Mini-Jeux : Rapport de projet

Application Web et Sécurité

OUHOCINE Sarah

Université de Versailles Saint-Quentin-en-Yvelines

Mai 2022

Table des matières

1 Introduction

Ce document présente notre rapport final pour l'application qui a été développée dans le cadre du projet de l'UE "Application Web et Sécurité". L'objectif de ce module est de découvrir le monde du web (**HTML** / **CSS** / **JS**) et de comprendre et assimiler les concepts avec un projet consistant à créer une application web.
Nous avons choisi de partir sur un site web proposant aux utilisateurs 3 mini-jeux.
Il se compose des jeux suivants : le Morpion, le Sudoku et le Snake.

2 Accueil

L'accueil est la page principale qui apparaît lorsqu'un utilisateur arrive sur notre site Web. C'est par le biais de cette page qu'une personne peut avoir le descriptif des trois mini-jeux que l'on propose.

2.1 Techniques utilisées

Notre projet a été développé via le framework **Express** et **NodeJS** pour mettre en place le serveur web de notre application. L'outil technique présent, en ce qui concerne la page d'accueil, est principalement **Bootstrap**. Ayant besoin d'avoir un outil permettant de naviguer entre les différents mini-jeux facilement, **Bootstrap** nous a permis de construire une barre de navigation réactive.

2.2 Structures

2.2.1 accueil.html

Ce fichier est celui qui permet de mettre en forme la page d'accueil. Le code est structuré autour de deux balises distinctes, la balise *head* et *body*.

La partie *head* regroupe les imports nécessaires pour les bibliothèques utilisées, le titre et le logo de l'onglet.

La partie *body* est décomposée en trois sous-section :

- La barre de navigation (*navbar*) et son menu déroulant (*dropdown-menu*). C'est le template commun qui est présent dans chaque code des différents mini-jeux.
- Le titre et le bouton 'Explore'.
- Le tableau descriptif des mini-jeux. Chaque case du tableau est issu de **Bootstrap** via l'instanciation de la classe *card border-primary*. Les trois éléments du tableau sont réunis grâce à la classe *card-group*.

2.2.2 accueil.css

Ce fichier est celui qui permet de manipuler le style de la page d'accueil. C'est dedans que l'on peut retrouver par exemple le détail pour avoir les hachures sur le texte. (template provenant de *Mandy Michael*)

2.2.3 accueil.js

Ce fichier est celui qui permet de manipuler la fenêtre de la page d'accueil. Lorsque le bouton 'Explore' est cliqué, la page subit un défilement (*scroll*). L'effet du scroll automatique se traduit par l'ajout de margin (*style.marginTop*) au niveau du titre, du bouton 'Explore' et la barre de navigation.

2.3 Problèmes rencontrés et solutions

Un problème que l'on peut retenir dans cette section est que nous avons eu du mal à rendre la page *responsive*. En effet, dans un premier temps, le titre 'Accueil' ainsi que le bouton 'Explore' se chevauchait lorsque la fenêtre se réduisait, ce qui n'était pas esthétique. La solution pour résoudre ce problème a été d'ajouter une classe *btn btn-primary* au bouton. **Bootstrap** rend directement ces éléments responsive, ce changement nous a permis de corriger ce défaut.

3 Morpion

Le Morpion (ou X/O) est le premier mini-jeu qui est proposé aux utilisateurs de notre application. La partie Morpion est la seule qui communique avec le serveur(**app.js**). Ce mini-jeux est en ligne et se lance lorsque deux joueurs sont connectés à un même salon.

3.1 Techniques utilisées

Le Morpion est un mini-jeu qui a bénéficié de nombreux outils techniques divers. Premièrement, pour établir les connexions des joueurs nous avons utilisé **Socket.io**. Cette bibliothèque permet aussi d'avoir une communication entre un joueur et le serveur. De plus, c'est grâce à cet outil que le mécanisme du Morpion a pu être simulé en mettant à jour la grille de jeu en temps réel et d'intégrer un channel de discussion intéractif. L'objectif d'utilisation de **Socket.io** était d'avoir des connexions bi-directionnelles entre clients et serveur afin de garder en permanence cette liaison tant qu'un utilisateur est connecté.

EJS est une technologie que nous avons employé afin de pouvoir introduire des boucles dans notre fichier **XO.ejs**. Le but recherché était de pouvoir afficher à un utilisateur, tous les joueurs qui ont été enregistré par notre base de donnée.

Nous avons décidé de stocker dans une base de donnée le pseudo des joueurs qui se connectent au Morpion. C'est pourquoi, nous avons employé comme système de base de donnée celui de **MongoDB**. Comme la base de donnée de notre application enregistre que des noms la seule requête qui sera exécuté est une requête d'insertion d'un String. Notre choix de s'orienter vers ce type de structure se justifie car nous avons qu'une seule table dans notre base donc aucune relation complexe(clés étrangère). Une base de donnée structurée en *NoSQL* suffit pour répondre à nos besoins, l'exploitation de **Mongoose** permet de modéliser notre base de donnée *MongoDB* et la lier à *NodeJS*. Enfin nous avons du mettre en ligne notre base donnée, ce qui implique la création d'un cluster via le cloud de **MongoDB** à l'aide de la plateforme **MongoDBAtlas**.

La bibliothèque **JQuery** a été sélectionnée pour faciliter la manipulation des éléments de la page du fichier **XO.ejs**. Plus particulièrement au niveau du formulaire de connexion, c'est cet outil technique qui permet de soumettre le pseudo d'un joueur au serveur. La partie intéractive, pour les symboles inscrits dans chaque case, est gérée par cette même bibliothèque.

La librairie **Bootstrap** est aussi présente dans cette partie. Elle est utilisée pour garder le template de la barre de navigation, avoir des pop-ups réactives et l'organisation des boutons et de la grille de jeu.

3.2 Structures

3.2.1 XO.ejs

Ce fichier fait office d'organisation de la page web du *Morpion*. La hiérarchie du code est la suivante :

- La barre de navigation : classe *navbar* et le menu déroulant *dropdown-menu*
- L'interface de connexion pour l'utilisateur : balise **form**, l'input a été programmé pour contenir un pseudo entre 2 et 20 caractères maximum. Pour éviter d'avoir des suggestions de pseudos déjà inscrits, il a fallu rajouter la ligne de code suivante : *required autocomplete="off"*
- La salle d'attente : message indiquant qu'un salon a bien été créé (classe *alert-success*), un spinner pour simuler un chargement (classe *spinner-border*) et un lien permettant d'inviter un autre joueur.
- La grille de jeu : balise **table** dont les cases sont formées à l'aide des balises **tr** et **td**. À noter que chaque case se caractérise par son id (*id="case-1-1"*).
- Quatre Pop-ups : la première étant créé manuellement à l'aide du fichier **XO.css**, les trois autres sont produites en manipulant la classe *modal* de **Bootstrap**.
- La liste des joueurs enregistrés : balise **table**, le tableau regroupe deux colonnes. La première colonne représente le numéro du joueur et la seconde permet d'identifier son pseudo.
- La BoxChat : elle comporte balise **div** afin d'afficher les messages. Le fait d'avoir des messages indiqués à la manière d'un réseau social est dû à la partie *messages :nth-child(odd)* du fichier **XO.css**. De plus, la BoxChat est mise en lien avec la base de donnée car elle récupère les pseudos des joueurs avec comme objectif de pouvoir envoyer des messages privés à un destinataire en particulier.

La plupart des balises présentes dans ce fichier ont dans leur déclaration la classe *d-none*. Cela permet de pouvoir les cacher et des les faire apparaître au moment voulu.(exemple : afficher la grille de jeu que si deux joueurs sont bien connectés à un même salon.)

3.2.2 XO.js

Ce fichier Javascript contient toutes les fonctionnalités du Morpion (fonction pour les messages et pop-ups, manipuler les différents affichages mais aussi le déroulement du mini-jeu). Les caractéristiques par défaut d'un joueur y sont précisées. Comme on n'affiche pas tous les éléments du fichier **XO.ejs** en même temps, l'affichage est manipulé par le Javascript (exemple de fonction : *affichageInGame*). C'est ce fichier qui met en connexion les différents évènements de la page web du Morpion au serveur. Chaque action est envoyée à l'aide de **Socket.io**, cette utilisation nous a permis de gérer les actions que devra diffuser le serveur par la suite comme par exemple l'envoi d'un message ou alors actualiser la grille de jeu du Morpion. Pour illustrer cela, on peut prendre le cas des deux fonctions *socket.on('newUser')* et *socket.on('play')*.

De nombreuses méthodes ont été implémentées voici quelques exemples :

— **startGame()** : lance une partie de X/O.
— **videCase()** : efface tous les symboles d'une partie.
— **setTurnMessage()** : affiche le tour du joueur.
— **refreshSaisie()** : efface l'input dès qu'un message est envoyé.
— **afficheNotif()** : permet d'afficher un symbole de notification sur le bouton **Chat** .

3.2.3 app.js

Ce fichier Javascript a pour rôle d'être le serveur de notre application. C'est lui qui fait appel à la bibliothèque **Express** et met en place les différentes routes permettant de naviguer entre les différentes pages web, à savoir les trois mini-jeux ainsi que celle qui amène à la page d'accueil. La connexion au cluster pour la base donnée(*mongoose.connect()*) et l'appel aux frameworks nécessaires au projet(*app.use('/bootstrap/js')*) est effectué dans ce fichier.

La structure du serveur est simple, elle se base sur deux axes importants :
— la connexion d'un joueur : *io.on('connection', (socket))*
— la déconnexion d'un joueur : *socket.on('disconnect', ())*

Une fois qu'un joueur est connecté, le serveur écoute les évènements potentiels qu'il est susceptible de recevoir et réagit en conséquence. Il est en permanence à l'écoute dès qu'un joueur est identifié. À chaque évènement reçu par le client(*ex :socket.on('play', (player))*), une réponse est immédiatement envoyée par le serveur(*io.to(player.roomId).emit('play', player)*).
Lorsqu'un joueur se déconnecte, le serveur avertit tous les autres joueurs connectés. De plus, on supprime ce joueur du tableau qui enregistre les utilisateurs connectés(*connectedUsers[]*).

Voici l'implémentation de **l'insertion** d'un Joueur dans la base de donnée :

```
1       let user = new User();
2       user.pseudo = player.username;
3       user.save();
```

Enfin, quatre fonctions sont présente dans ce fichier. D'une part pour créer un salon de jeu accompagné de son id. D'autre part avoir un mécanisme de sécurité qui vérifie l'unicité des pseudos et de modifier les doublons de manière aléatoire.

3.2.4 joueur.model.js

Ce fichier Javascript permet d'instancier la création de la table de notre base de donnée utilisant la méthode de *Schema* propre à la bibliothèque **Mongoose**. L'attribut en question est le pseudo d'un joueur.

3.3 Problèmes rencontrés et solutions

Au cours de l'implémentation de la partie Morpion de nombreux problèmes nous ont ralenti dans notre travail. Premièrement la gestion de superposition des différentes pop-ups avec l'interface de jeu était parfois mal organisée. La fenêtre modale pouvait apparaître sans pour

autant cacher les autres éléments or nous voulons que celle-ci passe en premier plan. Pour y remédier nous avons alors opté pour ajouter le style *d-none* aux classes des éléments devant être camouflés. Un point notable que nous avons mis du temps à comprendre est que lorsqu'un joueur se déconnecte, toutes les informations relatives à son socket sont détruites. De ce fait, il était compliqué de créer plusieurs évènements en lien avec cette action. Par exemple, pour arrêter une partie suite à une déconnexion il fallait stocker à l'avance chaque id du salon de chaque joueur afin de pouvoir envoyer des informations a l'adversaire seul dans un salon. Ce que nous pouvons retenir dans ce cas est que malgré le fait que la méthode de déconnexion de **Socket.io** n'accepte aucun argument, il est possible d'ajouter des informations aux sockets. (exemple :*socket.idRoom* et *socket.pseudo*).

En ce qui concerne la BoxChat, le seule problème qui est apparu était que parfois les messages affichés débordaient. Pour corriger cette étape nous avons ajouté au fichier **XO.css** l'indication : *overflow-y : auto*. Par ailleurs, l'émission de message privé fut complexe mais grâce à la méthode *broadcast*, il était possible que le serveur spécifie le destinataire d'un évènement. La distinction entre message global et privé a été rapidement rectifiée.

4 Sudoku

Le deuxième mini-jeu proposé par notre application est le Sudoku.

4.1 Techniques utilisées

Pour le développement de la partie Sudoku, aucune technique particulière n'a été employée. La programmation de ce mini-jeu est structurée en **HTML**, **CSS** et **Javascript**.

4.2 Structures

4.2.1 Sudoku.html

Ce fichier est celui qui permet de mettre en forme la page web du Sudoku. Le code est structuré autour de deux balises distinctes, la balise *header* et *body*. La partie *header* comporte les trois niveau de difficultés, un chronomètre, un bouton pour modifier le thème et un bouton de lancement. La balise *body* va contenir la grille du Sudoku.

4.2.2 Sudoku.js

Ce fichier Javascript contient toutes les fonctionnalités du Sudoku. Les modes de difficultés sont des clés représentées sous forme *d'array* composée de deux *String*([grille][solution]). Les variables qui ont été créées sont :

- Timer et TimeRemaining sont pour le temps et le temps restant pour finir la grille.
- selectedNum et selectedTile sont pour le numéro sélectionné et la case sélectionnée de la grille.
- Lives est le nombre de vie restant.
- disableSelect (pour éviter qu'un joueur sélectionne d'autre numéro à la fin de la partie).

De nombreuses méthodes ont été implémentées pour le mini-jeu :

- **startGame()** : lance une partie de Sudoku.

— **generateBoard()** : génère la grille de jeux à partir de la difficulté choisit.
— **startTimer ()** : démarre le chronomètre.
— **checkCorrect()** : vérifie si la nombre inscrit correspond à la valeur d'une case solution.
— **updateMove()** : appelle la fonction **checkCorrect()** et soustrait de 1 le nombre de vie en cas d'erreur.

4.3 Problèmes rencontrés et solutions

Un problème rencontré pour le Sudoku était que l'ajout de la barre de navigation décalait la grille. Après quelques recheches nous nous sommes aperçus que cela venait du fait que **Bootstrap** prenait déjà en compte une configuration prédéfinie au niveau des *box-sizing*. Nous avons dû désactiver ces paramètres par défaut afin de garder le positionnement de notre application.

5 Snake

Le dernier mini-jeu que l'on a voulu mettre en avant est le Snake. Les règles de notre mini-jeu sont qu'un serpent ne peut pas traverser les murs, il a à sa disposition plusieurs pommes qui influencent le score du joueur et la partie se termine si le serpent entre en contact avec un obstacle.

5.1 Techniques utilisées

Pour créer ce jeu, nous avons utilisé le Framework **Phaser**. C'est un framework open source de création de jeux vidéo 2D, il utilise :

— **HTML5 Canvas** : pour afficher le jeu.
— **JavaScript** : pour exécuter le jeu.

L'avantage d'utiliser **Phaser** est qu'il présentait déjà une variété d'outil pour nous permettre de développer un mini-jeu. C'est pourquoi, on a pu se concentrer sur la conception du jeu lui-même.

5.2 Structures

5.2.1 MainScene.js

Ce fichier **Javascript** comporte trois fonctions principales : *constructor()*, *create()* et *update()*. Elles consistent à initialiser et créer la scène du jeu et la mettre à jour à chaque instant t.

5.2.2 singlephase.js

Ce fichier **Javascript** détaille la configuration de la scène(longueur, largeur, couleur du thème).

5.2.3 Snake.html

Ce fichier regroupe du code html et une petite partie du code css (intégrée dans l'html) correspondant à l'affichage du Snake.

5.2.4 Snake.js

Ce fichier **Javascript** est utilisé pour la conception et l'implémentation des règles du jeu. Elle comporte cinq méthodes :

— **constructor(scene)** : Initialise la partie en récupérant les configurations de la scène et affiche les éléments du mini-jeu (score, meilleur score, timer, le nombre de pommes mangées.

— **keydown(event)** : Permet de diriger le serpent grâce au flèche directionnelle (Les boutons du clavier sont spécifiés par leur code ASCII).

— **update(time)** : Permet de mettre à jour la position du serpent et le timer.

— **positionApple()** : Permet de générer à une position aléatoire les différentes pommes ainsi que l'obstacle.

— **move()** : Permet de gérer les évènements liés aux mouvements du serpent, tels que la mise à jour du score, du meilleur score (si nécessaire), du nombre de pommes mangées, ainsi que la relance de la partie.

5.3 Problèmes rencontrés et solutions

Pas de problème particulies rencontré lors du codage du jeu Snake. En effet, les supports et documentations proposés par **Phaser** ont suffit pour répondre à nos besoins.

6 Hébergement

Afin de rendre accessible au public notre application web, nous avons dû héberger notre projet.

6.1 Avantages

Pour héberger notre application web, nous avons décidé d'utiliser **Heroku**. L'avantage d'Heroku est qu'il :

— est **gratuit** : Ce qui est idéal pour notre projet.

— supporte les bases de données **NoSQL** : Ce qui est le cas pour nous avec MongoDB.

— supporte **plusieurs** environnements d'exécution : *Heroku* fonctionne très bien avec *NodeJS*.

6.2 Déploiement

Heroku gère les déploiements d'applications web avec Git directement depuis le terminal grâce à l'Interface de ligne de commande (CLI Heroku). Pour cela, il faut installer Git et la CLI Heroku. Puis, ouvrir un compte sur le site et créer une nouvelle application web.

Les changements nécessaires afin de déployer notre application web ont été d'ajouter une ligne de code dans le fichier **app.js** :

```
1       const port = process.env.PORT
```

Ainsi que d'ajouter à la racine du projet un fichier *Procfile* contenant la commande d'exécution de l'application au démarrage :

```
web: node app.js
```

6.3 Problèmes rencontrés et solutions

Nous avons rencontré un problème lors du push du projet avec le fichier *Procfile*. La solution était de renommer ce fichier, pusher le projet, renommer le fichier à nouveau en *Procfile*, et enfin re-pusher le projet.

7 Conclusion

Notre projet de plateforme de mini-jeux nous a permis d'explorer le monde du web et d'apprendre à utiliser ces différents concepts. Nous avons assimilé comment créer et implémenter des fichiers **HTML** / **CSS** / **JS**. De plus, nous pouvons noter Bootstrap, qui propose différentes alternatives de designs pour un site web.
Les frameworks Express, NodeJS (pour mettre en place un serveur) et Phaser (méthodes pour la création de jeu 2D).
Nous avons aussi pû découvrir comment héberger notre application et la mettre en ligne grâce à Heroku.

Grâce à tout cela, notre projet correspond à nos attentes à savoir : des jeux qui sont accessibles facilement et gratuitement. Notre application étant correctement hébergée, vous pouvez la retrouver en suivant ce lien : **Mini-Jeu** : `mettrelelien`

Bien que notre projet soit finalisé, il peut toujours être sujet à des améliorations telles que
(sur la plateforme actuellement) : l'amélioration de la base de données pour le Sudoku l'amélioration de la sécurité (que se soit la plateforme ou les réponses des grilles du sudoku) et de la responsivité (que ça marche pour n'importe quel navigateur) les sons du Snake ?
(en plus de la plateforme) : ajouter d'autres jeux

8 Bibliographie

Voici la liste des liens qui nous ont permis de développer notre application web :
Express : `https://expressjs.com/`
NodeJS : `https://nodejs.org/fr/about/`
EJS : `https://ejs.co/#docs`
Phaser : `https://phaser.io/`
Score Phaser : `https://stackoverflow.com/questions/37408825/create-a-high-score-in-phaser`
Heroku : `https://devcenter.heroku.com/`
Heroku/Git : `https://devcenter.heroku.com/articles/git`
Socket.io : `https://socket.io/`
Socket.io documentation : `https://socket.io/docs/v4/`
MongoDB : `https://www.mongodb.com/`
Schéma Mongoose : `https://mongoosejs.com/docs/guide.html`
MongoDBAtlas : `https://www.mongodb.com/atlas/database`

Base HTML : https://developer.mozilla.org/fr/docs/Learn/Getting_started_with_the_web/HTML_basics

Base CSS : https://developer.mozilla.org/fr/docs/Web/CSS

Timer en Javascript : https://www.codegrepper.com/code-examples/javascript/how+to+start+a+timer+in+javascript

Convertisseur Timer Javascript : https://stackoverflow.com/questions/1322732/convert-seconds-to-hh-mm-ss-with-javascript

Modal Bootstrap : https://getbootstrap.com/docs/5.0/components/modal/

Barre de navigation Bootstrap : https://getbootstrap.com/docs/5.0/components/navbar/

Card-group Bootstrap : https://getbootstrap.com/docs/5.0/components/card/

Spinner Bootstrap : https://getbootstrap.com/docs/5.0/components/spinners/

JQuery : https://api.jquery.com

Window Location : https://developer.mozilla.org/fr/docs/Web/API/Window/location

Méthode Splice : ://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array/splice