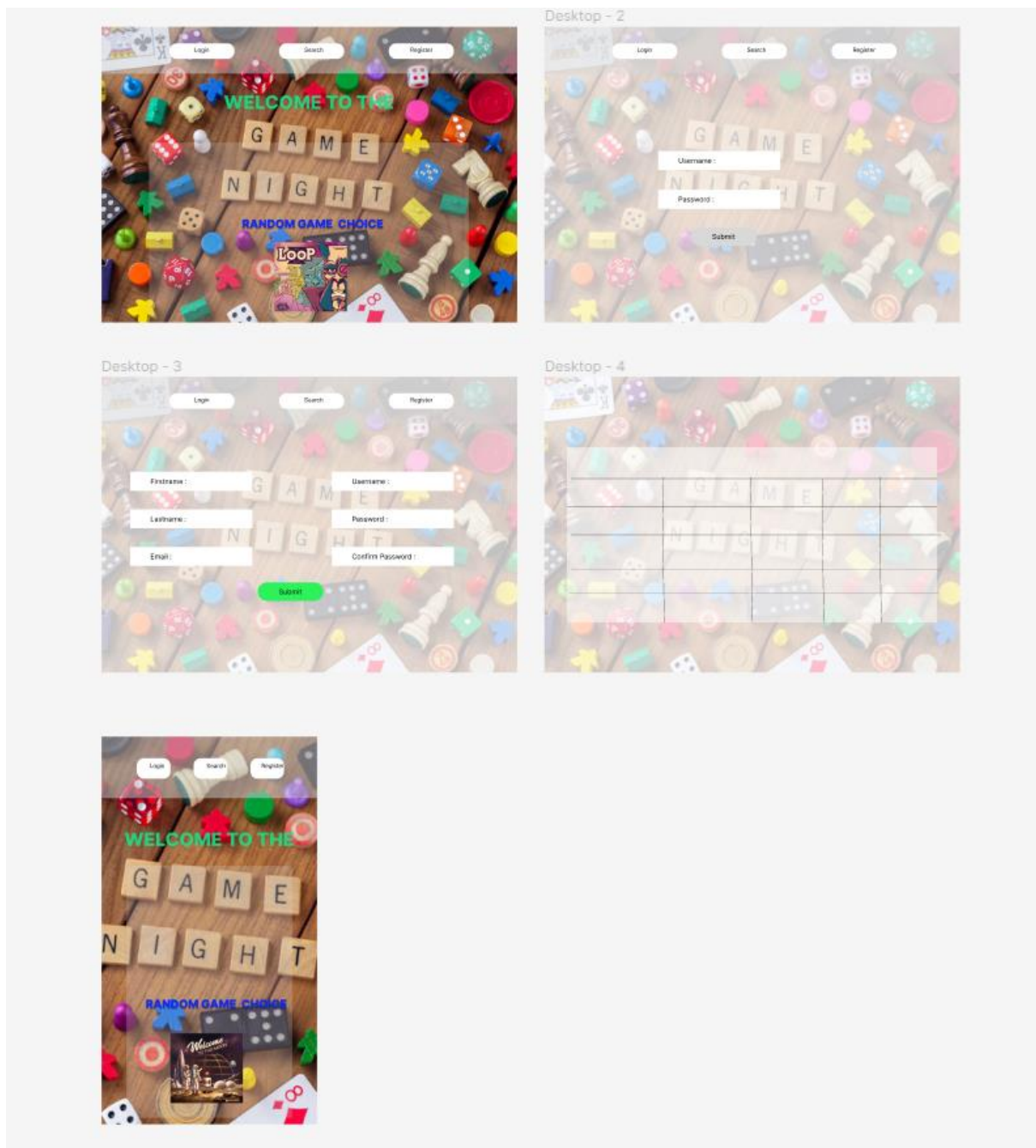
c. Développement des composants d'affichage des données.

Afin d'afficher les données, il est nécessaire de créer des composants, qui formeront la partie front-end.

▪ Maquettage du projet :

Pour réaliser cette partie, il est nécessaire de maquetter le projet afin de déterminer le visuel visé.

La maquette ci-après a été réalisée à l'aide de Figma.com.



La partie visible par l'utilisateur, ou front-end, a été développée selon l'arborescence suivante :

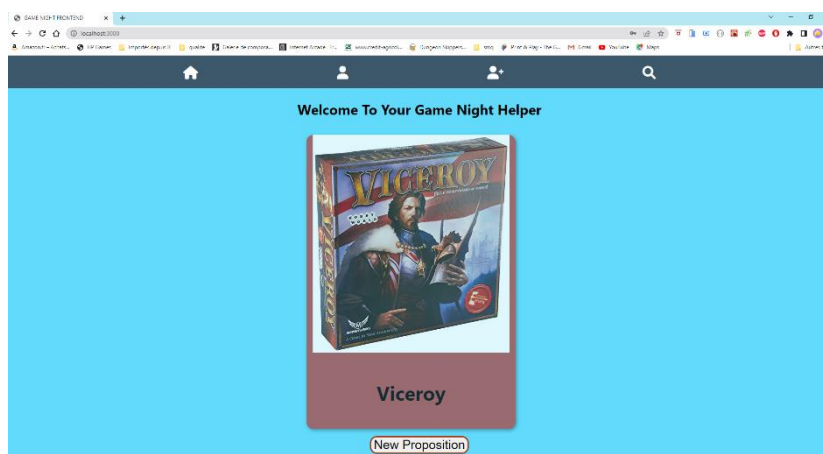
- « ***index.html*** » : ce fichier est l'équivalent d'un tableau blanc sur lequel on viendra afficher les informations selon les demandes de l'utilisateur et qui seront gérées par un script.
- « ***main.jsx*** » : ce fichier est le véritable point d'entrée de notre application. Ce composant permet de gérer les

informations utilisateur afin qu'elles soient accessibles sur les différentes « pages » du site par le biais d'un `UserContext`. Ce script va également permettre de gérer les différentes vues accessibles en utilisant le « `BrowserRouter` » du package « `react-router-dom` ».

- « ***App.jsx*** » : ce composant permet de créer des « routes », qui selon l'adresse destination, affichera tel ou tel composant et créera pour l'utilisateur une vue correspondante à sa demande que l'on pourrait considérer comme étant une « page web ». L'affichage de ces « pages » peut être conditionné par l'utilisation du « `UserContext` » déployé dans « `main.jsx` » qui nous permet de vérifier si un utilisateur est connecté et donc de lui donner accès à des composants réservés aux utilisateurs enregistrés du site. De la même façon on peut conditionner les accès en fonction du rôle de l'utilisateur (« ADMIN », « USER »). Ce composant contient également la barre de navigation qui permet d'aller d'une « page » à l'autre.
- Les « pages » :

Afin de faciliter la navigation, le site a été découpé en différentes « pages » qui regroupent chacune les composants nécessaires à la fonction qui lui est attribué.

- « ***Home.jsx*** » : Ce composant permet d'afficher un message d'accueil et un composant « `Appel.jsx` » que l'on évoquera plus tard.



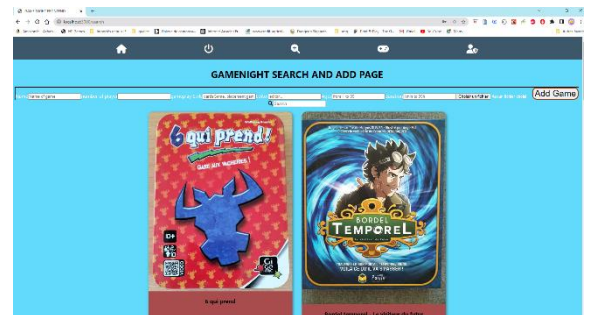
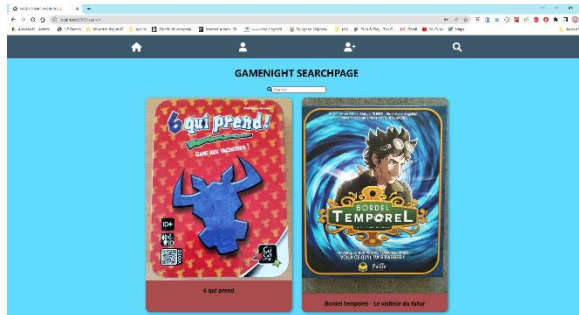
- « **RegisterUser.jsx** » : Ce composant affiche un formulaire permettant de créer un nouvel utilisateur. Il va aussi permettre de vérifier que les informations fournies par l'utilisateur respectent les règles dictées, comme par exemple, la longueur minimale d'un mot de passe, ou l'obligation d'avoir une adresse email respectant un certain format. Ces informations sont ensuite regroupées pour pouvoir être « envoyées » vers le back-end où elles seront de nouveau vérifiées selon des règles décrites dans le Manager.

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/register'. The page has a light blue background and a dark blue header with navigation icons. A white 'Sign Up' form is centered on the page. The form contains the following fields: 'Email:', 'Username:', 'Password:' (with a 'show' icon), 'Confirm Password:', 'Firstname:', and 'Lastname:'. A 'Register' button is located at the bottom right of the form.

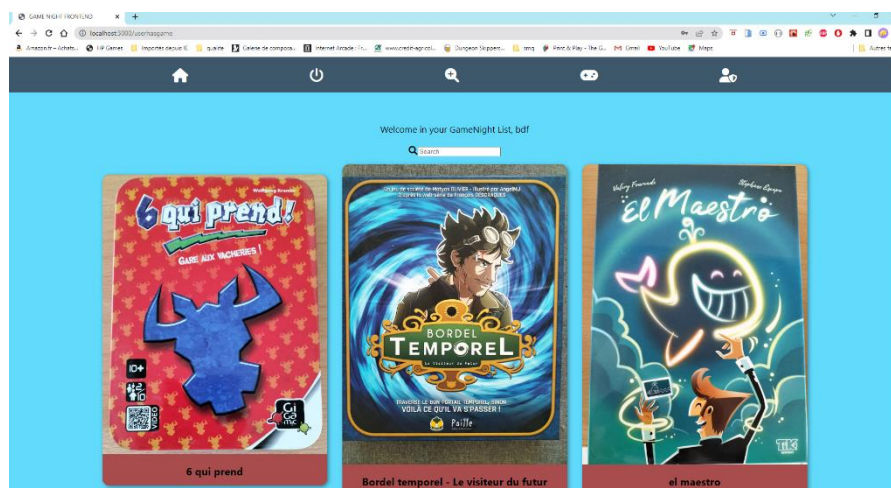
- « **Login.jsx** » : Ce composant permet l'affichage d'un formulaire de connexion.

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/login'. The page has a light blue background and a dark blue header with navigation icons. A white 'Log in' form is centered on the page. The form contains the following fields: 'Email:' (with the value 'test07@gmail.com') and 'Password:' (with masked characters '*****'). A 'LOGIN' button is located at the bottom of the form.

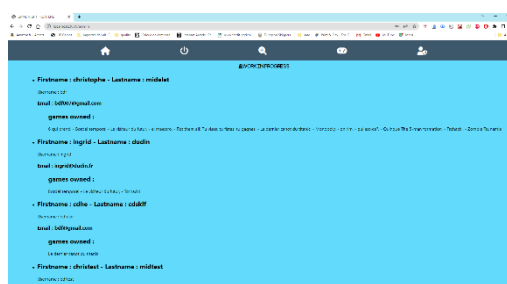
- « **Search.jsx** » : Ce composant varie en affichage selon que l'utilisateur est connecté ou non. Il fait appel à un composant qui permet d'afficher tous les jeux de société présent dans la base de données. Mais également de rechercher dans la base de données.



- « **User.jsx** » : Ce composant n'est accessible qu'aux utilisateurs connectés. Il affiche un composant listant les jeux de société que l'utilisateur a ajouté à sa liste personnelle. Cette page est personnalisée en y affichant le surnom que l'utilisateur s'est choisi.

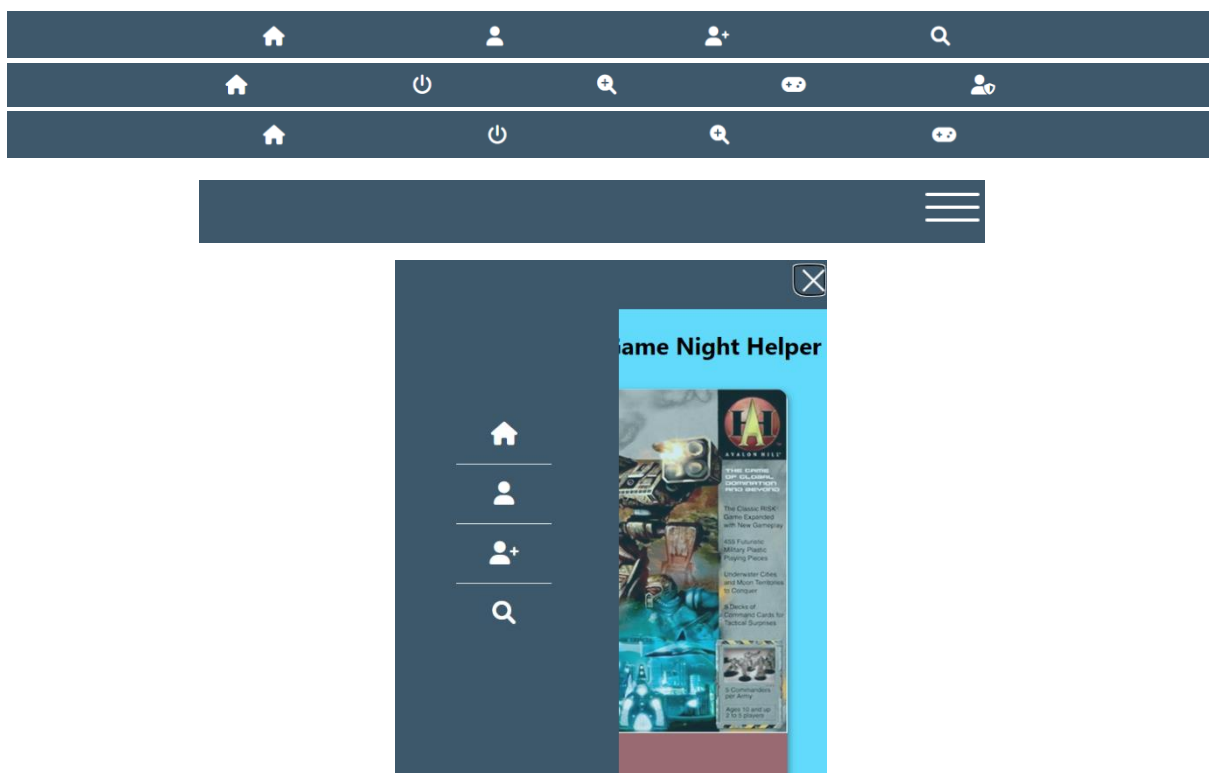


- « **Admin.jsx** » : Ce composant n'est accessible que par les utilisateurs ayant le rôle de « ADMIN ». Il permettra de gérer les utilisateurs. Pour le moment, il affiche la liste des utilisateurs avec leur email et la liste des jeux de société qu'ils ont ajoutés à leur collection personnelle.



Les composants (ou « pages ») précédent font appel à différents composants que nous allons maintenant passer en revue :

- « **navbar.jsx** » : Ce composant permet de faciliter la navigation au sein du site. Il est adaptatif en fonction des situations (connecté ou non, rôle « USER » ou « ADMIN »). Il est également adapté à la l'utilisation sur différentes tailles d'écran. Il passe d'un affichage en bandeau au niveau supérieur du site à un affichage de « menu burger » déroulant.




- « **Appel.jsx** » : Ce composant qui est affiché sur la page « **Home.jsx** » est composé d'un appel à une API extérieure (Board Games Atlas API) qui nous permet de récupérer les informations d'un jeu de société au hasard. Le rafraichissement des données de l'API se fait à chaque chargement de la page, mais on peut également utiliser un bouton qui aura le même effet qu'un rafraichissement sans changer effectivement de « page ». Ces données sont ensuite envoyées dans le composant suivant.

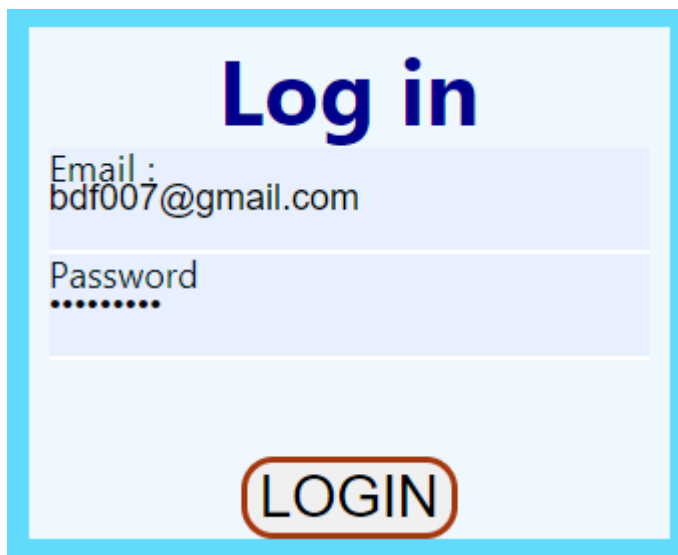
- « **CardGame.jsx** » : Ce composant est utilisé pour permettre l’affichage des données récupérées dans le composant précédent « **Appel.jsx** ». Il va afficher en premier lieu la photo et le nom du jeu de société. Et lorsque l’on passe la souris au-dessus, des informations supplémentaires apparaissent en surimpression. Il y a également un hyperlien permettant d’accéder à la fiche du jeu de société sur un site extérieur lié à l’API utilisée.
- « **GameList.jsx** » : Ce composant comme son nom l’indique permet d’afficher la liste de tous les jeux de société contenus dans la base de données. Chaque jeu de société affiché est cliquable et nous permet d’atteindre un composant (« **GameDetail.jsx** ») affichant les détails du jeu de société choisi dans une autre « *page* », ainsi que la possibilité d’ajouter ou de retirer ce jeu directement à la collection de l’utilisateur connecté (L’« ADMIN » peut également supprimer définitivement de la base données). Il fait également appel à un autre composant « **SearchBar.jsx** » qui lui permet de rechercher et de modifier « à la volée » les jeux affichées selon les données saisies.
- « **AddGame.jsx** » : Ce composant va permettre d’ajouter des jeux de société dans la base de données et ainsi les rendre disponibles pour tous les autres utilisateurs.
- « **AddImage.jsx** » : Ce Composant va permettre de rajouter des photos pour un jeu de société déterminé.

6. Présentation d'un jeu d'essai de la fonctionnalité la plus représentative

Pour cette présentation, nous allons détailler les étapes d'ajout, par un utilisateur enregistré, d'un jeu de société qui n'est pas déjà dans la base de données.

Pour réaliser cette action, un utilisateur va se connecter à son compte en cliquant sur « login » 

Il accède alors à la page de connexion.

A screenshot of a login form. At the top, the text "Log in" is displayed in a large, bold, blue font. Below it, there are two input fields. The first field is labeled "Email :" and contains the text "bdf007@gmail.com". The second field is labeled "Password" and contains a series of dots. At the bottom of the form, there is a button with the text "LOGIN" in a bold, black font, enclosed in a rounded rectangle with a blue border.

Les données saisies sont ensuite postées (si l'email et le mot de passe sont bien présents) vers la route user afin de vérifier les informations fournies.

Les informations envoyées vont donc arriver sur « index.js »

```
backend > src > routes > JS index.js > ...
1  const express = require("express");
2
3  const gameRoutes = require("./games.routes");
4  const userRoutes = require("./users.routes");
5  const photoRoutes = require("./photos.routes");
6  const userhasgameRoutes = require("./userhasgames.routes");
7
8  const router = express.Router();
9
10 router.use("/game", gameRoutes);
11 router.use("/user", userRoutes);
12 router.use("/photo", photoRoutes);
13 router.use("/userhasgame", userhasgameRoutes);
14
15 module.exports = router;
16
```

Elles sont ensuite redirigées vers « user.routes.js »

```
backend > src > routes > JS users.routes.js > ...
1  const express = require("express");
2
3  const { UserController } = require("../controllers");
4
5  const router = express.Router();
6
7  router.get(
8    "/",
9    UserController.authorization,
10   UserController.isAdmin,
11   UserController.browse
12  );
13  router.get("/logout", UserController.authorization, UserController.clearCookie);
14  router.get("/:id", UserController.read);
15  router.put(
16    "/:id",
17    UserController.authorization,
18    UserController.isSameId,
19    UserController.edit
20  );
21  router.post("/register", UserController.register);
22  router.post("/login", UserController.login);
23  router.delete("/:id", UserController.delete);
24
25  module.exports = router;
26
```

Etant donné que l'utilisateur est en cours de connexion, les informations vont continuer leur chemin sur la route router.post(« login », UserController.login)

```
118
119   static login = async (req, res) => {
120     const { email, password } = req.body;
121
122     if (!email || !password) {
123       return res.status(400).send("You must provide an email and a password");
124     }
125     try {
126       const [[user]] = await models.users.findByEmail(email);
127
128       if (!user) {
129         return res.status(403).send("Invalid email or password");
130       }
131       if (await models.users.verifyPassword(password, user.hashPassword)) {
132         // Create token
133         const token = jwt.sign(
134           { id: user.id, role: user.role },
135           process.env.ACCESS_JWT_SECRET,
136           { expiresIn: process.env.ACCESS_JWT_EXPIRESIN }
137         );
138         return res
139           .cookie("accessToken", token, {
140             httpOnly: true,
141             secure: process.env.ACCESS_JWT_SECURE === "true",
142             maxAge: parseInt(process.env.ACCESS_JWT_COOKIE_MAXAGE, 10),
143           })
144           .status(200)
145           .json({
146             id: user.id,
147             email: user.email,
148             username: user.username,
149             role: user.role,
150             firstname: user.firstname,
151           });
152       }
153     }
154   }
```

- 22 -


On constate que les informations vont subir différentes opérations afin de les vérifier et d'autoriser la connexion.


On commence par vérifier que les informations saisies contiennent bien un email et un mot de passe.

Ensuite on récupère les informations de la base de données par le biais de la fonction *findByEmail()*.

```
125     findByEmail(email) {
126         return this.connection.query(
127             `SELECT * FROM ${this.table} WHERE email = ?`,
128             [email]
129         );
130     }
```

Puis le mot de passe va être comparer avec le mot de passe haché et sécurisé. Une fois fait et validé, le composant va créer un token de sécurité qui sera ensuite inséré dans un cookie de connexion.

Le cookie est ensuite renvoyé vers la partie front-end, et permet la connexion à la page « User » représenté par  sur la barre de navigation.

L'utilisateur clique maintenant sur « Search and Add » 

Il a désormais accès au composant « **AddGame.jsx** »

GAMENIGHT SEARCH AND ADD PAGE															
Name	Name of game	number of player	gameplay	Style	cardsGame	placement game	Editor	editor	Ages	from 1 to 99	duration	1min to 99h	Choisir un fichier	Aucun fichier choisi	Add Game
<input type="text" value="Search"/>															

Les informations suivantes doivent être renseignées :

- Nom du jeu de société
- Nombre de joueurs
- Style de jeu
- Editeur
- Ages minimum
- Durée de jeu
- Une photo. (Avec un effet de preview lorsqu'une image est sélectionnée)

Ces 6 premières données vont être regroupées dans un objet appelé « *gameData* » et la photo dans un objet appelé « *photoData* » et qui seront envoyés (**ligne 71**) vers le back-end par le biais du service « *Axios* ».

```
18 > const handleFileChange = (e) => { ...
21   };
22
23 > const uploadPhoto = async (idGame) => { ...
36   };
37
38 > const addUserHasGame = async (idGame) => { ...
50   };
51
52   async function handleSubmit(e) {
53     e.preventDefault();
54 >     if (!file) { ...
56   }
57 >     if (!gamestate) { ...
59   }
60
61 >     const gameData = { ...
68   };
69     try {
70       await axios
71         .post("game", gameData)
72         .then((res) => res.data)
73         .then((data) => {
74           uploadPhoto(data.id);
75           addUserHasGame(data.id);
76           setFileOverview(null);
77           document.getElementById("file").value = null;
78           dispatch({ type: "RESET_FORM" });
79           navigate("/userhasgame");
80           return console.error("Game added successfully");
81         });
82     } catch (err) {
83       if (err?.response?.status === 400) {
84         return console.error("Game already exist");
85       }
86       if (err?.response?.status === 500) {
87         return console.error("Server error");
88       }
89       return console.error(JSON.stringify(err.message));
90     }
91   }
92 }
```

Les informations vont maintenant passées par la route post de « **games** .*routes.js* »

```
backend > src > routes > JS games.routes.js > ...
1   const express = require("express");
2
3   const { GameController } = require("../controllers");
4
5   const router = express.Router();
6
7   router.get("/", GameController.browse);
8   router.get("/:id", GameController.read);
9   router.put("/:id", GameController.edit);
10  router.post("/", GameController.add, GameController.browse);
11  router.delete("/:id", GameController.delete);
12
13  module.exports = router;
14
```


Les données vont passer par le Controller « **GameController.js** » et réaliser la fonction « *add* »

```
54     static add = (req, res) => {
55         const game = req.body;
56
57         // TODO validations (length, format...)
58         models.games
59             .insert(game)
60             .then([result]) => {
61                 res.status(201).send({ ...game, id: result.insertId });
62             })
63             .catch((err) => {
64                 console.error("error add", err);
65                 res.sendStatus(500);
66             });
67     };
```

Ces données vont être insérées dans la base de données par le biais de la requête SQL insert.

```
6     insert(game) {
7         return this.connection.query(
8             `insert into ${GameManager.table} (name, playerNumber, gameplayStyle, editor, ages, duration) values (?, ?, ?, ?, ?, ?)`,
9             [
10                game.name,
11                game.playerNumber,
12                game.gameplayStyle,
13                game.editor,
14                game.ages,
15                game.duration,
16            ]
17         );
18     }
```

Ensuite la route, nous fait réaliser la fonction « *browse* » qui permet de récupérer toutes les informations liées à tous jeux de société.

Ces données sont maintenant renvoyées vers le front-end, ou elles vont être traitées afin de récupérer l'id unique du jeu de société qui vient d'être créé.

Cet Id est associé à la photo ajoutée par l'utilisateur et l'objet nouvellement créé et renvoyé dans le back-end par le biais de la route post de « **photos.routes.js** »

```

backend > src > routes > JS photos.routes.js > ...
1  const express = require("express");
2
3  const { PhotoController } = require("../controllers");
4
5  const router = express.Router();
6
7  router.get("/", PhotoController.browse);
8  router.get("/:id", PhotoController.read);
9  router.put("/:id", PhotoController.edit);
10 router.post("/", PhotoController.uploadImage, PhotoController.postPhoto);
11 router.delete("/:id", PhotoController.delete);
12
13 module.exports = router;
14

```

Afin de poster une photo, on va d'abord passer par la fonction *uploadImage()*

```

193 static uploadImage = (req, res, next) => {
194   postPhotoObject(req, res, (err) => {
195     if (err) {
196       return res.status(400).send(err);
197     }
198     req.imgName = req.file.filename;
199     return next();
200   });
201 };

```

Cette fonction fait appel à *postPhotosObjects()*

```

9  const postPhotoObject = (req, res, next) => {
10    const storage = multer.diskStorage({
11      destination: (_, file, cb) => {
12        cb(null, path.join(__dirname, "../../public/assets/images/"));
13      },
14      filename: (_, file, cb) => {
15        cb(null, `${Date.now()}-${file.originalname}`);
16      },
17    });
18    const upload = multer({ storage }).single("file");
19    upload(req, res, next, (err) => {
20      if (err) {
21        res.status(500).send(err);
22      } else {
23        next();
24      }
25    });
26  };
27

```

Pour sauvegarder la photo de façon physique sur le serveur afin qu'elle soit accessible par tous les autres visiteurs, on utilise la package de gestion de fichier, **Multer**,

Maintenant que le fichier est renommé et sauvegardé, les informations vont être enregistrées dans la base de données par le biais de la fonction *postPhotos()*

```
110 static postPhoto = async (req, res, next) => {
111   const obj = JSON.parse(JSON.stringify(req.body));
112   const photo = {
113     name: req.imgName,
114     description: obj.description,
115     games_id: obj.games_id,
116   };
117
118   try {
119     const object = await models.photos.validate({ ...photo });
120     if (!object) {
121       return res
122         .status(400)
123         .send("One of the required data is missing or incorrect");
124     }
125     const [result] = await models.photos.insert(photo);
126     const [[photoCreated]] = await models.photos.find(result.insertId);
127     res.status(201).send(photoCreated);
128   } catch (err) {
129     res.status(501).send(err.message);
130   }
131   return next();
132 };
133
```

Dans cette fonction, les données reçues sont réarrangées dans un objet qui est ensuite envoyé à une fonction de vérification et de validation du modèle.

Une fois fait, les données sont enregistrées dans la base de données, et renvoyées vers le front-end.

Dans le même temps, l'id unique récupéré précédemment est également associé à l'id unique de l'utilisateur. Ces informations sont envoyées dans le back-end par le biais des la route *post* de « ***userhasgames.routes.js*** »

```

backend > src > routes > JS userhasgames.routes.js > ...
1  const express = require("express");
2
3  const { UserhasgameController } = require("../controllers");
4
5  const router = express.Router();
6
7  router.get("/:id", UserhasgameController.browse);
8  router.get("/user/:id", UserhasgameController.browseNameList);
9  router.get("/detailGame/:id", UserhasgameController.read);
10 router.put("/:id", UserhasgameController.edit);
11 router.post("/", UserhasgameController.add);
12 router.delete("/:id", UserhasgameController.delete);
13
14 module.exports = router;
15

```

Les données sont transmises à la fonction *add()* qui va les insérer dans la base de données avec la requête SQL « insert »

```

29  insert(users_has_games) {
30      return this.connection.query(
31          `INSERT INTO users_has_games (users_id, games_id) VALUES (?, ?)`,
32          [users_has_games.users_id, users_has_games.games_id]
33      );
34  }

```

Une fois cette fonction terminée, les données sauvegardées en locales dans le front-end sont effacées par le biais d'un « gameReducer.js » qui « reset » le formulaire.

```

1  const initialState = {
2      id: null,
3      name: "",
4      playerNumber: "",
5      gameplayStyle: "",
6      editor: "",
7      ages: "",
8      duration: "",
9  };
10
11  const gameReducer = (state, action) => {
12      switch (action.type) {
13          case "ID_GAME":
14              return { ...state, id: action.payload };
15          case "UPDATE_NAME":
16              return { ...state, name: action.payload };
17          case "UPDATE_PLAYERNUMBER":
18              return { ...state, playerNumber: action.payload };
19          case "UPDATE_GAMEPLAYSTYLE":
20              return { ...state, gameplayStyle: action.payload };
21          case "UPDATE_EDITOR":
22              return { ...state, editor: action.payload };
23          case "UPDATE_AGES":
24              return { ...state, ages: action.payload };
25          case "UPDATE_DURATION":
26              return { ...state, duration: action.payload };
27          case "RESET_FORM":
28              return { ...initialState };
29          default:
30              return state;
31      }
32  };

```

La page est ensuite rechargée et affiche la « page » « user » avec le nouveau jeu de société ajouté à la collection de l'utilisateur.

Il est désormais également disponible pour les visiteurs et les autres utilisateurs.

7. Vielle technologique

Recherche sur Jsonwebtoken : <https://www.ionos.fr/digitalguide/sites-internet/developpement-web/json-web-token-jwt/>

Recherche sur les différences entre yup et joi :
<https://stackoverflow.com/questions/66009560/yup-vs-joi-for-frontend-validation>

<https://egghead.io/blog/zod-vs-yup-vs-joi-vs-io-ts-for-creating-runtime-typescript-validation-schemas>

8. Description d'une situation de travail ayant nécessité une recherche sur un site anglophone

Pour ce projet, je souhaitais pouvoir utiliser une API extérieur.

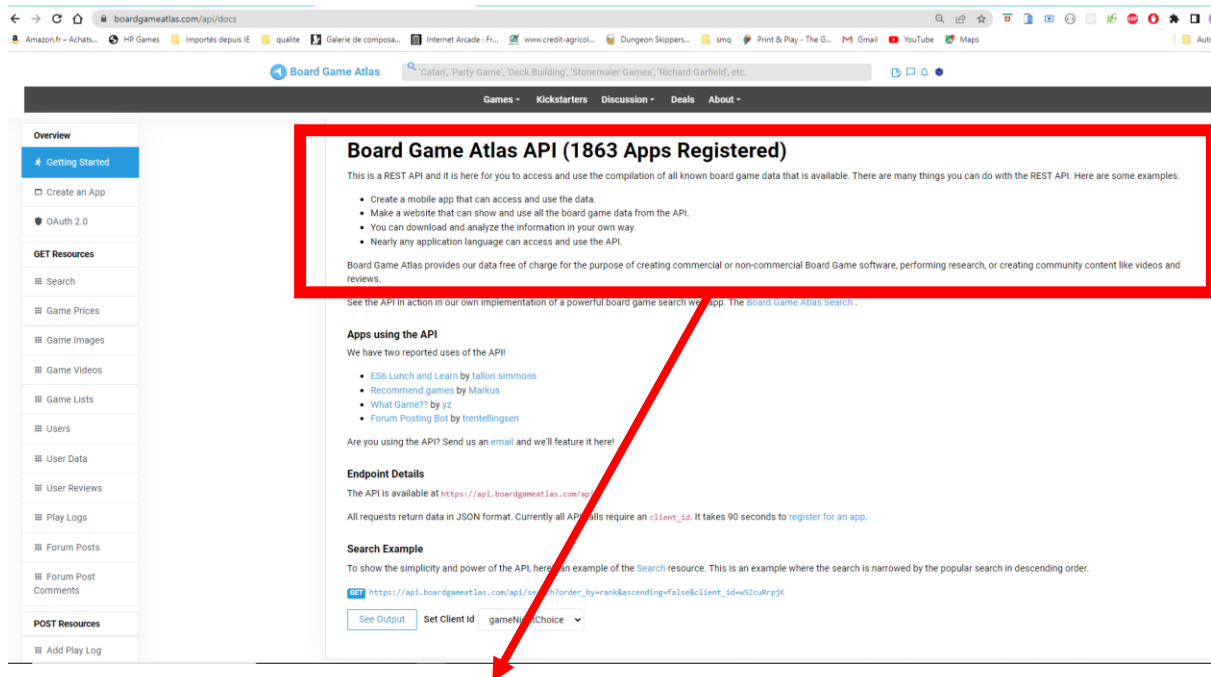
Mes recherches m'ont amené sur l'API *Board Games Atlas*, une API contenant une base de données de plusieurs dizaines de milliers d'entrées :

- <https://www.boardgameatlas.com/api/docs>

Cette API est utilisée par de nombreux sites et applications spécialisées dans le commerce des jeux de société.

La documentation est en anglais

9. Extrait du site de Board Game Atlas API



API Board Game Atlas (1863 applications enregistrées)

C'est une API REST (interface de programmation d'application * ndt) et c'est ici pour que tu accèdes et que tu utilises la compilation de toutes les données disponibles sur les jeux de société existant ou ayant existés. Il y a beaucoup de chose que vous pouvez faire avec l'API REST. Voici quelques exemples :

- Créer une app mobile qui peut accéder et utiliser les données.
- Créer un site web qui peut montrer et utiliser toutes les données des jeux de société de l'API.
- Vous pouvez télécharger et analyser les informations comme vous le souhaitez.
- Presque toutes les applications de traductions peuvent accéder et utiliser cette API.

Board Game Atlas fournit ces données sans frais dans le but de créer une app à but commercial ou non, effectuer des recherches, ou créer du contenu communautaire comme des vidéos et des notations.