Projet Web - Aaaah !

Rapport de projet

Auteurs :
Fabian VANBALBERGHE
Arnaud KOCH

M1 MEEF NSI

Année universitaire 2025–2026

I. Présentation du projet

Nous avons voulu implémenter lors de ce projet web, une version du jeu Flash « Aaaah ». Le principe du jeu est simple : on déplace notre personnage d'un point A (le spawn) à un point B (le carré de sortie vert). Il y a cependant de la physique, ce qui peut rendre la sortie inaccessible. Pour pouvoir aider les joueurs, un rôle spécial est attribué au joueur avec le plus de points : le guide. Il a la possibilité de créer des plateformes (se supprimant dans le temps) permettant d'accéder à la sortie.

Les joueurs possèdent alors 4 (ou 5 pour le guide) interactions : se déplacer, sauter, crier « AAAH » ce qui va pousser les autres joueurs autour, et enfin la capacité de créer des plateformes pour le guide. Lorsque tous les joueurs sont morts, ou ont terminé la partie, le jeu change de map et de guide.

II. Architecture générale

L'architecture du projet s'appuie sur les technologies suivantes :

II.1. Côté serveur

- Un serveur http Express, permettant d'utiliser TypeScript pour réaliser le traitement des requêtes et la gestion des ressources côté serveur.
- Deux bases de données sqllite afin de stocker les informations utilisateurs et les sessions
- Matter.js, pour le moteur de jeu

II.2. Côté client

- Html
- CSS
- Javascript

II.3. Protocoles

- Protocole HTTP :
  - Requête GET, avec ou sans paramètres, pour la récupération de ressources (pages html, et certaines données formatées en Json, comme la liste des parties)
  - Requêtes POST pour modification de ressource :
    - Authentification (enregistrement de nouveaux utilisateurs)
    - Connexion (enregistrement d'un utilisateur en session)
    - Déconnexion
- Socket.io pour la gestion d'une partie : création, connexion, transmission des données liées à la partie (évènements utilisateurs et données calculées par le serveur)

III. Fonctionnement de l'application

III.1. Inscription

L'inscription se fait via l'écran login.html. Lorsque le formulaire d'inscription est correctement rempli et validé par l'utilisateur, une requête POST est envoyée au serveur. Le serveur vérifie alors si l'utilisateur existe déjà. Si c'est le cas, une erreur est envoyée au client. Sinon, le serveur stocke les informations de l'utilisateur dans la base de données sqllite app.bd. Nous avons fait le choix, de stocker une version « hashée » du mot de passe afin d'assurer un semblant de sécurité. Le mot de passe est cependant envoyé en clair du client au serveur (pas d'https, TLS n'est donc pas utilisé).

La clé primaire de la table des utilisateur, puisqu'il est unique, nous servira alors d'identifiant d'utilisateur dans la session http de notre serveur Express.

Une fois l'enregistrement effectué, le serveur stocke l'identifiant en question dans la session afin de le connecter, avant de répondre à l'utilisateur.

III.2. Connexion

La connexion au site se fait via l'écran login.html. Pour qu'un utilisateur soit considéré comme connecté, il faut qu'un id lui étant associé soit enregistré dans la session côté serveur.

Ainsi, lorsque l'utilisateur a saisi son identifiant et son mot de passe, une requête POST est envoyée au serveur. Celui-ci vérifie l'existence du user dans la BD app.bd, récupère son identifiant unique, et le stocke dans une nouvelle session. Les sessions sont stockées dans une autre base de données sqllite : sessions.sqllite. Une abstraction de la gestion des sessions est effectuée grâce à l'instanciation d'un sessionMiddleWare qui est passé en paramètre du serveur Express. Celui-ci :

- Assure l'enregistrement d'une nouvelle session en BD, créée un cookie contenant l'identifiant de cette session, et la transmet au client.
- Lors de la réception d'une requête (contenant un cookie), charge les informations de la session dans dans la requête. Ce mécanisme nous permet de nous assurer qu'un utilisateur est connecté lors de la réception d'une requête côté serveur.
- Assure la suppression du cookie et de l'enregistrement en BD lorsque que nous supprimons l'utilisateur de la session.

En l'état, nous n'avons pas géré de limite de temps pour nos sessions. C'est donc une amélioration envisagée.

Par ailleurs, un middleware a été mis en place pour la gestion des utilisateurs non connectés : lorsqu'une requête HTTP de type GET est effectuée auprès du serveur sur une autre ressource autre que /login, celui-ci vérifie qu'un userId est bien défini dans la session. Si ce n'est pas le cas, le client est systématiquement redirigé vers /login via la Response.

III.3. Navigation

Le serveur met à disposition quatre page html : login.html, lobby.html, gestion-compte.html, partie.html.
Une fois connecté, l'utilisateur peut directement accéder à ces différentes pages via leur URL (requêtes HTTP GET).

- **login.html** : une fois l'utilisateur connecté, le client assure sa redirection vers le lobby. Dans le cas d'une inscription, le client est connecté, et un bouton généré par le javascript permet de le rediriger vers le lobby.
- **lobby.html** : Au chargement de cet écran, une requête HTTP GET est effectué en AJAX pour récupérer les informations des parties en cours. Cela permet :
  - L'affichage des parties en cours (Nom de partie, map en cours, nombre de joueurs), uniquement s'il en existe.
  - La constitution d'une URL propre à chaque chaque partie, associé à un bouton « Rejoindre ».

  Par ailleurs, le lobby permet de créer une nouvelle partie en saisissant un nom de partie et en sélectionnant une « première » map. En cliquant sur « Créer une partie », une redirection est effectuée vers une URL vers /partie, générée à la volée.
  La liste actuellement affichée n'est pas dynamique (il faut recharger la page). Une amélioration possible serait d'utiliser les une socket pour qu'elle se mette à jour dynamiquement, au fil de la création / suppression de parties, et de la connexion /déconnexion des joueurs sur les parties.
- **partie.html** : La récupération de cette ressource est effectuée à l'aide de paramètres passés en URL. Ce point est détaillé dans la section suivante.
- **gestion-compte.html** : Cet écran, assez sommaire, contient comme principale fonctionnalité un bouton de Déconnexion. Celui-ci effectue une requête POST vers le serveur, qui détruit alors la session. En cas de réussite, le client est notifié et redirige l'utilisateur vers l'écran de Login.

III.4. Accéder à une partie

Une fois connecté, un utilisateur peut se rendre sur l'URL partie.html afin d'accéder à une partie. Cet accès est basé sur l'utilisation d'une **requête GET du protocole HTTP**. Deux paramètres sont utilisés : **idPartie** et **mapId**.

Lors du chargement de la page côté client, le code javascript :
- Récupère ces paramètres
- Se connecte au serveur via l'ouverture d'une websocket socket.io

La présence des deux paramètres conditionne les actions effectuées :
- Si le paramètre **idPartie** est absent (ou null), une redirection est effectuée vers le lobby.
- S'il est présent, les échanges commencent sur la socket via un premier appel (« join »), avec ces deux paramètres. Le serveur prend alors le relai :
  - Si l'idPartie n'existe pas dans la liste des parties et que l'idMap est bien défini, une nouvelle partie est créée et la boucle de jeu est lancée : toutes les interactions client-serveur passent alors par la socket. L'idMap est essentiel : si ce paramètre n'est pas correctement définie, ou qu'il ne correspond à aucune map existante côté serveur, un évènement spécifique est envoyé vers le client pour lui indiquer l'erreur. Le client redirige alors l'utilisateur vers le lobby.
  - Si l'idPartie existe déjà, l'utilisateur est ajouté à la partie existante, et un évènement lui est envoyé pour lui indiquer que la connexion a bien eue lieu. Le moteur de jeu du serveur commence à lui transmettre les données du jeu.

III.5. Moteur de jeu

III.5.1. Système de jeu

Afin d'implémenter ce jeu, nous avons utilisé le moteur physique Matter.js qui nous permet de créer un vrai monde physique, avec de la gravité et des collisions.

Nous avons choisi une approche où le serveur stocke et calcule l'entièreté des informations « physiques » et envoie les données aux clients qui se contentent de les afficher. Le serveur fait donc office de modèle et les clients de vue et de controller. Une structuration en MVC aurait été préférable mais par manque de temps nous le laissons comme ça.

III.5.1.a. Utilisation de socket io

Nous avons fait le choix d'utiliser socket.io (basé sur TCP) pour gérer les communications entre le serveur et le client. L'utilisation de HTTP ne permettrait pas de rendre le jeu synchronisé et fluide pour les joueurs.
L'utilisation de socket nous permet d'avoir un serveur réactif (à 60 Hz) synchronisé, avec la gestion de salons (plusieurs parties peuvent être jouées simultanément) sans perte de données. En bref, elle permet d'avoir un vrai serveur pour le jeu.

III.5.1.b. Création de la partie

Lorsqu'un client se connecte au serveur via Socket.IO et envoie une requête de type join, le serveur doit déterminer s'il faut créer une nouvelle partie ou rejoindre une partie existante.

Chaque partie correspond à un salon (room) identifié par un partieId.

Si le salon demandé n'existe pas encore dans la structure parties, le serveur crée alors une nouvelle partie. À ce moment-là, il initialise le moteur physique Matter.js, génère les objets du monde (plate-formes, murs, zone de sortie, etc.), démarre la boucle de simulation (intervalle à 60 Hz) et enregistre la partie dans le dictionnaire parties. Le joueur à l'origine de la requête est ensuite ajouté à ce nouveau monde.

Si un joueur rejoint un salon déjà construit, nous ne recalculons pas tout, le serveur renvoie directement au nouveau joueur les infos stockées dans le dictionnaire parties en ajoutant le joueur à la partie en cours.

De cette manière, nous ne construisons le salon qu'une seule fois et que lorsqu'il y a des joueurs qui le demandent. Lorsqu'il n'y a plus de joueurs dans le salon, nous supprimons le salon des dictionnaires et stoppons l'interval.

De plus, si un client se déconnecte, nous le retirons proprement de la partie en cours.

III.5.1.c. Interaction joueurs

Les clients envoient des requêtes de déplacement au serveur qui calcule physiquement ce déplacement dans le moteur physique. Le serveur renvoie alors à tous les clients de la room les changements opérés.

Ainsi, le serveur est le seul responsable de la simulation physique, ce qui permet de garder le jeu cohérent pour chaque joueur et empêche la triche côté client.

Cette architecture permet également d'ajouter de nouveaux types de contrôles ou d'interfaces (clavier, mobile, IA) sans modifier la logique du serveur, puisque seules les entrées changent, pas le modèle physique.

- Déplacements : le serveur va modifier la vélocité en x des joueurs lorsqu'il reçoit une demande de déplacement. Pour le saut, on utilise une fonction vérifiant qu'il y a bien du sol en dessous du joueur (sinon le joueur pourrait sauter à l'infini) puis on modifie sa vélocité en y. À noter que la vitesse est modifiée en fonction de la taille du joueur (définie dans la map). Au minimum il ira 0.5 x moins vite que la vitesse de base (si le joueur est très gros) et 2 x plus vite si le joueur est petit. Nous n'avons pas encore exploité pleinement cette mécanique de gameplay mais l'implémenter nous semblait pertinent.
- Ah : nous calculons une distance autour du joueur ayant appuyé sur « AH » puis en parcourant tous les joueurs du salon, nous vérifions s'il est dans le centre de détection du joueur et nous le poussons en fonction d'une variable PUSH_FORCE et de sa distance. Nous avons aussi ajouté un timing de cooldown limitant l'utilisation du « AH ». Pour ce faire, nous stockons dans un dictionnaire l'utilisateur ayant utilisé le « AH » et le moment auquel il a appuyé + la variable de temps de cooldown défini. Il suffit ensuite de vérifier que now est bien supérieur à la valeur enregistrée dans le dico pour pouvoir réutiliser le « AH ».
- Rôle spécial « le guide » : ce rôle permet au joueur ayant le plus de points de créer des plateformes pour aider à atteindre la sortie. Nous stockons une liste de positions lorsque le joueur bouge sa souris avec le clic enfoncé, que nous envoyons au serveur lors du relâchement du clic. Le serveur construit alors plein de petits rectangles côte à côte dessinant finalement un gros trait. Nous avons ajouté la suppression automatique de ces plateformes au bout de 5 secondes. Dans la même logique que pour le cooldown du « AH », nous ajoutons le timer auquel est apparue la plateforme. Il suffit ensuite de parcourir toutes les plateformes créées et de supprimer celle avec le timer dépassé.

III.5.1.d. Réinitialisation des parties

Lorsque tous les joueurs ont terminé (ou sont morts), la room réinitialise la partie courante, change de map et redéfinit un nouveau guide (voir dans la partie d'après). Afin d'avoir un monde physique cohérent, nous devons alors supprimer l'entièreté des bodies créés dans la map précédente, charger la map suivante et reconstruire les joueurs en fonction de la position de départ indiquée dans la map.

Nous avons ajouter aussi le fait que lorsqu'un joueur tombe en dessous d'une coordonnée y défini par KILL_Y, il se fait tuer par le jeu pour ne pas calculer en permanence et pouvoir gérer de façon cohérente le reset d'une partie.

III.5.2. Système de classement

Nous avons ajouté un système de classement.

Lorsque le premier joueur termine, il gagne 20 points, 16 pour le second, 13 pour le troisième puis 10 pour les suivants. Mourir n'accorde aucun point. Les points sont stockés dans un dictionnaire avec comme clé l'userId et comme valeur, ses points actuels. Ceci permet d'accéder rapidement aux valeurs sans collisions (l'userId est unique, c'est la clé auto-incrémentée de la table users).

Le joueur ayant le plus de points devient guide et ses points redescendront à 0 la partie suivante.

III.5.3. Vue & Controllers

III.5.3.a. Structure globale

La vue et les controllers sont mélangés dans notre js côté client. Nous avons toute une partie où sont gérés les inputs côté clavier. Dans cette partie, nous récupérons les inputs de l'utilisateur et nous envoyons la demande au serveur comme expliqué précédemment.

Nous stockons certaines informations nécessaires telles que le dictionnaire des joueurs avec leurs positions, le dictionnaire de plateformes dessinées, etc.

Dans une boucle sobrement intitulée loop, nous affichons toutes les informations envoyées par le serveur telles que la position des joueurs actuels, les plateformes, etc. Nous avons rajouté des pseudos au-dessus des joueurs, écrits en jaune pour le guide, et un petit feedback pour le dessin du guide (trait blanc avant de relâcher). Cette boucle est appelée par le navigateur avec requestAnimationFrame(loop) en fonction du GPU (souvent à 60Hz).

Nous gérons aussi toutes les émissions du serveur vers le client, affichant des erreurs s'il y en a, et mettant à jour le jeu si besoin.

Version mobile :

Nous avons rajouté aussi des boutons initialement invisibles, s'affichant uniquement si le client possède un écran tactile. Ces boutons sont des redirections des inputs claviers adaptés pour les mobiles.

Afin de garder un affichage cohérent peu importe notre machine, nous avons ajouté côté client un petit calcul pour redimensionner via le CSS, le canva (comme ça on ne modifie pas le jeu côté serveur). Cela a demandé une petite adaptation aussi pour dessiner en tant que guide si l'écran est redimensionné, chose que nous avons faite en récupérant la taille du canva actuel et en adaptant en fonction.