

ChessJS

Maxime Borgeaud
Cpnv, Pré-TPI 2024



Table des matières

1	Analyse préliminaire	3
1.1	Introduction	3
1.2	Objectifs.....	3
2	Analyse / Conception.....	4
2.1	Diagramme de classe	4
2.2	Maquettes	6
2.3	Concepts	11
2.4	Méthode de gestion de projet	11
2.5	Planification	11
2.6	Stratégie de test.....	13
2.7	Tests de requêtes	13
2.8	Risques techniques	14
2.9	Base de données.....	15
2.9.1	MCD	15
2.9.2	MLD	15
2.9.3	Db finale	16
3	Réalisation.....	16
3.1	Structure des dossiers & fichiers	16
3.1.1	Dossier racine du projet.....	16
3.1.2	Dossier « backend »	17
3.2	Fonctionnalités présentes.....	18
3.3	Description des tests effectués.....	20
3.4	Erreurs restantes	20
3.5	Archives.....	20
3.5.1	Diagramme de classe	21
3.5.2	MCD	21
3.5.3	MLD	22
3.6	Liste des documents fournis	22
4	Conclusions	23
5	Annexes.....	24
5.1	Résumé du rapport du TPI / version succincte de la documentation	24
5.2	Sources – Bibliographie.....	24
5.3	Journal de travail	24
5.4	Manuel d'Installation	24
5.5	Manuel d'Utilisation.....	24
5.6	Archives du projet.....	24

1 Analyse préliminaire

1.1 Introduction

Le projet consiste en un jeu d'échec en réseau. Les joueurs pourront créer de multiples parties avec différents adversaires. Il sera possible d'arrêter la partie en cours puis de la continuer ultérieurement

1.2 Objectifs

Le premier objectif est d'avoir un jeu d'échec fonctionnel. Le déplacement des pièces, la détection des « échecs » & « échecs et mat » doivent se faire selon les règles. Il y a certaines règles que je n'implémenterais pas, notamment le pat, le petit et grand roque ainsi que le changement de grade du pion lorsqu'il atteint le bord de l'adversaire. Plutôt que d'implémenter ces petites règles, j'ai préféré me concentrer sur le deuxième objectif décrit ci-dessous.

Le second objectif est la gestion en temps réel du déplacement des pièces. Je profite de ce projet pour apprendre la manipulation de base des « WebSocket » avec NodeJS. Le serveur doit gérer l'affichage en temps réel en fonction de la couleur du joueur. Les 2 joueurs doivent avoir leurs « camps » de leurs coté et donc les calculs de déplacements doivent être adaptés également. Lorsqu'un joueur déplace une pièce, le déplacement se fait en temps réel sur l'écran de tous les joueurs connectés à la partie.

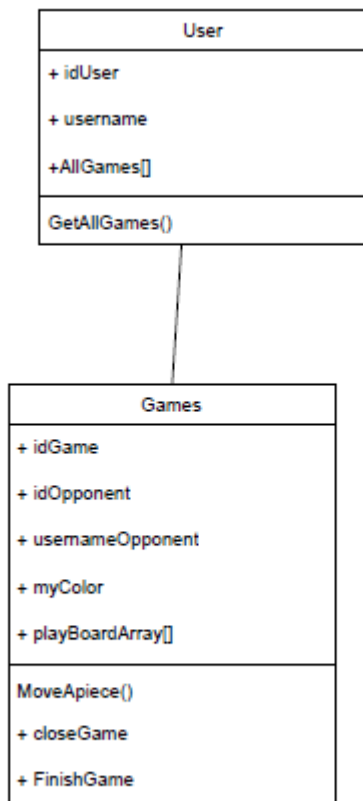
Un troisième objectif est de créer une gestion efficace des utilisateurs lors du push en production. Lors de mon dernier projet, j'ai eu des problèmes d'authentification des utilisateurs car le navigateur bloquait les requêtes POST. A priori l'implémentation des tokens « JWT » devrait résoudre le problème.

Un objectif serait du push en production sur le serveur de swisscenter. Normalement le serveur est dans tous les cas en « production » sur le réseau local.

2 Analyse / Conception

2.1 Diagramme de classe

Classes frontend :

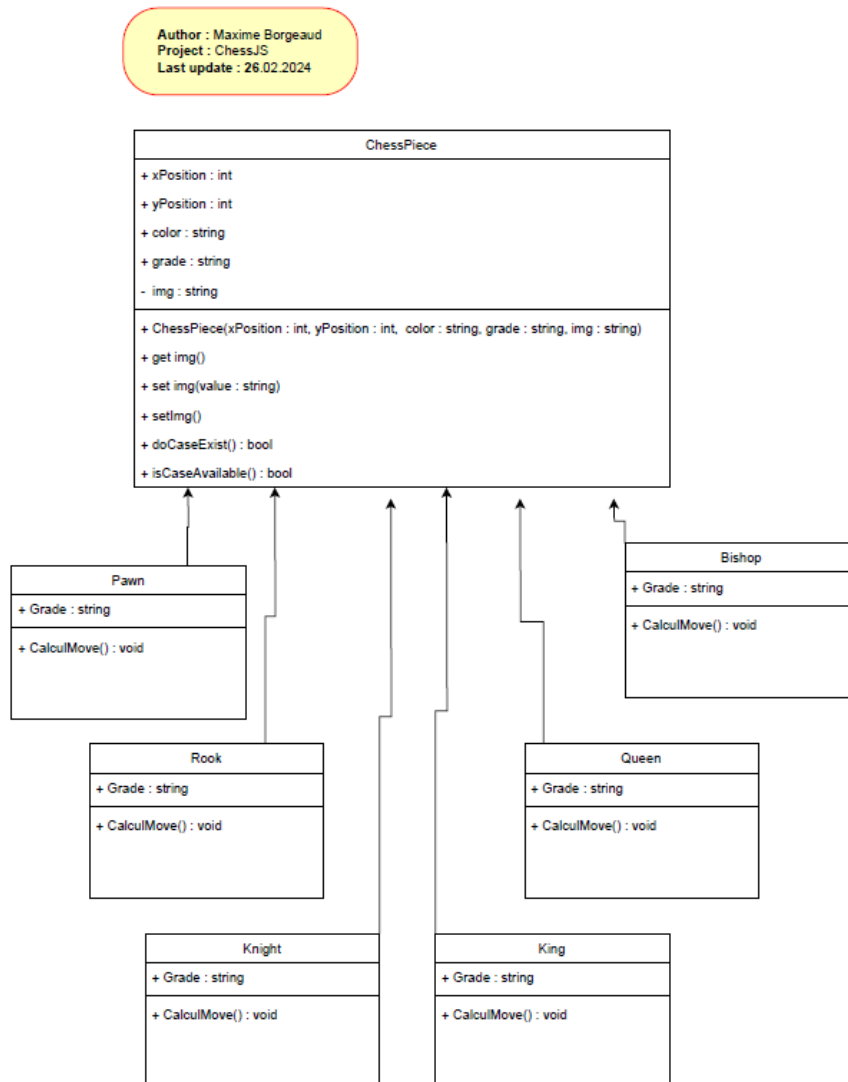


J'ai finalement décidé de rajouter ces deux petites classes afin d'avoir une structure de code simplifiée pour le frontend.

La classe « User » sera stockée dans le sessionStorage ou localStorage et sera utilisée afin de faciliter la gestion d'une partie (créer une nouvelle partie, reprendre une partie en cours etc). Le tableau « AllGames » contiendra toutes les infos de toutes les parties non-finies du joueur actif.

La classe « Games » sera utile pour la gestion d'une partie en cours actuellement, notamment la gestion du tableau de jeu en temps réel, finir une partie etc). Si vous lisez ceci en cours de projet, il est possible qu'il y aie quelques changement.

Classe backend :



2.2 Maquettes

Page d'accueil « user non-logged » :

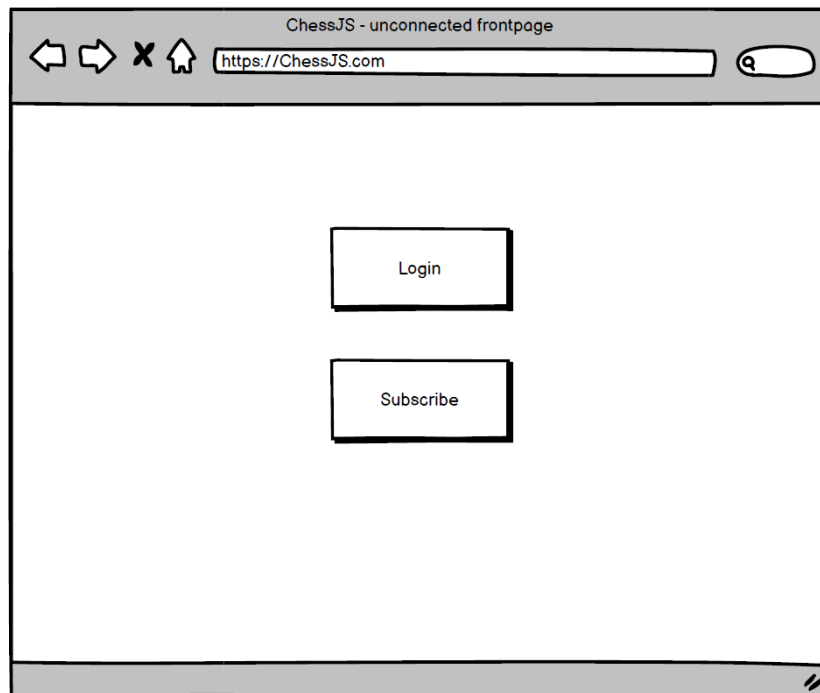


Figure 1

Page d'accueil « user logged » :

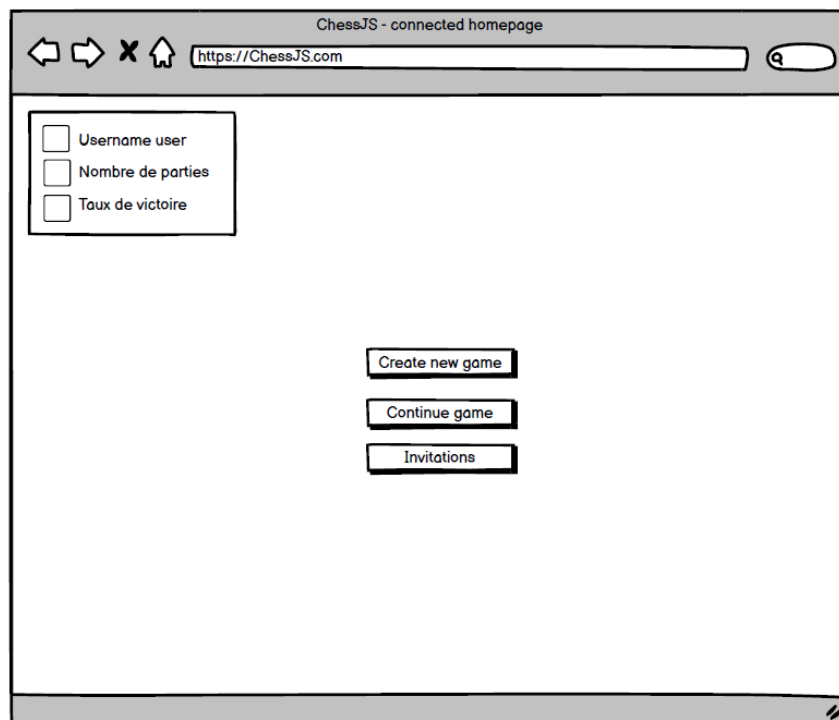


Figure 2

Page de création d'une partie :



Figure 3

Le créateur de la partie choisit sa couleur

Page pour reprendre une partie commencée précédemment :

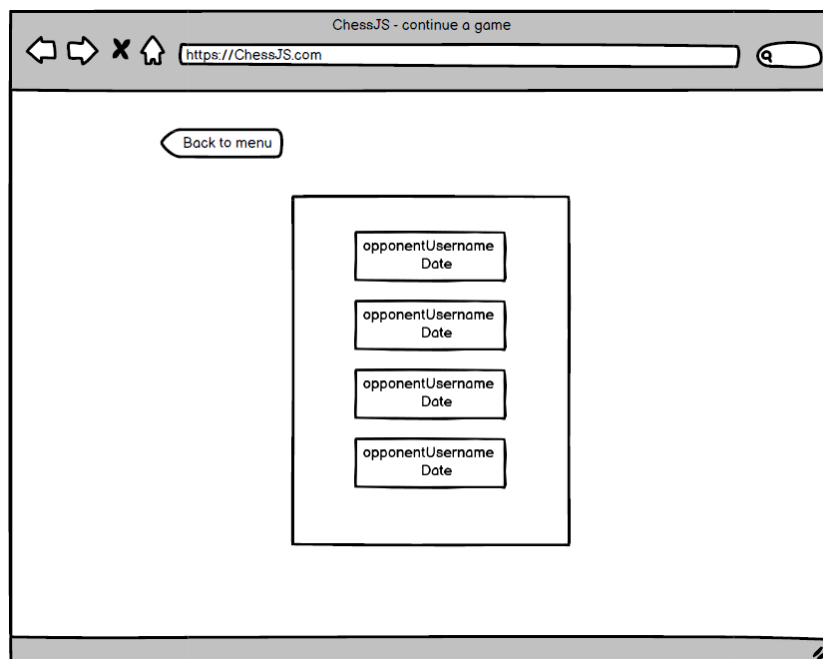


Figure 4

Le nom de l'opposant et la date de création de la partie s'affichent

Page de jeu :

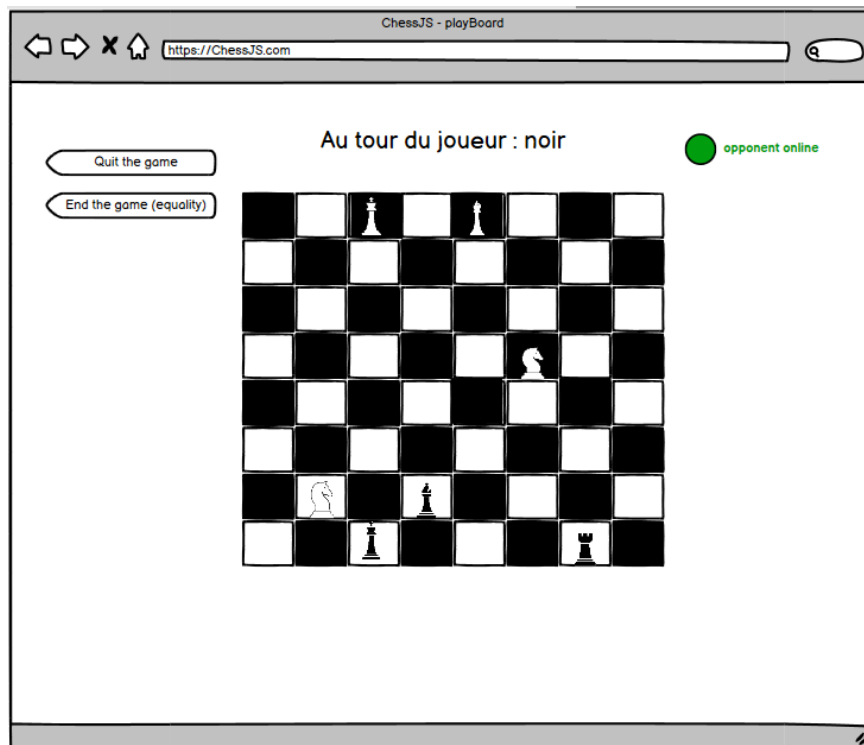


Figure 5

Page de jeu avec « ClickActif » :

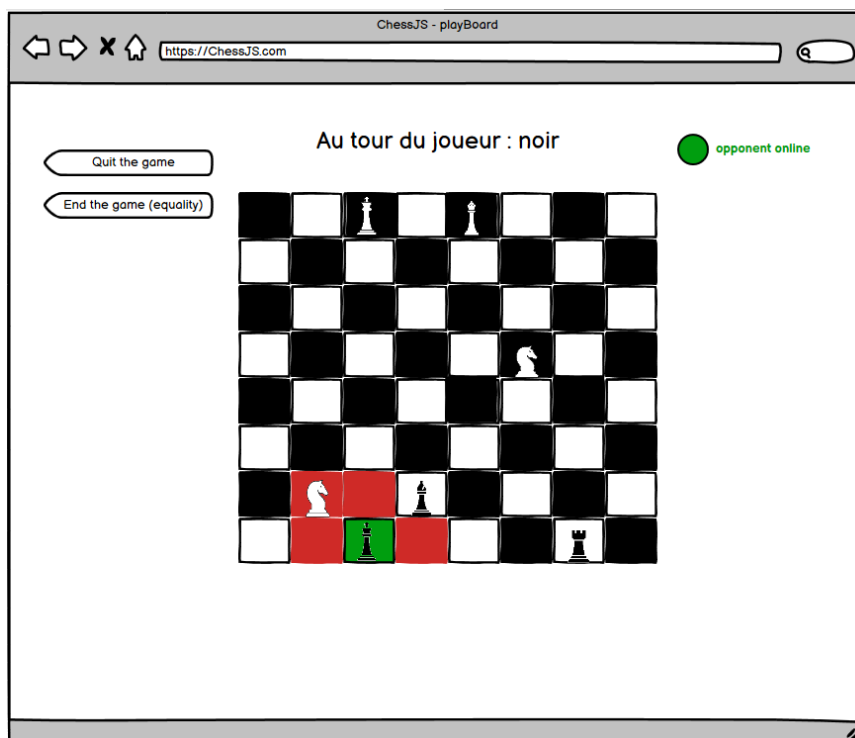


Figure 6

Le joueur noir clique sur son roi pour afficher ses déplacements possibles

Page de jeu « echec & mat » :

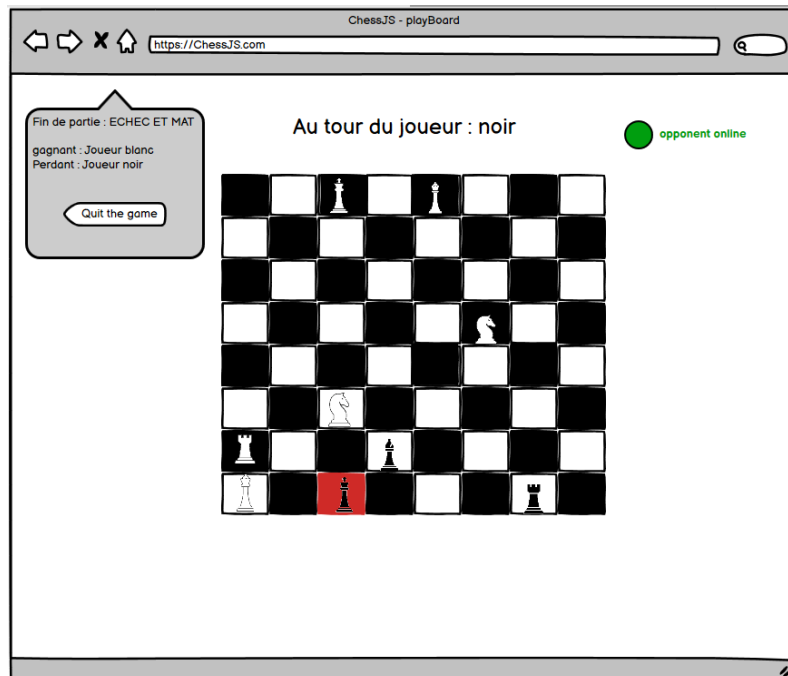


Figure 7

Page de jeu « demande de fin de partie (égalité) » :

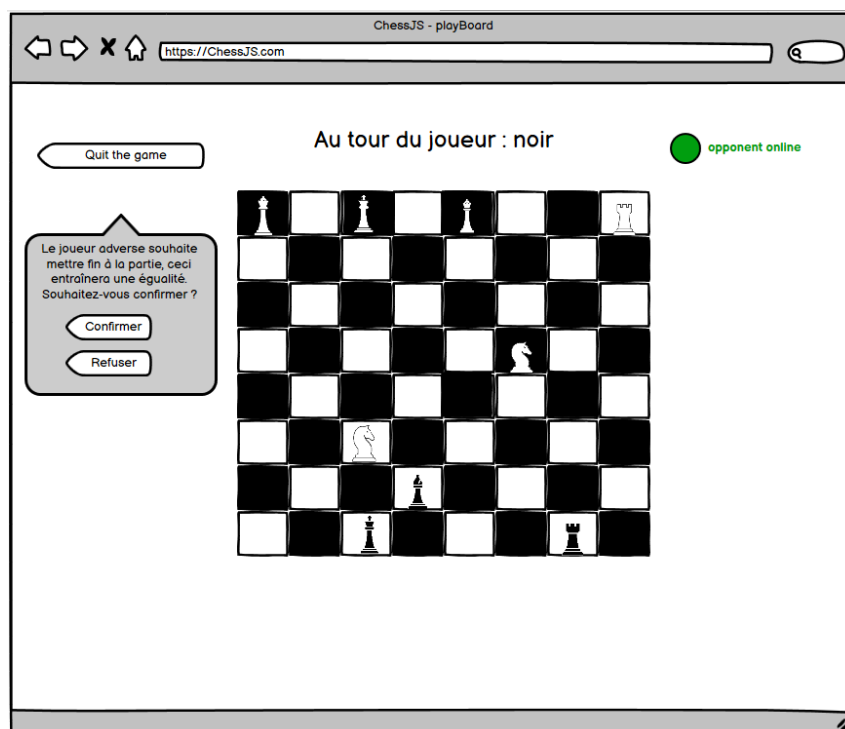
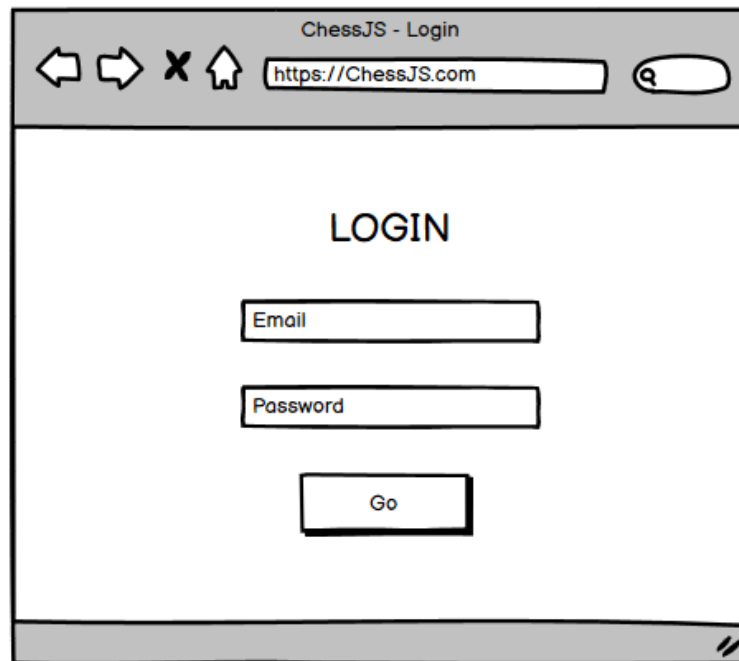


Figure 8

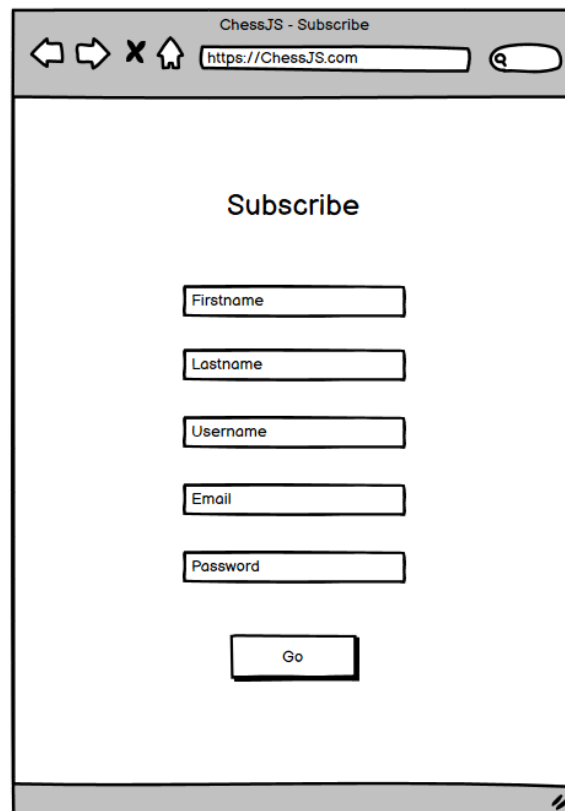
Page de login :



The diagram illustrates a web browser window for a login page. The browser's title bar reads "ChessJS - Login". The address bar contains the URL "https://ChessJS.com". The main content area is white and features the word "LOGIN" in a large, bold, sans-serif font. Below the title, there are three input fields: "Email", "Password", and a "Go" button. The "Email" and "Password" fields are rectangular with rounded corners and a thin border. The "Go" button is a smaller rectangular button with a thin border. The browser window has a grey header bar and a grey footer bar. The footer bar contains a small icon in the bottom right corner.

Figure 9

Pas d'inscription :



ChessJS - Subscribe

https://ChessJS.com

Subscribe

Firstname

Lastname

Username

Email

Password

Go

Figure 10

2.3 Concepts

Voici les principaux éléments principaux qui composent le projet

- ⇒ Un serveur web NodeJS Express
- ⇒ Une base de donnée MariaDb
- ⇒ Un serveur WebSocket NodeJS
- ⇒ Un système d'authentificationJWT
- ⇒ Stockage des mots de passe encryptés (hash)
- ⇒ Modèle MVC
- ⇒ POO

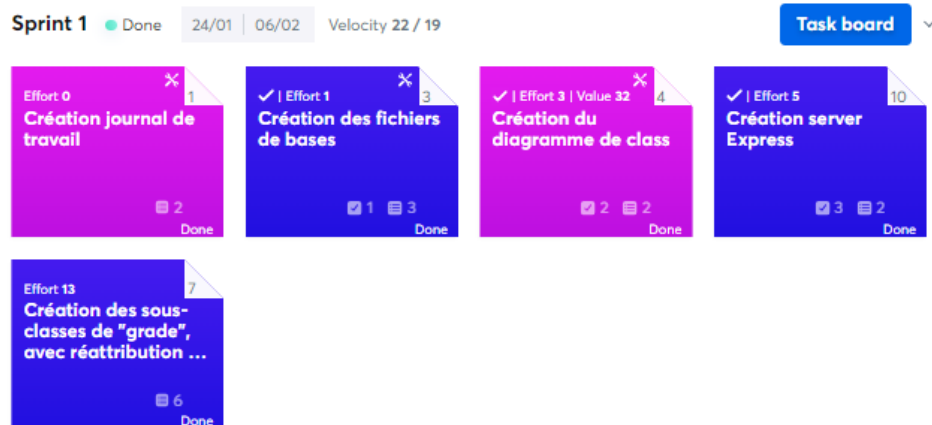
2.4 Méthode de gestion de projet

J'utilise la méthode Agile. J'ai défini 5 sprints de deux semaines dans lesquels j'ai mis les tâches à faire, qui sont détaillés dans la section ci-dessous

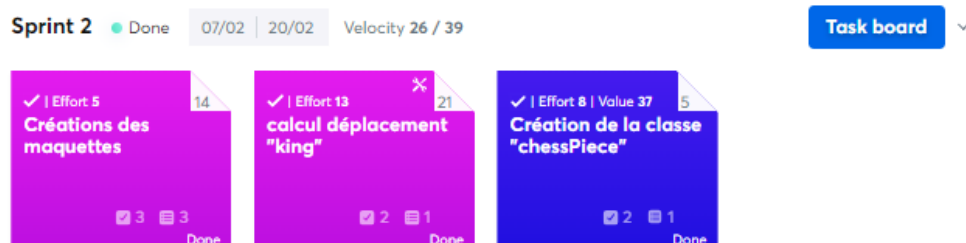
2.5 Planification

La planification se trouve dans le IceScrum du projet que je vous remets ci-joints :
<https://icescrum.cpnv.ch/p/CHESSJS/#/planning>

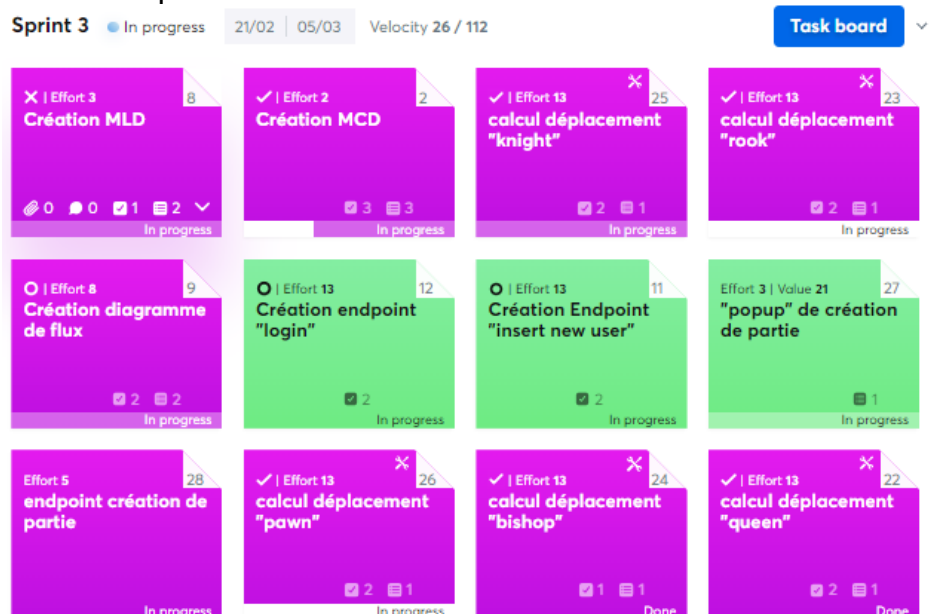
Sprint goal : La structure de base du projet est posée



Sprint goal : Déplacements possibles des pièces visible (en partie réussi)

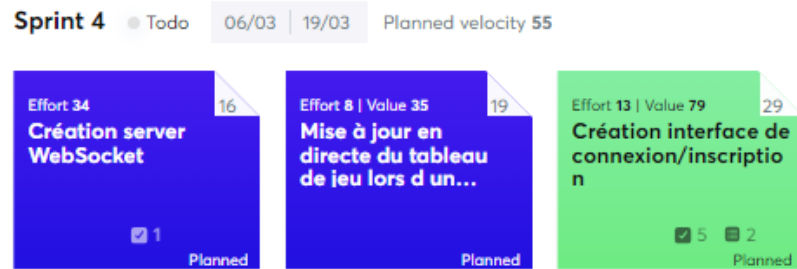


Sprint goals : Affichage de tous les déplacements possibles + Les endpoints nodeJS de connexions/inscriptions sont faits



A l'origine, tous les calculs de déplacements des pièces étaient dans le sprint précédents. Ils ont dû être repoussés car le temps n'a pas été suffisant pour terminer ces tâches à 100%. (90% de ces tâches étaient complètes à la fin du sprint 2)

Sprint goal : Déplacement des pièces opérationnel



Sprint goal : Code fin de partie + finitions de l'interface



2.6 Stratégie de test

J'ai pris le parti de ne pas faire de test unitaire et au contraire de tester mon code de façon « manuelle » avec des « console.log() ». J'ai conscience que ce n'est pas la façon la plus propre de réaliser un projet de développement mais la raison de ce choix et mon manque de temps et de connaissance en la matière. Les tests unitaires en Javascript ne faisant pas partie des cours que nous avons suivi. J'ai préféré axer mes recherches d'acquisitions de connaissances dans les WebSocket, POO en JS et JWT plutôt que les tests unitaires.

2.7 Tests de requêtes

Vous trouverez ci-dessous un exemple de code une fonction qui envoie une requête vers le serveur et retourne une valeur. Cette fonction insère un nouvel utilisateur dans la base de données. Afin de tester cette requête et le code qui le traite voici la procédure que j'ai faite :

- 1) Ecriture du code
- 2) Envoi de la requête au serveur
- 3) Affichage de la réponse dans la console client
- 4) Comparaison du comportement attendu & comportement effectif dans la db selon la réponse (si le message renvoyé est une erreur, il ne doit pas y avoir eu d'insert et également l'inverse).

- 5) Check si une reponse s'affiche dans la console client (console.log

```
export default async function startNewGameRequest(idGame){
  console.log("entrée start newGame Request : ")
  try{
    const request = await fetch(`http://10.229.32.215:3000/api/startNewGame?id
      method: 'GET',
      headers: {
        'Content-Type': 'application/json'
      }
    })
    if(request.ok){
      let result = await request.json()
      return result
    }else{
      console.log("insertion a failé")
      return null
    }
  }catch(error){
    console.log(error)
  }
}
```

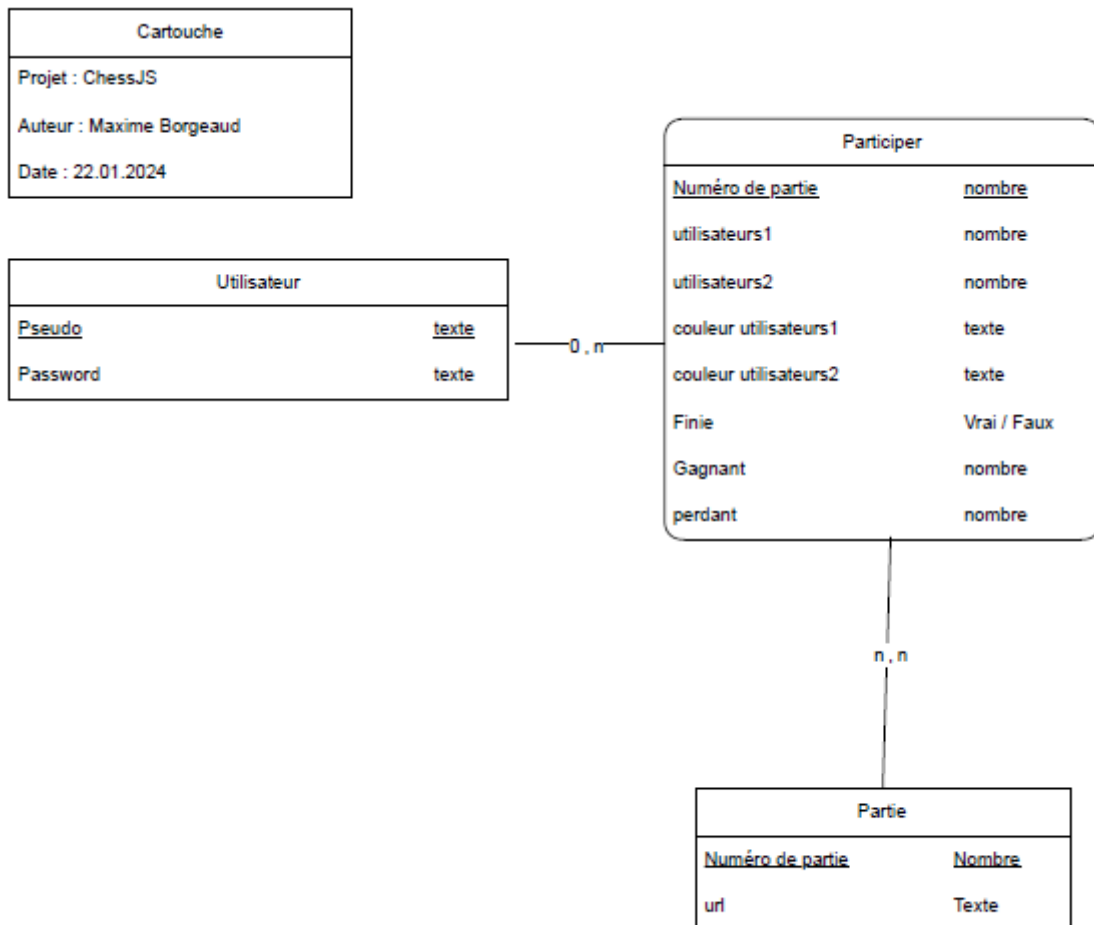
2.8 Risques techniques

Le principale risque est mon manque de connaissance en matière de WebSockets. Etant un sujet que j'apprend en cours de projet, il n'est pas impossible que certaines difficultés ralentissent grandement mon avancement du projet.

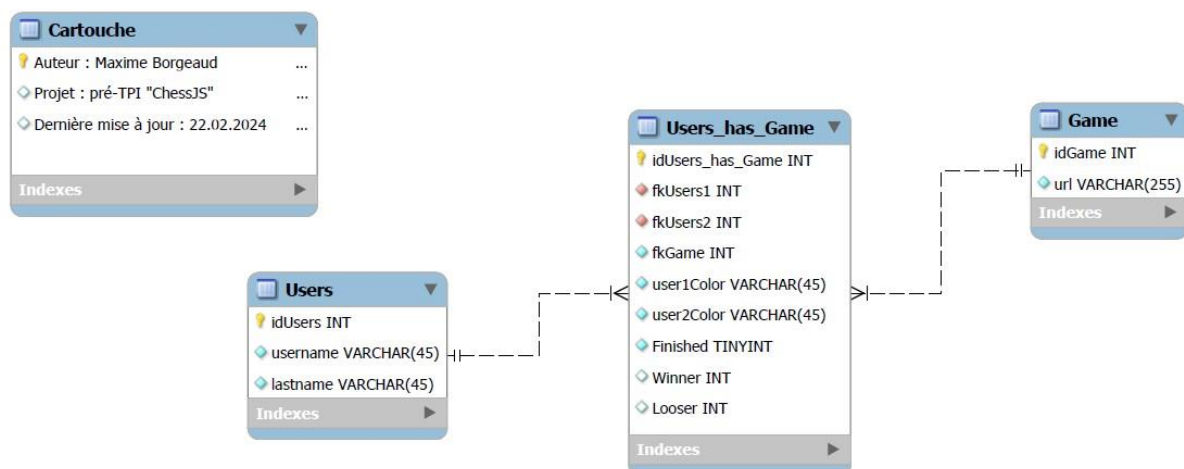
Un autre risque technique est le fait que mon serveur web communique avec des requêtes http, Google chrome bloque automatiquement ces requêtes.

2.9 Base de données

2.9.1 MCD



2.9.2 MLD



2.9.3 Db finale

Je vous précise que les clefs jaunes représentent les primary keys et les clefs grises représentent les champs qui ont une contrainte d'unicité

Table « Games » :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	idGame	int(255)			Non	Aucun(e)		AUTO_INCREMENT
2	url	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)		

Table « UserAsGame » :

#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	idUserAsGame	int(255)			Non	Aucun(e)		AUTO_INCREMENT
2	fkUser1	int(255)			Non	Aucun(e)		
3	fkUser2	int(255)			Non	Aucun(e)		
4	fkGame	int(255)			Non	Aucun(e)		
5	fkActualPlayer	int(255)			Oui	NULL		
6	user1Color	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)		
7	user2Color	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)		
8	finished	tinyint(1)			Non	0		
9	winner	int(255)			Oui	NULL		
10	looser	int(255)			Oui	NULL		
11	createdAt	date			Non	Aucun(e)		

Table « User » :

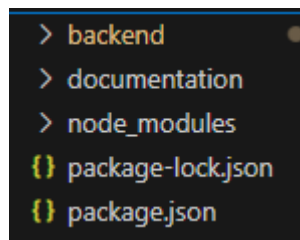
#	Nom	Type	Interclassement	Attributs	Null	Valeur par défaut	Commentaires	Extra
1	idUser	int(255)			Non	Aucun(e)		AUTO_INCREMENT
2	username	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)		
3	password	varchar(255)	utf8mb4_general_ci		Non	Aucun(e)		

3 Réalisation

3.1 Structure des dossiers & fichiers

3.1.1 Dossier racine du projet

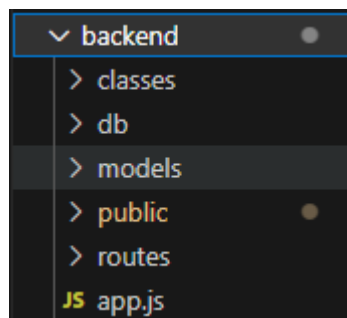
Le serveur tourne sur le port 3000 de l'ordinateur hôte, une fois démarré, l'application est disponible à l'url : <http://adresseIP:3000>. Je conseille l'utilisation de « Microsoft Edge » car parfois Google Chrome bloque les requêtes HTTP
Voici la structure du server :



« backend » : contient tous les fichiers de code du projet
« documentation » : contient tous les fichiers de documentation du projet.
« node_modules » : contient toutes les librairies nécessaires au bon fonctionnement de l'application

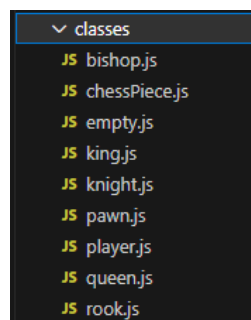
« package-lock.json » & « package.json » : Contiennent les informations sur les dépendances

3.1.2 Dossier « backend »

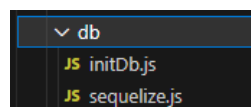


Voici la structure générale du dossier « backend », qui est détaillée précisément ci-dessous

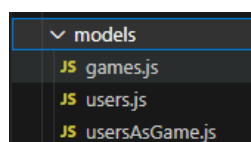
« app.js » : Il s'agit du fichier racine du server Express. Lorsqu'une requête http vise le server, elle est traitée dans ce fichier. Si aucune route n'est trouvée pour la requête, le serveur renvoie une erreur 404.



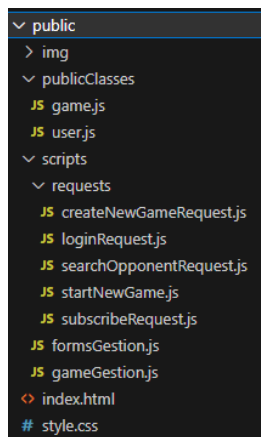
« classes » : Il contient les fichiers de classes non-accessibles depuis le client. Ces classes sont utilisées pour la gestion des pièces lors d'une partie. Effectivement les calculs de déplacements, si échec ou non se feront sur le serveur et non sur le client.



« db » : Il contient le fichier « sequelize » qui fait la connexion avec la base de données. Également le fichier initDb qui fait la synchronisation entre les modèles et la db elle-même. Ce deuxième fichier est exécuté lors du démarrage du server



« models » : Contient les modèles qui servent à faire des requêtes SQL à la db via la librairie « sequelize ». Ces modèles correspondent à des objets identiques aux tables de la db.



« public » : Contient tous les fichiers qui sont automatiquement envoyés au client lorsqu'il fait une requête qui pointe vers la racine du serveur

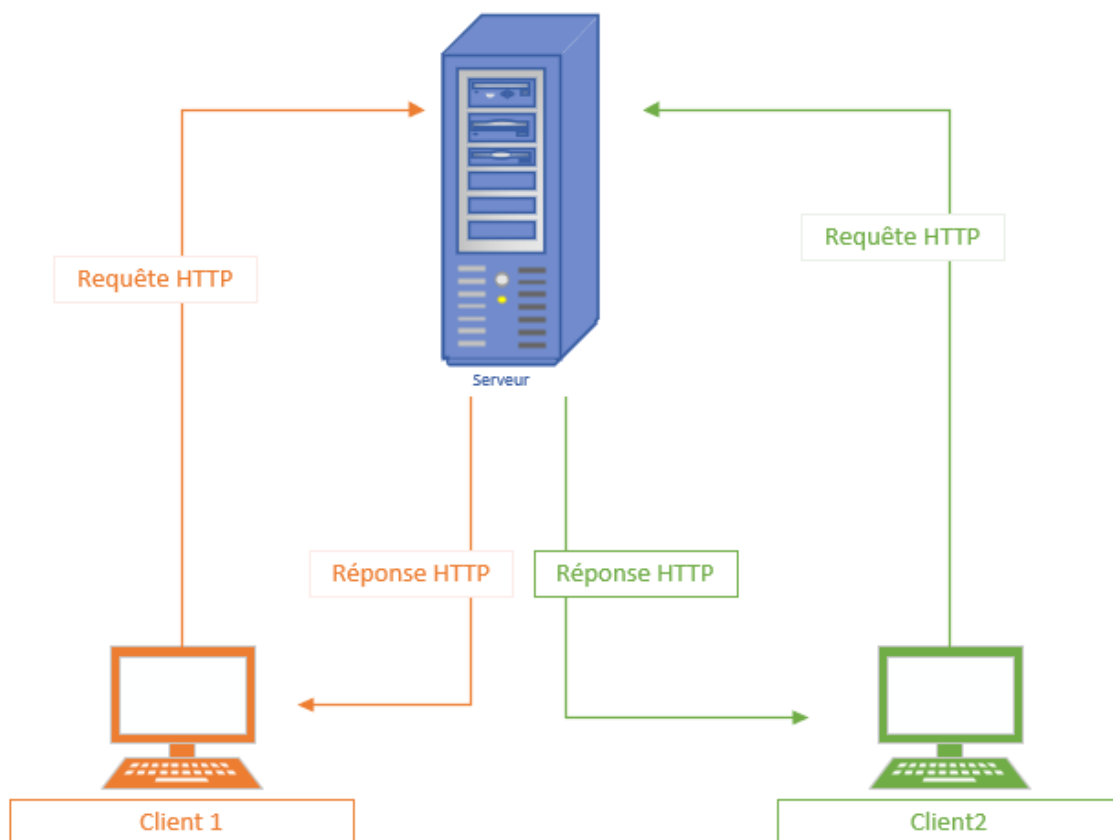
- Le dossier « scripts » contient les fichiers javascripts qui servent pour la gestion de l'interface frontend.
- Le dossier « publicClasses » contient des classes qui rendent le code frontend plus simple

3.2 Requêtes http

Le serveur NodeJS comprend 2 types de requêtes : http & websocket. Voici des schémas la façon dont le serveur gère ces différentes requêtes :

3.2.1 Requêtes http

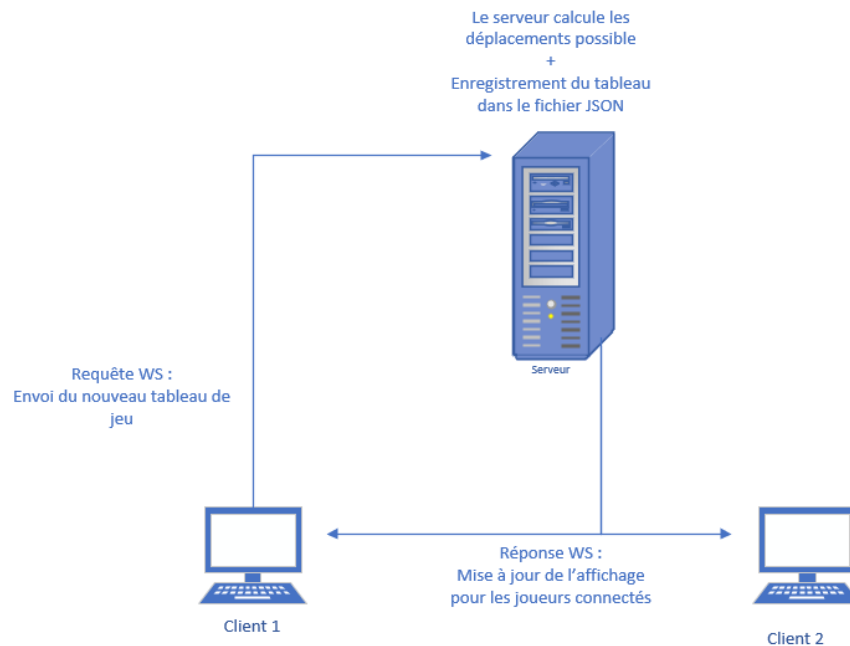
Ce sont les requêtes classiques du protocole http. Lorsqu'un client effectue une requête, le serveur répond à ce client et personne d'autre. Dans ce cas, le serveur a besoin de recevoir une requête d'un client pour pouvoir répondre, sinon il ne peut rien faire.



3.2.2 WebSockets

Une WebSocket est une connexion http persistante, c'est à dire que le serveur et les clients restent connectés en permanence durant une période déterminée.

Concrètement cela signifie que le serveur peut envoyer une requête vers le client sans que celui-ci ne demande quoi que ce soit. J'ai donc utilisé cette technologie pour mettre à jour le tableau de jeu en temps réel lorsqu'un joueur déplace une pièce.



3.3 Fonctionnalités présentes

- 1) Le formulaire d'inscription d'un nouvel utilisateurs fonctionne parfaitement. Il enregistre un nouveau record dans la table « user » et le mot de passe est crypté de façon « hash(10) » grâce au module « bcrypt » de NodeJS.
- 2) Le formulaire de Login fonctionne également. Il va chercher l'utilisateurs demandé, compare le mot de passe entré et le compare avec celui enregistré dans la db. Le serveur renvoie au client si la connexion a fonctionné ou non.
- 3) La fonction OpenConnexion() → Lorsque l'user est connecté, soit via le formulaire de login ou d'inscription, cette fonction est appelée, ce qui ouvre le menu principal et fixe les infos de l'utilisateurs dans le sessionstorage.
- 4) Les pièces ainsi que leurs déplacements possible peuvent s'afficher dans le tableau de jeu. Le client envoie une requête au serveur, qui effectue tout les calculs de déplacements puis renvoie les informations au client qui les affiche correctement sur la page web de destination.

3.4 **Description des tests effectués**

Pour chaque partie testée de votre projet, il faut décrire:

- *les conditions exactes de chaque test*
- *les preuves de test (papier ou fichier)*
- *tests sans preuve: fournir au moins une description*

3.5 **Erreurs restantes**

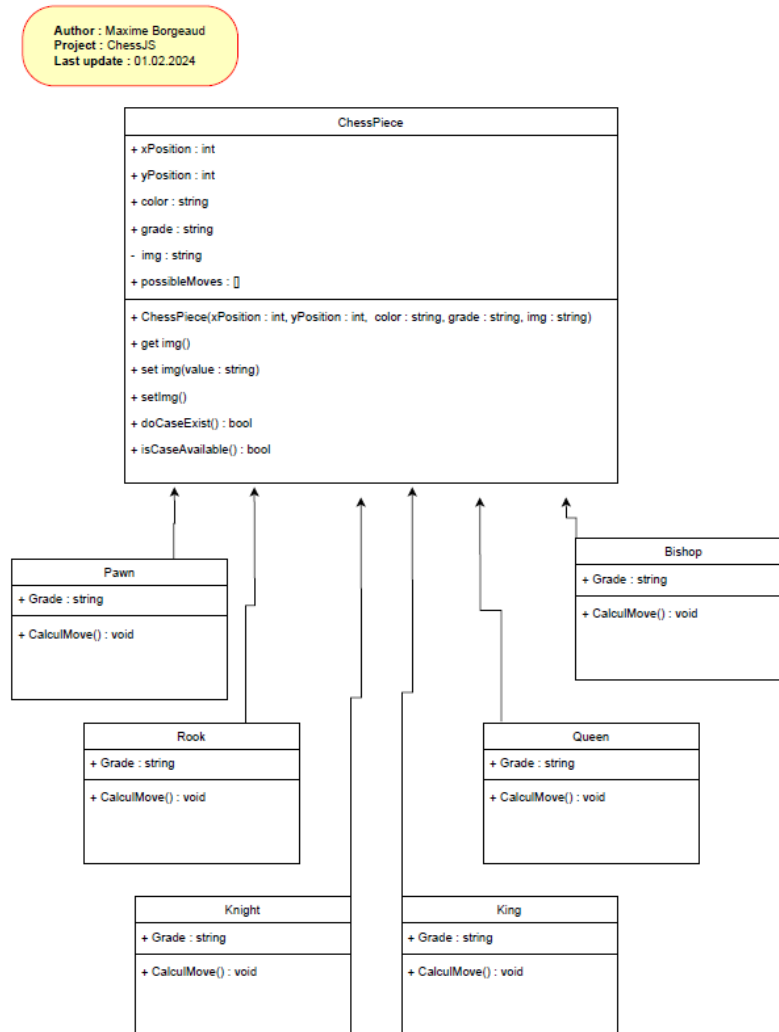
S'il reste encore des erreurs:

- *Description détaillée*
- *Conséquences sur l'utilisation du produit*
- *Actions envisagées ou possibles*

3.6 **Archives**

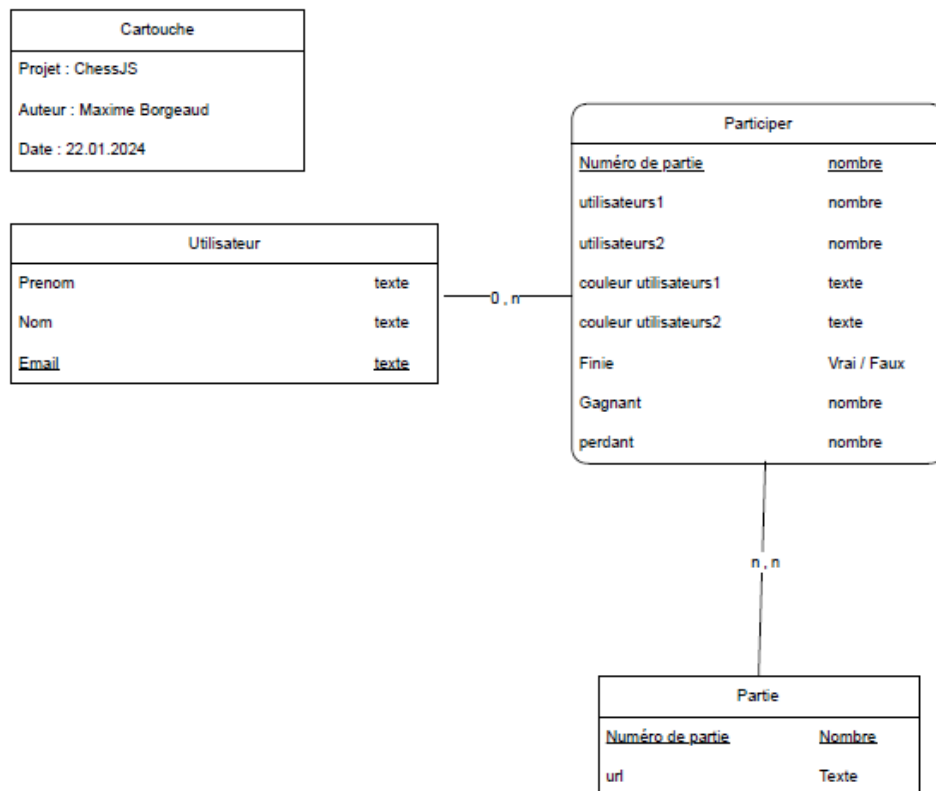
Vous trouverez ci-dessous les premières versions de certains éléments du projet. Ces éléments ont dû être modifiés pour des questions d'optimisations ou il y a eu suppressions d'éléments inutiles ou redondants :

3.6.1 Diagramme de classe

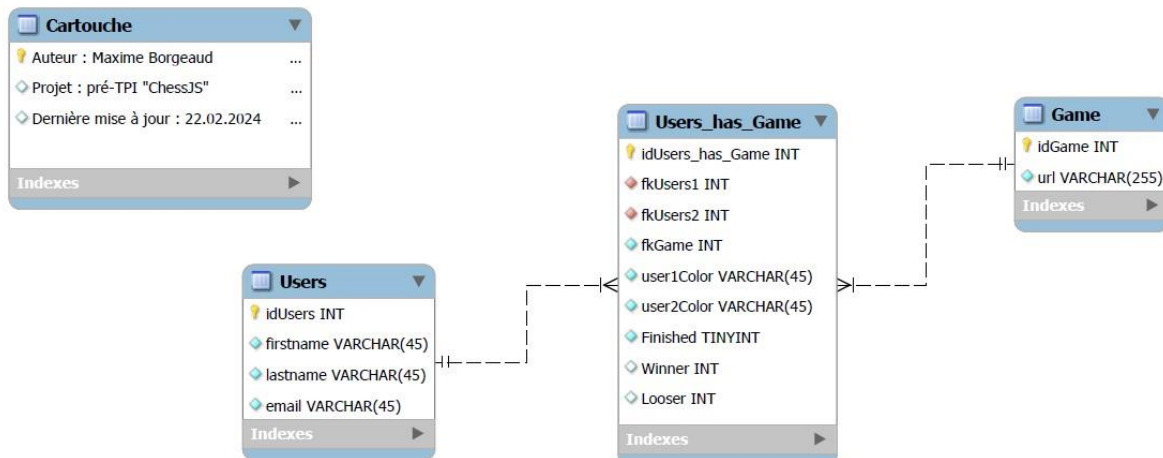


3.6.2 MCD

Dans ce diagramme et également dans le MLD ci-dessous, j'ai remplacé les 3 champs « nom, prénom et email » par « username » pour des questions de simplicités



3.6.3 MLD



3.7 Liste des documents fournis

Lister les documents fournis au client avec votre produit, en indiquant les numéros de versions

- le rapport de projet
- le manuel d'Installation (en annexe)
- le manuel d'Utilisation avec des exemples graphiques (en annexe)
- autres...

4 Conclusions

Développez en tous cas les points suivants:

- *Objectifs atteints / non-atteints*
- *Points positifs / négatifs*
- *Difficultés particulières*
- *Suites possibles pour le projet (évolutions & améliorations)*

5 Annexes

5.1 Résumé du rapport du TPI / version succincte de la documentation

5.2 Sources – Bibliographie

Liste des livres utilisés (Titre, auteur, date), des sites Internet (URL) consultés, des articles (Revue, date, titre, auteur)... Et de toutes les aides externes (noms)

5.3 Journal de travail

Date	Durée	Activité	Remarques

5.4 Manuel d'Installation

5.5 Manuel d'Utilisation

5.6 Archives du projet

Media, ... dans une fourre en plastique "

Classes :

Première idée de classes :