

CAS-CONTACT - RAPPORT DE PROJET

Groupe JL

www.cascontact.xyz

Description technique et justification des choix	2
Back-end	2
Communications client-serveur	2
Lobbies et salles d'attentes	2
Maître du jeu	6
Sécurité	8
Authentification	9
Base de données	10
Front-end	11
Page d'authentification	12
Page du leaderboard	13
Page des parties	13
Page de jeu	15
Sous-composants de la page de jeu	18
Version mobile	20
Communication vocale entre les joueurs	21
Fonctionnement en mode projet	21
Réunions	21
Gitlab	21
Documents partagés	22
Gantt	22
Clockify	22
Communication	23
Rapports individuels	24
Paul LABAYE (chef de projet)	24
Pierre LAFOSSE	25
Qinyue LIU	26
Shufeng JIANG	28
Clément LIGNEUL	29
Karim KHATER	31
Lafdhal AHMED	32
Paco KLEITZ	33

Description technique et justification des choix

Le projet se base sur une structure client/serveur avec un client web. Le serveur est développé en python, et nous avons utilisé Vue.js pour le client. La communication entre les 2 est assurée à l'aide de websockets.

Back-end

Le serveur est développé en python. Nous avons opté pour cette option car nous avons tous au moins un peu d'expérience avec ce langage. Pour notre utilisation, il présente beaucoup d'avantages: c'est un langage très populaire, d'où découle une abondance de documentation accessible facilement sur internet. Il est aussi réputé pour ses nombreuses bibliothèques facilitant le développement déjà relativement rapide en comparaison avec d'autres langages dû à son approche haut niveau limitant les sources de bugs et offrant de nombreuses fonctionnalités très pratiques dans ses fonctions natives.

Communications client-serveur

Nous avons utilisé la bibliothèque socketio pour faire communiquer le client et le serveur. Cette bibliothèque introduit le paradigme de programmation événementielle en offrant la possibilité de facilement gérer les événements en créant des "handlers" ainsi que d'en émettre. Les événements peuvent avoir des données, raison pour laquelle nous les avons généralement appelées des messages au cours du développement. Ces messages ou événements sont envoyés par le serveur vers les clients pour les informer de l'état de la partie et des clients vers le serveur pour l'informer des décisions prises par les joueurs. Pour des raisons de sécurité, le serveur vérifie la cohérence de tous les messages reçus avant de mettre l'état de la partie à jour.

L'utilisation de cette bibliothèque offre de nombreux avantages: elle existe pour python et pour javascript et est compatible entre les deux langages tout en étant facile d'apprentissage et d'utilisation. Beaucoup de ses fonctionnalités se sont avérées très utiles pour le développement. Premièrement, la présence de « rooms » a joué un rôle majeur dans le développement du système de parties et de salles d'attente. Ces "rooms" sont à voir comme un groupe de clients connectées au serveur, facilitant grandement l'envoi de messages à un sous-ensemble des clients connectés. Lorsqu'un joueur s'authentifie sur le site, il est envoyé dans la room principale, depuis laquelle il a accès à toutes les parties publiques. Ensuite, lorsqu'il rejoint une partie (publique ou privée) il est envoyé dans une room spéciale pour cette partie, mise en place lorsque la partie a été créée. Nous nous servons également des rooms pour différencier les joueurs d'une même partie, pour par exemple permettre aux joueurs éliminés de ne communiquer qu'entre eux grâce au chat. Avant d'être connecté au serveur via une connexion socket après l'authentification, le client envoie des requêtes http au serveur. Pour cette partie, nous avons utilisé fastapi. C'est une api

Lobbies et salles d'attentes

Généralités et fonctionnement

Le système de parties permet de créer et de gérer plusieurs "rooms" qui s'exécutent indépendamment les unes des autres. Chaque room contient une partie comprenant un nombre de joueurs défini à l'avance (entre 5 et 15). Dans le menu principal, un joueur connecté peut choisir de créer une nouvelle partie ou d'en rejoindre une qui n'a pas son nombre de participants au complet. Pour rejoindre une partie il peut en choisir une proposée par le serveur via une liste. Il

peut aussi se connecter à une partie “privée” nécessitant un code. Ce code est saisi dans un champ approprié sur l’écran de sélection de la partie. À la validation, le nouveau joueur est directement ajouté et connecté à la partie. Lorsqu’un joueur se connecte ou n’est dans aucune partie, il est dans la room générale appelée “General lobby”.

Le système de parties est représenté sous la forme d’un dictionnaire associant le nom d’une partie avec une liste d’éléments. Dans le code il s’agit de la variable globale “game_tab”. La structure est de la forme:

```
Nom_room : [Game, nb_actuel, nb_max, code, nom_admin, typing_list, ban_list]
```

- Game : objet de type Game, il contient toutes les données relatives à la partie en cours (liste des joueurs, phase en cours, etc)
- nb_actuel : c’est le nombre de joueurs actuellement dans la partie
- nb_max : c’est la capacité maximum de joueurs pour cette partie
- code : vaut None si la partie est publique. Sinon c’est le code qui permet de rejoindre la partie privée
- nom_admin : nom de l’administrateur actuel de la partie
- typing_list : la liste des joueurs en train d’écrire (propre à chaque room)
- ban_list : la liste des personnes bannies de cette partie
- Nom_room : nom de la partie (constitué du nom du créateur concaténé à “_room”)

Contexte et messages échangés entre back-end et front-end

Dans le code, l’évènement `join_room` permet de rejoindre une partie mais aussi d’en créer une. Tout dépend des paramètres du message.

Syntaxe du message pour créer/rejoindre une partie

```
join_room : {name, room, game, public, new, max, p_code, game_room}
```

Certains paramètres n’apparaissent que dans certains cas.

Description

- name: nom du joueur
- room: room actuelle du joueur
- game[0/1]: 0 -> on cherche à rejoindre “General lobby”, 1 -> on cherche à rejoindre une partie
- public[0/1]: 0 -> on veut créer/rejoindre une partie privée, 1 -> on veut créer/rejoindre une partie publique
- new[0/1]: 0 -> on veut rejoindre une partie ou “General lobby”, 1 -> on veut créer une nouvelle partie
- max (si création) : nombre max de joueurs dans la partie créée
- game_room (si rejoindre une partie publique) : nom de la partie à rejoindre

- p_code (si rejoindre une partie privée) : code privé permettant de rejoindre une partie (privée)

Création d'une partie

- L'utilisateur saisit un nombre de joueurs max et clique sur créer une partie publique
- Demande de création d'une nouvelle partie publique par le client au serveur
- Si la création d'une nouvelle partie est possible, le serveur le notifie au client et transfère le client qui a fait la requête dans une room avec les droits "admin de partie".
- Le serveur ajoute la partie à la liste de parties publiques si elle l'est.

Rejoindre une partie

- Le client envoie au serveur la demande pour rejoindre
- Si la réponse est positive, le serveur ajoute le client à la partie

Réception de la liste des parties disponibles

Lorsqu'un joueur crée ou rejoint une partie, le serveur va mettre à jour le tableau des parties et l'envoyer à l'ensemble des clients. Ce tableau est aussi envoyé au client lors de sa connexion au serveur. On passe par le message "update_table".

Système de bannissements

- Un joueur ne peut bannir un autre joueur de la partie que s'il est admin de cette partie.
- Le joueur banni est ajouté à la liste des bannis de la partie et ne peut plus la rejoindre.
- Le bannissement est impossible dans la room "General lobby"
- Le bannissement peut se faire avant ou après le début de la partie

On utilise le message "ban".

Si le joueur à bannir est valide (présent et dans la game), ce dernier quitte la room dans laquelle il se trouve et est redirigé dans "General lobby". Il est évidemment logique qu'un joueur ne puisse s'auto bannir.

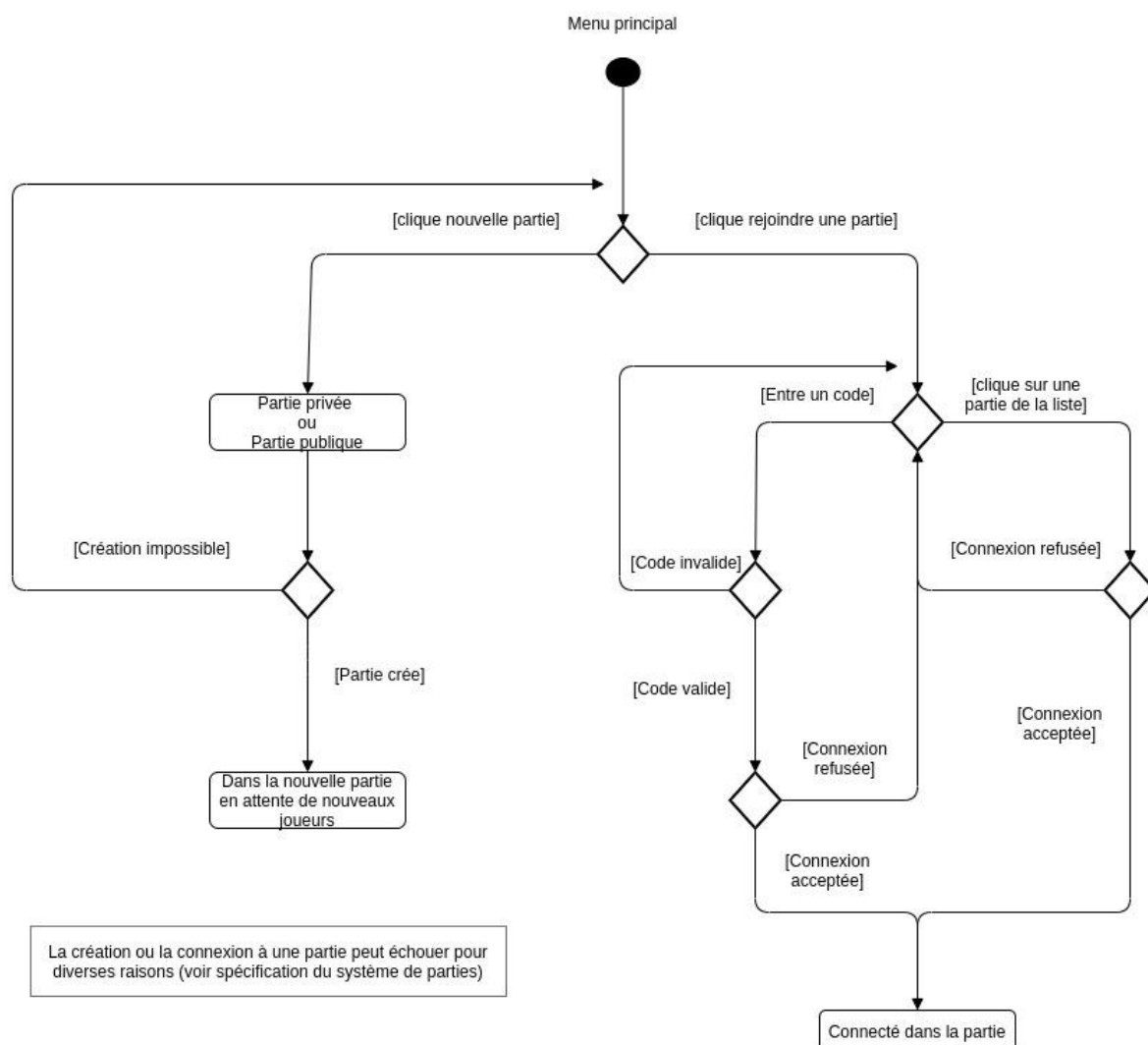
Début d'une partie

On entend par début d'une partie le commencement de l'application de l'algorithme du maître du jeu. Une partie peut commencer lorsqu'il y a au moins 5 joueurs dans la même room. L'admin dispose d'un bouton lancer, qui déclenche le début de la partie. C'est là que le client envoie "start_game".

Côté serveur, on assiste au fonctionnement suivant :

- Réception et vérification de la validité du message via la fonction "start_game". Tenter de commencer une partie en étant dans General lobby ne produit aucun effet (le bouton n'étant accessible que dans une partie).
- Si le message est validé, alors on fait appel à la fonction "ready_to_start". Cette fonction permet de démarrer la partie si la room cherchée est bien dans le tableau de parties. En cas d'erreur, un code est renvoyé et est interprété par la fonction "start_game".

Diagramme de création/connexion à une partie



Maître du jeu

Généralités et fonctionnement

Le maître du jeu est le module qui contient toute la logique d'une partie de jeu. Une partie consiste en une suite de phases, dont certaines ne se déclenchent que sous certaines conditions. Tous les joueurs doivent endosser un rôle choisi aléatoirement par l'algorithme. En fonction de la phase du jeu et du rôle d'un joueur, celui-ci aura différentes actions à accomplir.

Les rôles

Les rôles qu'un joueur peut prendre sont les suivants:

- Citoyen: C'est le rôle le plus basique de notre jeu, les citoyens ne peuvent rien faire de particulier
- Infecté: Ces joueurs sont les opposants de tous les autres. Une phase leur est dédiée durant laquelle ils peuvent désigner un joueur à éliminer.
- Didier Ratoul: Il s'agit de notre adaptation du personnage de la sorcière dans le jeu du loup-garou original. Par partie, ce joueur pourra une fois éliminer un joueur de son choix et une fois sauver le joueur désigné par les infectés de l'élimination.
- Anti-masque: Ce joueur peut à sa mort désigner un joueur à éliminer avec lui.
- Infirmière: Elle peut découvrir le rôle d'un autre joueur à chaque tour

A cela viennent s'ajouter des états qui altèrent ce qu'un joueur peut faire:

- Mort: Le joueur a été éliminé de la partie. Il peut tout de même continuer à la regarder mais ne peut plus interagir avec le jeu
- Le ministre: Lors du vote du joueur, le vote de ce joueur compte double. A sa mort, il choisit son successeur

Les phases

Les phases s'exécutent selon l'ordre suivant :

[nurse] → night_vote → [cd] → [am_death_night] → [election] → [minister_death] → day_vote → [minister_death] → [am_death_day] → [minister_death] → début

Les phases en crochets ne se déclenchent que sous certaines conditions comme détaillé plus bas.

- day_vote: C'est le moment où tout le monde vote pour éliminer quelqu'un. Cette phase se déclenche à tous les tours.
- night_vote: C'est le moment où les infectés votent pour transmettre le covid à quelqu'un. Cette phase se déclenche également à chaque tour.
- election: Cette phase ne se déclenche qu'au premier tour de la partie. C'est à ce moment que les joueurs doivent élire leur ministre.
- minister_death: Cette phase se déclenche que si le ministre vient de mourir. Il doit alors choisir son descendant.
- am_death_day: Durant cette phase l'anti-masque choisit qui il veut éliminer. Elle ne se déclenche que si l'anti-masque a été choisi pendant la phase du day_vote.
- am_death_night: : Durant cette phase l'anti-masque choisit qui il veut éliminer. Elle ne se déclenche que si l'anti-masque a été choisi pendant la phase du night_vote.

- cd: C'est le tour de Didier Ratoul, il doit choisir s'il utilise ou pas ses potions. Le serveur montre au cd qui est la victime des infectés avant qu'il puisse faire son choix. Cette phase ne se déclenche que si Didier Ratoul est en vie et qu'il n'a pas encore épuisé ses actions.
- nurse: Cette phase ne se déclenche que si l'infirmière est en vie, cette dernière doit choisir le joueur dont elle veut découvrir le rôle.

Les phases ont une durée prédéterminée. Cependant, on peut écourter une phase. Les joueurs dont c'est le tour, une fois leur choix effectué, peuvent confirmer. À partir de là, ils ne peuvent plus modifier leur vote. Si tous les joueurs concernés confirment, la phase se termine.

Fin de la partie

- Soit tous les joueurs sont (soit morts, soit infectés)
- Soit aucun joueur en vie n'est infecté
- Soit seulement 2 joueurs sont présents dans la partie

Règles du chat

Pendant la partie, les joueurs ne peuvent pas communiquer entre eux comme ils veulent, certaines règles s'appliquent. Par exemple, les citoyens ne doivent pas pouvoir communiquer entre eux la nuit. Les messages du chat sont donc traités selon les règles suivantes.

La nuit, les infectés peuvent parler entre eux, mais les autres ne voient pas. Pour les autres phases, tout le monde peut communiquer. Les morts, peu importe leur rôle et la phase en cours voient tous les messages, mais leurs messages ne sont vus que par les autres morts.

Messages propres au maître du jeu

Pour que les clients puissent donner des informations au serveur et pour que le serveur puisse donner des informations sur l'état du jeu aux clients nous avons défini une liste de messages à la signification bien définie. Dans l'optique de ne pas se perdre, nous avons décidé d'utiliser un nombre minimal de messages différents.

Message "chat"

Ce message est utilisé pour écrire quelque chose dans le chat. Le serveur renvoie ensuite le message à ces destinataires qui varient en fonction du rôle du joueur et de la phase du jeu. Par exemple, les morts ne peuvent écrire qu'aux morts, pendant la phase `night_vote`, seuls les infectés peuvent utiliser le chat et ainsi de suite.

Message "designate"

Ce message est utilisé à chaque fois qu'il faut désigner un joueur. Les clients envoient des messages "designate" pour indiquer pour qui ils votent. Le serveur renvoie des "designate" en retour pour indiquer aux autres joueurs qui vote pour qui.

Message "commit"

Ce message est utilisé pour que les clients disent au serveur qu'ils valident leur choix. Si tous les joueurs ayant à faire un choix (tous les joueurs en vie pendant le vote du jour et celle du vote du maire, tous les infectés pendant le vote de la nuit, l'antimasque pendant les phases `am_death`, Didier Ratoul pendant la phase `cd`, l'infirmière pendant la phase `nurse`, le ministre pendant la phase `minister_death`), le maître du jeu pourra passer à la suite sans attendre l'intégralité du timer. Une fois qu'un joueur a validé son choix, il ne peut plus revenir en arrière.

Message "send_role"

Ce message est utilisé au début de la partie par le serveur pour informer du rôle qui lui a été attribué.

Message "start_phase"

Ce message sert au serveur à annoncer aux clients le début d'une nouvelle phase de jeu.

Message "show_nurse"

Ce message est utilisé par le serveur pour indiquer à l'infirmière le rôle du joueur qu'elle a désigné lors de la phase nurse.

Message "cd_choice"

Ce message est utilisé par le client dont le rôle est Didier Ratoul pour informer le serveur des décisions qu'il prend durant la phase cd qui lui est dédiée.

Message "bye"

Ce message est utilisé quand un client quitte une partie. Quand il est envoyé vers le serveur, c'est pour lui indiquer qu'il se déconnecte. Le serveur renvoie ensuite l'information aux autres clients.

Message "show_casualties"

Le serveur utilise ce message pour informer les joueurs des éliminations au cours de la partie.

Sécurité

Tout notre trafic passe par https. Nous utilisons le protocole TLS1.3, qui est plus performant et sécurisé que ses prédécesseurs, avec les chiffrements recommandés par Mozilla. Nos certificats SSL proviennent de Let's Encrypt qui est une autorité reconnue.

Notre framework web backend refuse automatiquement les requêtes dont les champs ne sont pas du type attendu. Nous vérifions également la validité de la valeur de ces champs (ex: pseudo alphanumérique) afin de ne pas avoir de surprises par la suite.

Le serveur backend refuse également les requêtes venant de sites tiers, nous acceptons uniquement les requêtes depuis nos noms de domaines (CORS).

La base de donnée tourne dans un container docker, elle n'est donc pas accessible depuis l'extérieur, seul les autres container docker locaux peuvent communiquer avec.

Les injections SQL ne sont également pas possible, notre ORM (SQLAlchemy) vérifiant automatiquement les chaînes de caractères avant de faire les requêtes à la base de données.

Les mots de passe des utilisateurs sont cryptés via l'algorithme Bcrypt et stockés dans la base de données. Lorsque l'utilisateur s'identifie, son mot de passe est envoyé via https à notre serveur, qui le crypte et compare le hash obtenu avec celui dans l'entrée BDD de l'utilisateur cherchant à se connecter.

Nous aurions voulu stocker le token d'authentification via un Cookie (httpOnly) qui n'est pas accessible via javascript du côté client. Uniquement notre serveur aurait pu le récupérer, pour vérifier l'identité de l'utilisateur, via requête https (secureCookie).

Malheureusement, la personne chargée de l'authentification client nous ayant quitté, nous n'avons pas eu le temps d'implémenter ça et avons donc simplement utilisé le localStorage du navigateur pour le stockage des tokens. Nous sommes conscients que cette méthode n'est pas optimale, elle est cependant utilisée par beaucoup de sites malgré le potentiel risque de sécurité (attaque XSS).

Le token n'est valable que 24h, ce qui nous semble un bon compromis entre usabilité et sécurité, l'utilisateur n'a à se reconnecter que 1 fois par jour et si le token est perdu/volé il sera inutilisable sous 24h.

Le routing du frontend ne laisse pas l'utilisateur accéder à une page autre que celle de connexion/inscription sans être authentifié. Lors de la connexion, le jeton d'authentification de l'utilisateur est passé au serveur, qui associe son id socket au nom d'utilisateur déchiffré depuis le jeton. Il est donc impossible pour un utilisateur d'usurper de l'identité d'un autre utilisateur sans avoir son jeton. Après 24h, l'utilisateur devra se reconnecter, cela permet d'éviter d'avoir des jetons perdus qui n'expirent jamais.

Aussi, nous avons pris des mesures contre les tricheurs. Dans le MJ, tous les messages reçus sont vérifiés avant d'être traités en fonction du contexte de la partie (rôle et phase). Si le message est jugé invalide, il est rejeté.

Authentification

L'authentification du client se fait via un Json Web Token fourni par le backend. L'utilisateur s'inscrit à notre site, ce qui crée une entrée dans la BDD contenant son email et mot de passe cryptés qui seront utilisés pour s'authentifier par la suite. Lorsque l'utilisateur veut se connecter, il envoie son e-mail et mot de passe via HTTPS au backend, qui crypte le mot de passe et le compare avec celui de la base de données pour l'utilisateur lié à l'email. Si ils sont égaux, le backend retourne un token d'authentification au client qui contient son nom d'utilisateur crypté. L'utilisateur peut ensuite envoyer ce token pour prouver son identité, seul le backend peut en voir le contenu, en le décryptant avec la clé que lui seul détient. Ce token n'est que valide 24h, il ne sera ensuite plus considéré comme valide par le backend. Le token est envoyé au backend juste après l'initialisation de la connexion socket. Il est envoyé par message socket, ce qui permet de récupérer à la fois le token et l'id de la connexion socket avec le client. On ajoute ensuite le username décodé depuis le token couplé au sid ayant envoyé le token à un registre connected. Le programme peut ensuite vérifier le username d'un client à partir de son id socket, ce qui permet d'identifier l'identité de l'expéditeur des messages socket.

Nous avons fait le choix de l'utilisation d'un token pour le côté

Base de données

La base de données est une BDD PostgreSQL qui tourne dans un container docker. Elle n'est accessible que par les autres services docker. Ses données sont partagées avec le serveur (la machine) via un volume partagé pour qu'elles persistent même après la mise hors service du container. On n'utilise pas de SQL directement, toutes les communications avec la BDD se font via un ORM qui abstrait la couche SQL en code python. Pour ceci nous avons choisi sqlalchemy. Le service propose une bonne documentation et son utilisation est recommandée avec fastapi. Le mot de passe de la BDD est stocké en variable d'environnement sur la machine faisant tourner docker.

La BDD sert uniquement à créer, connecter et obtenir le score des utilisateurs.

Toutes les vérifications des données sont faites par le serveur backend qui s'assure que les données entrées dans la BDD sont correctes, les injections SQL sont gérées par l'ORM qui vérifie les strings.

Les mots de passe sont stockés cryptés via l'algorithme Bcrypt. Comme nous avons une base de données très simpliste, PostgreSQL est largement suffisant.

Notre unique table User comporte les champs suivants:

id	int
username	string
email	string
score	int
hashed_password	string

Front-end

Le choix des technologies pour le front-end a été une étape difficile dans le projet. Dans un premier temps, nous avons décidé d'utiliser Angular. Nous avons fait ce choix pour pouvoir profiter d'une solution « complète ». Nous ne voulions pas avoir à utiliser plusieurs outils pour le client, et nos recherches nous ont naturellement mené vers Angular. Cependant, n'ayant pas une grande expérience dans le développement web, notre équipe front-end a eu beaucoup de mal à commencer à développer avec Angular. Et pour cette raison, le framework est difficile à s'approprier si l'on est débutant en développement web. Par exemple, le langage utilisé pour coder avec Angular est le Typescript. Bien qu'étant un simple superset du javascript, ce détail, entre autres, a perturbé les esprits et nous a empêché de nous lancer. Une des raisons majeures était aussi la structure complexe des projets Angular. Nous avons donc décidé de changer de framework pour Vue.js, qui est beaucoup plus léger et qui requiert une moins grande expérience en développement web pour se lancer. Ce changement a eu lieu dans la semaine du 16, nous avons donc pris un léger retard sur le développement du client par rapport à celui du serveur. Après cela, le développement a été plutôt rapide, car nous avons déjà un plan précis du client, et savions ce que nous avions à faire.

Au début du projet, chaque fonctionnalité était développée de manière isolée, avec un fichier javascript, un fichier html et un fichier css. Par la suite, nous avons décidé d'utiliser une structure web-pack, qui faciliterait grandement l'intégration. Cela consiste en un projet tenant dans un seul répertoire, avec une structure de fichiers se focalisant sur les composants vue. Un fichier représente un composant, avec, dans le même fichier, un template html, un script js, et une feuille de style css. Bien qu'un peu déroutant pour l'équipe front-end au début, ce mode de développement a joué un rôle essentiel dans l'intégration et les tests, puisque nous avons réussi à effectuer le changement à temps avant de devoir tester et intégrer les composants entre eux.

Pour la connexion avec le back-end, nous avons utilisé des web-sockets. En effet, même si le jeu est en tour par tour, nous avons besoin de communiquer les messages en temps réel. Par exemple, lors d'un vote, on veut pouvoir afficher qui vote pour qui, et tout ça avant la fin du vote, pour que les joueurs soient au courant de ce qu'il se passe. Les websockets sont donc la technologie qui s'est présentée à nous de façon naturelle. Nous avons choisi d'utiliser socketio. Pour le client, un plugin vue-socketio est disponible, fonctionnel et facile d'utilisation. Cela nous a donc grandement facilité la tâche.

Le client se divise en 4 composants majeurs.

Page d'authentification

The image displays two screenshots of the CAS-CONTACT authentication interface, which has a dark blue background and white text.

Top Screenshot (Registration Page):

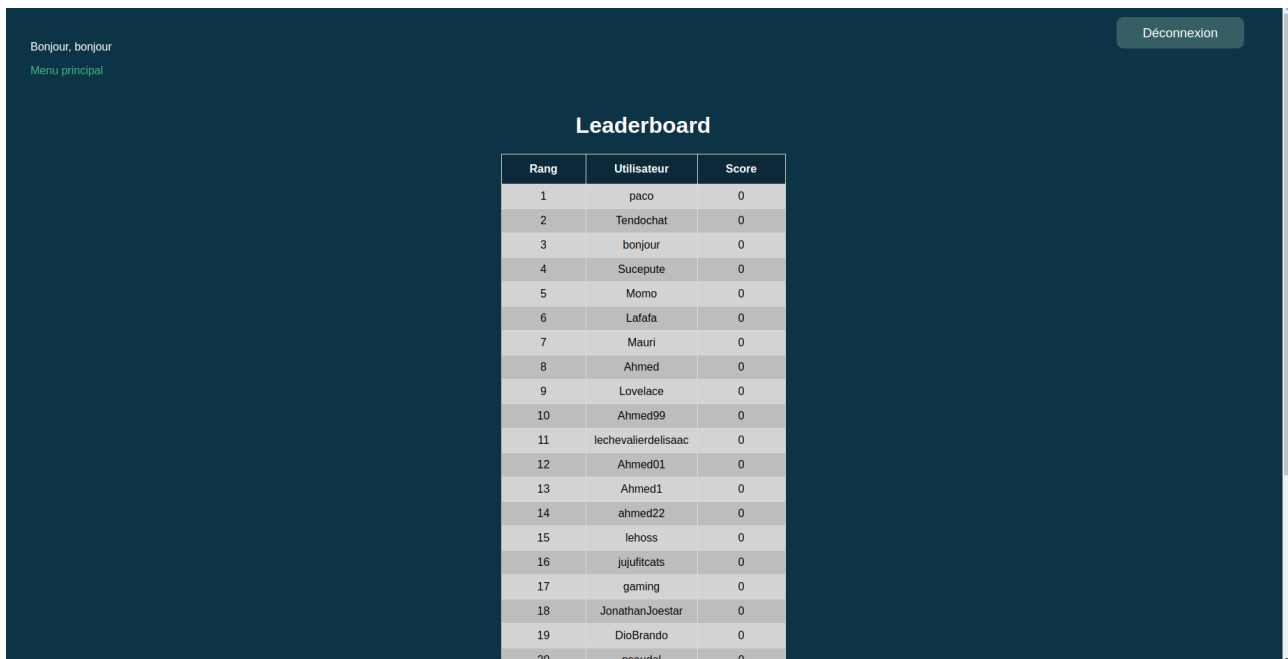
- CAS-CONTACT** (Title)
- [Inscription](#) | Connexion (Link)
- Pseudo (Label)
- (Input field)
- E-mail (Label)
- (Input field)
- Mot de passe (Label)
- (Input field)
- Confirmation (Label)
- (Input field)
- (Button)

Bottom Screenshot (Login Page):

- CAS-CONTACT** (Title)
- Inscription | [Connexion](#) (Link)
- E-mail (Label)
- (Input field)
- Mot de passe (Label)
- (Input field)
- (Button)

Pour l'authentification, le client n'est pas encore connecté avec une connexion socket. Les fonctions appelées lorsqu'on clique sur les boutons Inscription / Connexion utilisent axios pour poster des requêtes html vers le back-end, avec les informations d'authentification entrées par l'utilisateur en paramètres.

Page du leaderboard



The screenshot shows a dark-themed web interface. In the top left, there is a greeting 'Bonjour, bonjour' and a link 'Menu principal'. In the top right, there is a 'Déconnexion' button. The main content area is titled 'Leaderboard' and contains a table with 20 rows. Each row has three columns: 'Rang' (Rank), 'Utilisateur' (User), and 'Score' (Score). The scores are all 0.

Rang	Utilisateur	Score
1	paco	0
2	Tendochat	0
3	bonjour	0
4	Sucepute	0
5	Momo	0
6	Lafafa	0
7	Mauri	0
8	Ahmed	0
9	Lovelace	0
10	Ahmed99	0
11	lechevalierdelisaac	0
12	Ahmed01	0
13	Ahmed1	0
14	ahmed22	0
15	lehoss	0
16	jujuficats	0
17	gaming	0
18	JonathanJoestar	0
19	DioBrando	0
20	pseudal	0

Lors du chargement, le composant envoie une requête http au serveur, qui lui répond avec les 30 entrées ayant le plus haut score dans la base de données. On peut y accéder depuis la page des parties.

Page des parties



The screenshot shows a dark-themed web interface. In the top left, there is a greeting 'Bonjour, bonjour' and a link 'Leaderboard'. In the top right, there is a 'Déconnexion' button. The main content area has two sections: 'Rejoindre une partie privée:' and 'Rejoindre une partie publique:'. The private section has a text input field for 'Code de la partie' and a 'REJOINDRE' button. The public section has a table with two rows, each showing a username and a score. At the bottom, there is a 'CRÉER UNE PARTIE' button.

Rejoindre une partie privée:

Code de la partie

REJOINDRE

Rejoindre une partie publique:

tiboinsh4pe	3/5
Kasper	2/5

CRÉER UNE PARTIE

Il s'agit d'un menu, où l'utilisateur peut voir la liste des parties publiques qu'il peut rejoindre en cliquant dessus. La page permet également de rejoindre une partie privée, en renseignant le

code de la partie, qui est généré lors de sa création. Ces dernières ne sont pas affichées dans la liste des parties, car elles sont privées.

Rejoindre une partie publique:

p1 *1/5*

Type de partie : Publique

Le nombre du participant: - 5 +

Créer

CRÉER UNE PARTIE

Également, cette page offre la possibilité de créer une partie. Avant de la créer, on y choisit le type (publique, privée) et le nombre de joueurs maximum.

Fonctionnement

Handlers

`update_table`: met à jour la table des parties. Elle est stockée dans une variable `tables`.
`auth_ok`: met à jour la variable `player_name`.
`change_room`: met à jour la variable `room`.
`error`: affiche le message d'erreur

Lors du chargement du composant, il émet le message suivant :

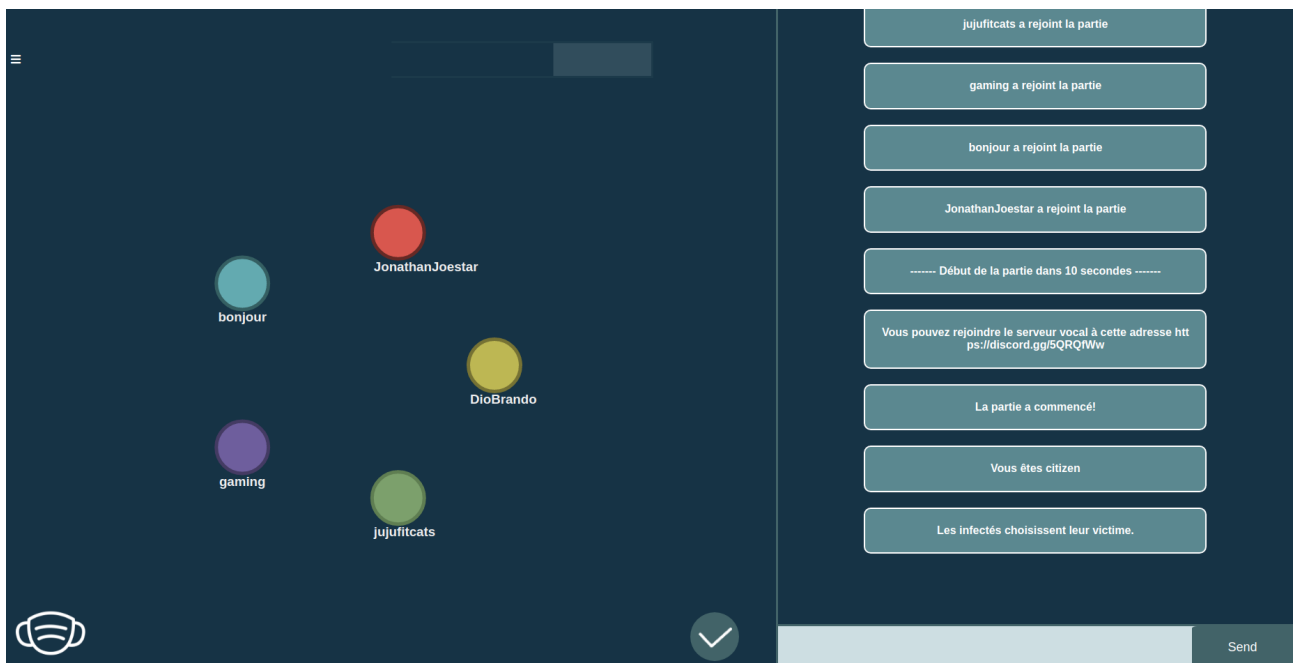
"auth", {token: localStorage.getItem("jwtoken")}. Le back-end vérifie le token et envoie les messages `auth_ok` et `update_table`.

Lorsque l'utilisateur clique sur créer une partie et choisit ses paramètres, on appelle la fonction `create_lobby`. Lorsqu'il clique sur une partie publique, où qu'il entre le code d'une partie privée et qu'il valide, on appelle la fonction `joingame`. Les deux fonctions émettent un message "join_room" et changent le composant à afficher à l'aide du routing.

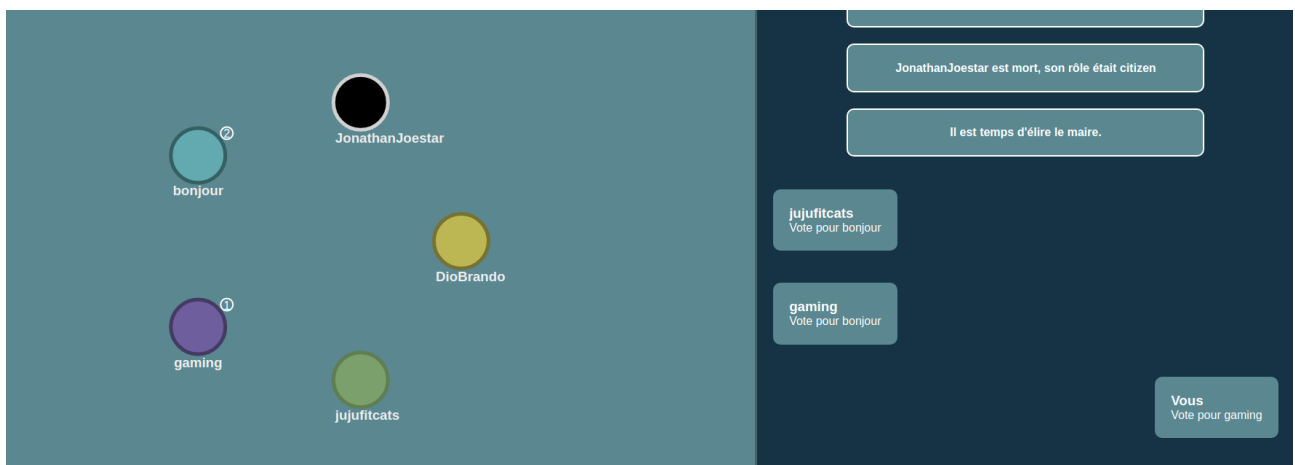
Lorsque l'utilisateur clique sur le bouton déconnexion, on enlève le token du localStorage, on revient à la page de connexion, et on ferme la connexion socket.

La liste des parties publiques fait usage d'un composant externe, `Table`. Les parties sont affichées dynamiquement selon la liste contenue dans `tables`. Le composant `Table` écoute un événement 'clique', et quand cet événement lui arrive, il va signaler à son père `JoinPartie.vue` avec une variable "p_room" qui stocke la valeur de `room` pour rejoindre la partie publique correctement.

Page de jeu



Page de jeu de nuit



Affichage des votes

C'est la page qui est affichée lorsqu'on rejoint une partie. Cependant, cette page ne sert pas uniquement à jouer. Elle sert aussi de salle d'attente. En effet, tant que le créateur de la partie n'a pas lancé le jeu, ou que le nombre maximum de joueurs n'est pas atteint, tous les joueurs d'une partie sont réunis dans une salle d'attente, et la page de jeu s'affiche sur leur écran. Ils peuvent d'ores et déjà utiliser le chat, avec les autres joueurs de la même partie. Dans cette page, on peut accéder à un menu, qui permet de gérer différents paramètres. Avant de lancer la partie, le créateur peut y modifier les rôles qui y seront présents, et le nombre d'infectés parmi les joueurs. Il peut également exclure les joueurs. C'est aussi dans ce menu qu'on retrouve le bouton pour quitter la partie.

Fonctionnement

La page est composée de plusieurs composants : un composant pour les joueurs, la page garde une liste des joueurs qui est affichée grâce à ce composant. Un composant timer, qui permet d'afficher la représentation du timer en haut de l'écran. Un composant rôle, qui affiche le pictogramme du rôle correspondant. Le composant du chat, et le composant du panneau de configuration. Elle communique avec tous les autres composants via des emit sur "ServerBus", qui est une nouvelle instance de Vue. Elle est importée dans chaque composant qui veut communiquer avec les autres. Dans le composant dans lequel on veut récupérer les données, on utilise ServerBus.\$on.

Handlers

`end_game`: Envoie le message "end_game" au composant `joinPartie`, qui émettra ensuite "join_room" pour remettre le joueur dans la room générale.

`list_ingame`: reçoit la liste des joueurs et appelle la fonction `addplayer`, qui met à jour leur position.

`start_game`: met à jour les variables `nb_infected`, `nurse`, `am`, `cd` qui représentent la présence ou pas de ces rôles dans la partie.

`send_role`: distribue les rôles et appelle `affichezrole`, qui s'occupe de l'affichage du pictogramme associé.

`start_phase`: est déclenché lors d'un changement de phase. Déclenche le timer, change la couleur du fond si nécessaire. Il met également à jour les variables permettant de contrôler qui vote.

`show_nurse`: affiche le rôle renvoyé par le backend uniquement pour l'infirmière.

`designate`: Le message exprime le résultat d'un vote et contient l'auteur du vote, sa cible et la phase. Le handler appelle `addvote`, qui met à jour les pastilles indiquant le nombre de votes pour chaque joueur. Ensuite il met à jour le reste des informations en fonction du message, par exemple il appelle `ajoutmini`, qui positionne l'écharpe du maire après son élection.

`show_casualties`: Change l'affichage des joueurs en fonction des morts.

`game_winner`: Message reçu à la fin de la partie. Affiche le pop-up de victoire et lance le timer avec 1 min.

Lorsqu'un joueur rejoint la partie, on appelle `addPlayer`, qui s'occupe de positionner les joueurs. Elle calcule une position en fonction du nombre de joueurs pour qu'ils soient toujours affichés en cercle. `handleResize` et `handlevoteResize` servent à repositionner les joueurs pour différentes tailles d'écran.

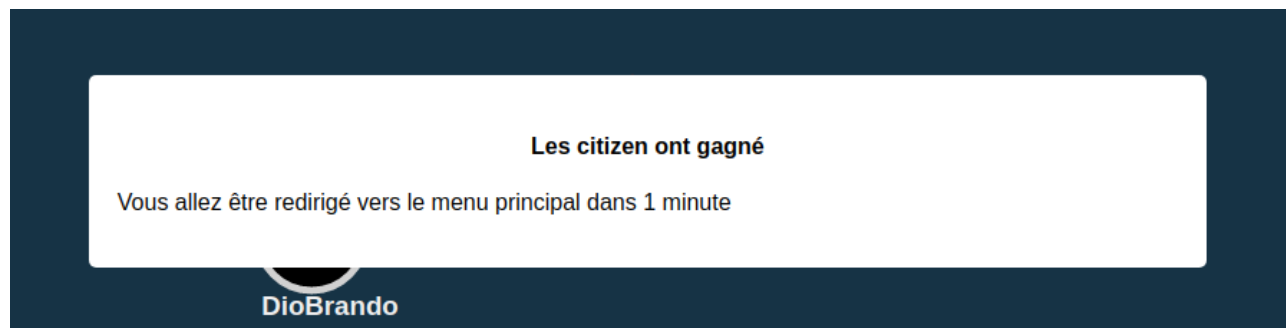
Les méthodes `mouseover` et `mouseout` permettent de gérer l'affichage lors de la phase `cd`. Lors de son choix, les joueurs morts au dernier tour lui apparaissent en noir. Lorsqu'on passe la souris dessus, ils retrouvent leur couleur de base. Quand aux autres joueurs, si on passe la souris dessus, ils apparaissent en noir.

Lorsqu'un joueur clique sur un autre joueur, on appelle la méthode `choice`. Cette méthode vérifie que le joueur qui a cliqué a le droit de voter. Si c'est le `cd`, alors on émet "cd_choice" avec avec son choix: sauver ou tuer la cible, sinon on émet "designate". Lorsqu'il clique sur le bouton confirmer, on émet le message "commit".

On a 2 composants timers différents, qui alternent pour afficher le temps restant. Cela sert à pouvoir couper une animation, par exemple si un `commit` a lieu.

Pendant les votes, nous avons fait le choix d'afficher, à côté des joueurs, le nombre de personnes ayant voté pour eux. Cela permet aux joueurs d'avoir une représentation plus directe de

l'état du vote. Quand un joueur change son vote, un message est également affiché dans le chat. On peut noter la présence d'un bouton "confirmer", qui sert à valider son vote. Une fois que tous les votes sont validés, la phase de vote prend fin, même si le temps imparti n'est pas fini. Cela permet de ne pas attendre la durée de la phase à chaque vote.

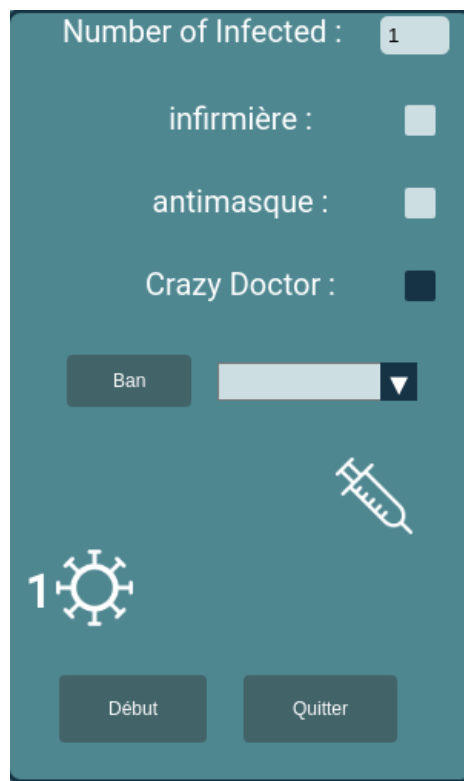


Ecran de victoire à la fin d'une partie

Sous-composants de la page de jeu

Panneau de configuration

Fonctionnement

The image shows a vertical admin configuration panel with a teal background. At the top, it has a label 'Number of Infected :' followed by a light blue input field containing the number '1'. Below this are three rows of labels and checkboxes: 'infirmière :' with an unchecked checkbox, 'antimasque :' with an unchecked checkbox, and 'Crazy Doctor :' with a checked checkbox. Further down is a 'Ban' button next to a text input field and a dropdown arrow. In the center, there is a white syringe icon. At the bottom left is a gear icon with the number '1' next to it. At the bottom right are two buttons: 'Début' and 'Quitter'.

Panneau de configuration version “admin”

Avec ce panneau, l’admin peut exclure un joueur, régler le nombre d’infectés et la présence d’autres rôles que citoyen ou infecté avant de lancer la partie. Il peut également choisir de lancer la game. Pour l’admin et tous les autres joueurs, le menu contient un bouton quitter. Comme il tire toutes ses données de la page de jeu, le composant ne comporte pas de handler. Les paramètres de la partie sont stockés dans des variables. Lorsque l’admin clique sur le bouton pour lancer la partie, un emit “start_game” est produit, avec les paramètres contenus dans les variables.

Lorsque l’utilisateur clique sur le bouton pour quitter, un emit “quit_game” est lancé, puis un join_room avec en paramètre General lobby, pour revenir à la room principale.

Le composant contient un sous-composant BanForm. Dans ce sous-composant, on a 2 fonctions principales. BanPlayer, qui se déclenche lorsqu’on clique sur exclure. Elle emit le message “ban” avec le nom de l’utilisateur correspondant. La 2e fonction, addbannedlist, sert à mettre à jour la liste des joueurs dans le sous-composant. Elle est appelée avec les données tirées de game lors de la création du composant, puis est rappelée chaque fois qu’un nouveau joueur se connecte. La liste des joueurs est stockée dans localStorage.

Chat

Handlers

get_message: ajoute le message reçu au composant du chat

log_out_message: affiche un message de déconnexion

is_typing: met à jour la liste des joueurs qui sont en train d'écrire

start_game: met à jour la variable game_type

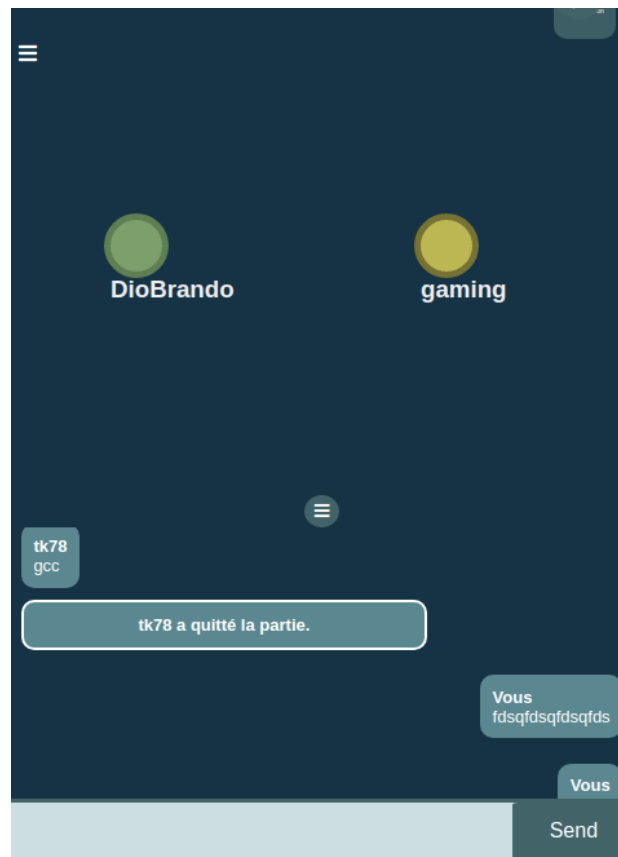
end_game: met à jour la variable game_type

Fonctionnement

Le chat a 2 manières de fonctionner: en game et en lobby. Cela est géré avec la variable `game_type`. En lobby, il n'y a aucune règle particulière, tout le monde dans une même salle de jeu peut envoyer des messages. Cependant pendant une game, le chat doit suivre les règles énoncées dans le MJ. On fait usage du message `"game_send_chat_room"` lorsqu'un utilisateur clique sur envoyer ou appuie sur entrée pendant le jeu. Les 2 manières utilisent le même composant `ChatMessage.vue` pour afficher les messages dynamiquement.

Le chat tient à jour une liste des utilisateurs qui sont en train de taper pour les afficher en bas de la fenêtre. Cette fonctionnalité est faite en utilisant `v-for` pour parcourir un sous-composant `ChatTyper.vue`, dont le style est obtenu par récupérer les paramètres de couleurs que le `game.vue` a envoyé. Lorsque quelqu'un commence à taper, il est ajouté dans la liste, et un timer est lancé. Si le timer arrive au bout, il est retiré de la liste. Dès que l'utilisateur tape sur son clavier, le timer est remis à 0. Pour ceci on émet le message `"typing"` ou `"game_typing"` avec un status `"start"` pour le commencement, et `"end"` à la fin du timer. On utilise `"typing"` dans le lobby et `"game_typing"` sinon, car pendant la partie, on ne veut pas afficher tous les utilisateurs qui sont en train de taper, cela pourrait trahir les infectés pendant qu'ils débattent par exemple.

Version mobile



Nous avons également développé une version mobile du client. La page d'authentification n'a subi aucun changement de par sa simplicité, cependant il y a des différences sur les pages des parties et page de jeu. La page de jeu change totalement de disposition. Le chat n'est plus à droite mais couvre toute la moitié inférieure de l'écran.

L'utilisateur dispose d'un bouton pour agrandir ce dernier. On a profité de v-show pour réaliser cette fonctionnalité. On a préparé deux Chats, une qui prend toute la taille de la page, l'autre qui prend qu'une demie de la page. Si le bouton a été cliqué par l'utilisateur, alors on affiche le chat qui prendra toute la taille de l'écran, sinon, ce chat sera caché et l'autre sera affiché. Il prendra alors toute la taille de l'écran.

Cette fonctionnalité découle du fait que le chat est la partie la plus importante du jeu. En effet, quand on y réfléchit, le jeu pourrait être joué en ligne de commande. L'interface visuel représentant les joueurs sert seulement à voter en cliquant dessus, mais cela pourrait être effectué par des commandes. La page des parties, quant à elle, voit son champ "code de la partie" disparaître. La liste des parties prend la totalité de l'écran, et au bas de l'écran on a un bouton "créer partie" et un bouton "partie privée", qui ouvrent tous deux un pop-up. Le pop-up de création de partie est le même que dans la version desktop, et le pop-up de partie privée contient un champ et un bouton valider.

Communication vocale entre les joueurs

Dans notre jeu, la communication entre les joueurs joue un rôle capital. Les infectés doivent se faire passer pour des citoyens pour ne pas être exclus, les citoyens ayant des rôles importants doivent cacher leur rôle tout en exploitant quand même les informations qu'ils ont pu acquérir. Pour cette raison, nous avons estimé que l'utilisation d'un simple chat à l'écrit briderait l'intérêt de notre jeu. Nous avons alors étudié les possibilités pour permettre aux joueurs de communiquer oralement entre eux.

Nous avons d'abord étudié le développement de notre propre chat vocal avec webRTC, mais nous pensions que travailler sur une telle fonctionnalité serait trop compliqué et trop long pour notre projet. Nous avons ensuite eu l'idée d'utiliser discord couplé à un "bot", un utilisateur automatique capable d'accomplir beaucoup d'actions grâce à des fonctions programmables.

Bien qu'apparut il y a quelques années seulement, Discord s'est rapidement fait une importante place sur le marché des services de communication en ligne. Très simple d'utilisation, il fonctionne sur téléphone grâce à ses applications android et iOS, sur linux, mac et windows grâce à un programme dédié ainsi que sur navigateur pour les utilisateurs de systèmes d'exploitation exotiques ou ceux ne souhaitant rien télécharger.

Il est possible de créer un compte Discord, mais l'utilisation du service "en tant qu'invité" sans compte est également possible.

Le temps de développement d'un "bot" est assez simple grâce à une bibliothèque python dédiée. En plus de cela, les garanties de portabilité et de qualité de l'utilisation d'un service aussi établi ont fini de faire pencher la balance en faveur de son utilisation.

Fonctionnement en mode projet

Afin d'exploiter au maximum le potentiel de chaque membre de l'équipe, nous l'avons séparé en 2: l'équipe front-end et l'équipe back-end. Au début, nous étions 4 en back-end et 6 en front-end, mais rapidement, le manque de main d'œuvre côté front-end s'est fait ressentir, et Ahmed a alors changé d'équipe.

Réunions

Nous effectuons une réunion par semaine, qui suivait toujours à peu près le même déroulement, avec quelques variations selon l'étape du projet. Au début de la réunion, chacun explique son travail de la semaine, pour que tout le monde soit au courant. Ensuite chacun pose ses questions aux personnes concernées, s'il y en a.

Pour atteindre une telle communication, il a fallu guider les équipes et solliciter les membres à participer à la conversation. Ensuite, on compare le travail effectué au Gantt, et on adapte. Si une tâche semble prendre du retard, il est arrivé de déplacer un autre membre de l'équipe sur la tâche. Si elle est terminée avant la deadline, la personne qui y était assignée peut alors en commencer une autre.

En général, les tâches étaient finies au bon moment selon le Gantt. Les assignations étaient ainsi faites en fonction des tâches terminées et celles encore en cours. Bien entendu, chaque membre de l'équipe a des tâches qu'il préfère faire, il a donc fallu également gérer cet aspect du projet. Malheureusement, ce n'est pas toujours possible de satisfaire tout le monde. Quand le besoin s'en présentait, nous effectuons également un point le vendredi soir, c'est-à-dire à peu près au milieu de la semaine de travail. Il est également arrivé que nous fassions des points

spécifiques au front-end, ou encore à telle ou telle fonctionnalité. Quelquefois, les membres de l'équipe front-end, travaillant sur une fonctionnalité dépendante du back-end, avaient besoin de communiquer au membre responsable de cette partie du back-end, mais ne le disaient pas forcément. Le rôle du chef de projet ici était de les mettre en relation, d'encourager ou même d'organiser un appel vocal.

Gitlab

Tout au long de notre projet, nous avons utilisé un dépôt git pour travailler. Cela nous a permis de mettre notre travail en commun, et de voir l'avancement du travail des autres membres de l'équipe. En revanche, nous avons eu une mauvaise utilisation de ce dépôt. En effet, nous avons voulu suivre un gitflow qui consistait à ne merge avec master seulement les fonctionnalités « production ready ».

Choisir ce flow a été une erreur puisqu'il nous a empêché d'utiliser pleinement le dépôt, résultant en un nombre de branche supérieur au nombre de membres dans l'équipe, supérieur même au nombre de fonctionnalités dans le projet final. Cette mauvaise utilisation nous a porté préjudice quelquefois, surtout côté front-end. Il nous est arrivé de rencontrer des problèmes liés à la version du back-end utilisé, ou bien à la version des fonctionnalités que nous essayions d'intégrer ensemble, alors qu'une autre version était déjà développée par la personne responsable de la fonctionnalité. Si le projet était à refaire, nous utiliserions un autre gitflow et surtout, nous ferions attention à ne pas tomber dans la surutilisation de branches différentes.

Documents partagés

Pendant un long moment au début du projet, les spécifications ont été un outil de nécessité, puisqu'elles nous ont permis d'avoir un objectif et des précisions dans notre développement. Nous avons utilisé un document partagé sur google docs. Cependant, les spécifications ne sont pas restées telles qu'elles ont été rédigées au début. En effet, en développant, les équipes se sont rendu compte des points que nous n'avions pas spécifiés, et de ceux que nous avons mal spécifiés. Certains étaient irréalisables, ou tout simplement pas la meilleure solution. Les spécifications ont été alors corrigées, pour que le reste de l'équipe puisse toujours s'y référer et y trouver les bonnes informations.

Gantt

Le gantt est disponible à cette adresse :

https://quire.io/w/cas-contact/62/C_Version_mobile?view=timeline

Au début du projet, nous avons eu beaucoup de mal à mettre en place un Gantt. Cela a été dur pour nous de nous projeter, car nous n'avions pas vraiment d'idée des tâches que nous allions devoir accomplir. De même, avant de commencer à développer, nous avons décidé de faire usage d'un modèle de développement "évolutif", qui consiste en le développement rapide d'une première version fonctionnelle, puis l'ajout des différentes fonctionnalités par la suite. Nous étions censés suivre le plan suivant : chat, puis fonctionnement du plateau de jeu, puis salles d'attente, puis authentification, puis des ajouts concernant l'accessibilité. Nous nous sommes évidemment rendu compte que ce modèle n'était pas réalisable dans notre cas, puisqu'il est impossible de faire travailler 10 personnes en même temps sur uniquement un chat, ou uniquement l'authentification. C'est pour ces raisons que notre version « finale » du Gantt n'a vu le jour qu'à la 4^e semaine du projet. Nous avons eu, avant ça, quelques autres versions, mais qui ne correspondaient pas au réel développement du projet. Après cette version finale, nous nous y sommes tenus la majeure

partie du temps, et avons réussi à respecter la plupart des deadlines, sauf pour l'authentification, dont nous avons dû recommencer l'implémentation. De plus, une fois le front-end disponible pour tester, les parties du back-end précédemment développées se sont "réouvertes", au fur et à mesure que nous trouvions des bugs. Les assignations ont continué à varier au fur à mesure que nous nous rendions compte de ce que les différentes tâches impliquaient en termes de temps de travail. On peut notamment mettre en évidence le changement d'équipe de Ahmed, au vu de la quantité de travail que représente le front-end par rapport au back-end.

Clockify

Nous avons utilisé l'outil clockify pour garder une trace de nos heures de travail.

Communication

Pour le document de communication, nous avons choisi de produire une histoire audio. Notre choix a été influencé par plusieurs choses :

Premièrement, les podcasts et séries audio sont un format qui ne cesse de gagner en popularité ces derniers temps.

Deuxièmement, nous nous sommes dit que c'était un moyen de communication liant l'originalité à la faisabilité. Bien conscient que nous sommes loin d'être des experts en audiovisuel, nous lancer dans un court métrage aurait certainement abouti à un résultat résolument plus amateur, sinon raté. Avec le peu d'expérience et de moyens à notre disposition, la réalisation d'une histoire audio nous semblait être plus à notre portée.

Troisièmement, nous voulions donner une certaine atmosphère à notre jeu. Notre source d'inspiration pour ce projet, le jeu du loup-garou, est avant tout un jeu d'ambiance. Pour en profiter au mieux, les joueurs se mettent généralement dans une pièce tamisée, le maître du jeu joue des intonations dans l'effort de plonger les joueurs dans une certaine atmosphère. Nous nous sommes alors dit qu'une histoire audio serait un excellent moyen de retranscrire cela.

Vous trouverez notre histoire [en suivant ce lien.](#)

Rapports individuels

Paul LABAYE (chef de projet)

Mon rôle

Je n'ai pas été chef de projet dès le début, ce n'était pas mon premier choix. Cependant, l'équipe et moi nous sommes rendu compte des difficultés qui allaient pouvoir être éprouvées par Fariza (originellement cheffe de projet) à endosser ce rôle. En effet, étant salariée, elle n'allait pas pouvoir se consacrer pleinement au projet, et cela risquait de le ralentir. J'ai donc pris la décision de lui en parler, et d'endosser le rôle de chef de projet.

Ce que j'ai fait

Mon rôle principal a été de mettre en relation les gens, et de les diriger. Nous nous sommes très vite aperçus, lorsque j'ai « pris ma fonction », que le travail était réalisé beaucoup plus vite et de manière beaucoup plus instinctive si quelqu'un dirigeait les réunions et réalisait les attributions du travail. La plupart du temps, je n'ai pas eu besoin de prendre d'initiative à proprement parler vis-à-vis du développement, étant donné que les personnes responsables des fonctionnalités avaient une connaissance technique supérieure à la mienne concernant cette dernière. En revanche, même si les membres du groupe, la plupart du temps, savaient à peu près ce qu'ils avaient à faire, je pense que le fait d'en discuter avec moi et les autres membres du groupe les aidait beaucoup. Je dirais même que mes connaissances techniques moins importantes que celles des responsables des fonctionnalités ont permis de débloquent certaines situations en y apportant un regard différent (comme l'aurait fait n'importe quel autre membre de l'équipe, en l'occurrence c'était mon rôle). Pour ce qui est du rôle de chef de projet, j'ai donc : préparé et animé les réunions, réalisé la version finale du Gantt, rédigé les comptes rendus, rédigé en partie le rapport final, mis en relation les différents membres de l'équipe et pris certaines décisions quant au développement.

Rôle directeur artistique

En outre, j'ai également joué le rôle de « directeur artistique » du projet. En effet, je suis responsable de tout l'ihm. J'ai mis en place la plupart des maquettes et décidé de l'apparence du jeu. Parallèlement, je me suis assuré que tout le monde respecte les chartes graphiques mises en place.

Rôle développeur

J'ai également développé certaines fonctionnalités. J'ai majoritairement travaillé sur le chat, et son intégration avec le reste du projet. Au début, toutes les fonctionnalités étaient développées en parallèle avec chacune un fichier html, un javascript et un css. J'ai pris la décision d'utiliser une structure de projet avec webpack en utilisant Vue-cli. Cela consiste à séparer le projet en fichiers contenant chacun un composant (template html, style css et script js). J'en ai pris l'entière responsabilité. Pour ça je me suis documenté, j'ai repris une partie du chat que Qinyue et Clément avaient développé avec une structure « classique » pour l'adapter à la structure webpack et y ajouter le plugin vue-socketio. Par la suite, au cours de plusieurs réunions, j'ai demandé à l'équipe front-end d'adapter le travail qu'ils étaient en train de faire pour qu'il colle à la structure webpack. J'ai notamment fait un point avec l'équipe front-end pour leur montrer comment faire. J'ai pensé que ce serait plus facile pour tout le monde si (en complément de la documentation) quelqu'un leur

montrait comment fonctionnait une telle structure. Je pense que cette initiative a considérablement amélioré notre manière de travailler, et a surtout rendu l'intégration bien plus abordable.

Pierre LAFOSSE

Au cours de ce projet, j'ai principalement travaillé du côté "backend" du jeu. Ma première tâche importante a été de concevoir et d'implémenter l'algorithme qui cadence une partie sur le serveur, que nous avons le plus souvent appelé le maître du jeu ou encore le MJ en référence aux jeux de rôles dans lesquels un participant est chargé de narrer l'histoire et de faire avancer le jeu. Une fois cela terminé, j'ai également travaillé sur le bot discord qui a pour but de faciliter la communication vocale entre les joueurs. Finalement en fin de projet, j'ai également fait le montage de notre histoire audio, notre document de communication.

Le maître du jeu

Cette partie de mon travail a commencé par une assez longue phase de spécification. Du fait que d'autres parties du projet, comme principalement l'algorithme du jeu du côté client, allaient dépendre de cette spécification pour fonctionner correctement, j'ai fait de mon mieux pour détailler suffisamment afin qu'il n'y ait pas de problèmes par la suite. J'ai apporté une attention particulière à la description des messages échangés, donc à leur format et à leur signification en fonction de l'état du jeu. Globalement, je pense que cela a bien fonctionné puisque nous n'avons pas eu de problèmes particuliers sur ce côté là et je n'ai refait que de rares et mineurs modifications à cette spécification en cours de projet.

Une fois cette partie terminée, j'ai commencé l'implémentation. Comme nous avons décidé de travailler du test driven development, les tests et l'implémentation allaient de paire. Comme cela faisait plus d'un an que je n'avais plus fait de python, il m'a fallu quelques sessions de travail pour me remettre dans le bain et pour ne plus rajouter de points-virgule à la fin de chaque ligne. J'ai aussi pris un peu de temps à comprendre comment me servir de socket.IO, la bibliothèque que nous avons utilisé pour les communications entre websockets. Heureusement, une fois habitué, elle est très simple d'utilisation et m'a certainement fait gagner beaucoup d'heures comparé à d'autres solutions.

Par contre, les tests m'ont posé plus de problèmes. Je ne m'étais servi des outils de test de python que dans des cas très simples jusque-là, et je me suis vite rendu compte que j'avais fortement sous-estimé le temps requis pour faire des tests sur un projet de cette envergure. Il a par exemple fallu que j'apprenne à réellement me servir de bouchons et de mocks pour pouvoir tester des bouts de code utilisant des fonctionnalités de socketIO sans pour autant avoir recours à des communications réseau.

Malgré cela, le développement de cette partie s'est à mon sens bien passé. Les tests ont visiblement fait leur effet puisque Ahmed, le très efficace chargé du développement de l'algorithme de jeu côté client n'a été confronté qu'à relativement peu de bugs en intégrant son travail au mien.

Le bot discord

Par le passé, j'avais déjà développé un bot discord pour un projet personnel. J'ai donc assez rapidement retrouvé mes marques. Il nous fallait des salons vocaux accessibles uniquement aux joueurs d'une partie donnée, ce que j'ai géré grâce à des liens d'invitation vers notre serveur discord ajoutant attribuant automatiquement à toute personne les utilisant un rôle propre à leur partie. Ainsi, seuls les joueurs de la partie seront en mesure de se connecter au salon vocal leur

étant attribué. Pour ne pas compliquer les choses inutilement, tous les joueurs d'une partie sont kick du serveur discord à la fin de la partie pour ne pas avoir de problèmes lors des parties ultérieures. Ce système pourrait être amélioré, par exemple en liant un compte discord à un compte Cas contact, mais cela forcerait tous les joueurs souhaitant communiquer à l'oral à avoir un compte discord ce que nous avons décidé de ne pas faire.

Au final, le fonctionnement est assez simple: au début de chaque partie, le bot génère un lien d'invitation spécial et propre à cette partie. Tous les joueurs souhaitant rejoindre le salon vocal peuvent alors se connecter et discuter jusqu'à la fin d'une partie où le bot doit supprimer le rôle de la partie, son salon vocal et kick ses joueurs.

Là où nous avons eu plus de mal fut sur l'intégration du bot discord au reste du projet. Je ne m'étais pas assez renseigné au préalable et cette partie a donc donné du fil à retordre à Clément qui s'en est chargé en fin de projet.

Le montage de l'histoire

Le montage de l'histoire a été tâché qui m'a pris plus de temps que prévu. Pour que notre document de communication soit attrayant, j'ai beaucoup misé sur l'ajout de bruitages et de sons de fond. Clément m'avait déjà fait une sélection d'effets sonores utilisables que j'ai agrandie en trouvant des ressources libres de droit où dont l'utilisation nous était autorisée.

Il a fallu faire de nombreuses sélections entre des prises ratées ou refaites, améliorer autant que possible des prises imparfaites en supprimant par exemple autant de bruits de bouche que possible, équilibrer les volumes des différents personnages présents dans la scène et ainsi de suite.

Au bout du compte, nous sommes contents du résultat. C'était une première pour nous tous, mais nous avons tout de même réussi à aboutir sur un document de communication original dont nous sommes fiers.

Qinyue LIU

Mon rôle

Durant ce projet, j'ai travaillé dans la team front-end. Je me suis principalement travaillé avec vue.js en m'occupant du routing et des composants Chat.vue et Joinpartie.vue, ainsi que de leurs fils.

Ce que j'ai fait

- apprendre comment marcher les sockets
- apprendre le vue
- faire le chat avec le framework vue et un peu de css
- faire la partie lobby_system
- corriger des bugs dans ces deux parties
- faire le routing entre les pages
- faire les styles pour certains pages
- afficher la liste des joueurs en train d'écrire avec la couleur de chaque joueur

Les techniques de vue les plus importantes dans les pages sur lesquelles j'ai travaillé

Les composants "fils"

Par exemple `ChatTyper.vue`, `ChatMessage.vue`, `Table.vue`, etc... sont des composants fils car dans le projet, les messages dans le chat et les tables dans `Joinpartie` doivent être dynamiques. Cela veut dire que ses affichages doivent changer selon les opérations faites par les joueurs. Ils doivent aussi avoir un style css, donc si on les fait comme des composants, les données de leurs composants fils changent dynamiquement selon le composant du père. Il est ainsi plus facile d'ajuster des styles css.

Pour afficher une liste de composants, on se sert d'un `v-for` et d'une clé dans les props du composant. On utilise un `v-bind` pour la clé et les valeurs que l'on veut adapter dynamiquement au composant

`v-on`, `v-model`, etc...

`v-on` permet de ne pas avoir à créer des event listeners. Il sert à superviser certains événements comme `key-down`, `key-up` ou `click` qui sont les plus fréquemment utilisés. `V-model` lie les valeurs dans les inputs avec les variables locales dynamiquement, ce qui rend le debug et l'affichage plus facile.

Routing avec les données

Avant de travailler sur le routing, il faut ajouter toutes les pages nécessaires dans le fichier `index.js` qui se trouve le répertoire du routeur pour qu'elles soient connues de lui.

Pour passer de la page `JoinPartie` à la page `game.vue`, il faut créer une route avec le nom et la salle. Pour vue, il faut juste ajouter ces deux valeurs dans le query avec un chemin de routing. Pour les récupérer, on peut se servir de `$route.query`.

Difficultés et solutions

Ajouter des styles pour chat-messages

Au début, j'ajoutais les styles avec la propriété `style` des éléments du DOM. Cette façon de faire n'était pas pratique du tout. Avec l'aide de Paul j'ai appris comment marchent les composants fils et comment les parcourir. Finalement, j'ai résolu ce problème en utilisant `v-for` et `v-bind`.

Cliquer pour rejoindre une partie publique et routing

Dans le vue, je n'arrivais pas à ajouter d'event listener dans le composant fils, j'ai donc cherché des solutions en ligne. Finalement j'ai découvert que le fils peut envoyer un signal au père en utilisant `$emit` quand il supervise l'événement `click`. Quand le père reçoit le signal de son fils, il peut appeler la fonction `join`. Pour pouvoir routing dans la page `game.vue` avec les données, il faut juste ajouter les données dans le query avec un chemin de routing.

Afficher les joueurs qui sont en train de taper avec leurs couleurs

Cette tâche était difficile pour moi car Ahmed s'était occupé de la création des joueurs dans `game.vue`. Après analyse de son code j'ai finalement compris les joueurs sont ajoutés. J'ai alors utilisé une de ses fonctions pour transmettre les paramètres nécessaires à ma page `Chat.vue`. Il y a eu quelques bugs, comme par exemple le fait qu'à chaque fois qu'un ensemble de paramètres était ajouté à ma style-liste, le `v-for` parcourait tout sans vérifier si un joueur donné était en train de

décrire ou non... Pour rectifier ce problème, j'ai donc utilisé une liste dynamique contenant les noms des joueurs de la typingList et qui utilise les noms et la style-liste pour trouver le bon style.

Ce que j'ai appris dans ce projet

- Il ne faut pas oublier de pull à chaque fois avant de modifier le projet...
- Comment le frontend et backend collaborer avec les sockets
- Pas mal de choses dans vue et vue-cli

Shufeng JIANG

Mon rôle

J'ai principalement travaillé sur le front-end.

Technologies utilisés

Tout d'abord il faut construire des sites web avec html et css. Puis pour réaliser les fonctions j'ai appris comment le socket fonctionne et étudié les structures qui sont écrites par backend et trier et afficher les données qu'on a besoin, comme on n'a pas décidé le framework à utiliser, j'ai réalisé les fonctions en JS.

Ce que j'ai fait

Ensuite, d'après la discussion de notre groupe, nous avons décidé d'utiliser le framework Vue, donc je relie les actions de Vue avec les fonctions de JS. Après j'ai les transforme les codes en type de vue composant et vue.cli.

Enfin, on détecte et répare les bugs, change les styles ou ajoute les nouveaux modules et réalise les pages de la version mobile.

Les techniques en détail

css :

la « position: absolute » pour fixer les positions de chaque module

l'unité « vm » pour faire la taille des mots correspondre à la taille de la fenêtre

display pour contrôler si on affiche le module ou pas

etc

vue :

- v-on relier les actions et les fonctions
- v-model afficher dynamiquement les variables
- v-for parcourt les variables dans les tables
- mettre les différentes composantes dans les fichier .vue etc.

Clément LIGNEUL

Dans ce projet j'étais associé à l'équipe back-end et j'ai principalement travaillé sur la partie communication entre les clients et le serveur mais aussi sur la communication entre les clients. Cela comprend la gestion des parties mais aussi toutes les fonctions qui permettent le bon déroulement du jeu et qui ne concernent pas le maître du jeu.

Implémentation du chat de partie

Le premier module sur lequel j'ai travaillé est le chat qui permet aux joueurs de discuter entre eux au cours d'une partie. J'ai d'abord fait une implémentation intégralement en python (serveur et client) de manière à pouvoir mieux comprendre le fonctionnement des communications via socketio. Le chat était très basique, il fonctionnait en ligne de commandes et les affichages étaient plus proches de l'ascii art plutôt que d'un client web. Après des tests concluants j'ai pu passer le client en version javascript, ce qui est déjà plus proche de ce que nous souhaitons faire pour le projet. Afin de garder une maîtrise totale sur ce que je faisais j'ai pris le temps d'ajouter des commandes spéciales qui servaient à obtenir certaines informations de debug. Une fois que le chat en version js est devenu utilisable pour le projet, Céline a pu l'adapter en version vuejs afin qu'il soit complètement intégré avec le reste du client. La principale difficulté de cette tâche était la gestion des rooms. Ce problème est encore plus important avec la mise en place du système de parties.

Pour pouvoir garder les informations sur les utilisateurs connectés et les réutiliser plus facilement j'ai créé un tableau qui regroupe toutes les informations des joueurs au même endroit (dictionnaire global appelé "connected"). Ce tableau joue un rôle fondamental sur l'ensemble du reste du projet.

Implémentation du système de parties

Le système de parties a pour but de guider le joueur en vue de rejoindre ou de créer sa propre partie. Il a d'abord fallu définir ce qu'était une partie. Il s'agit d'une room particulière dans laquelle se rassemblent un certain nombre de joueurs. Un joueur à la tête de ce groupe est appelé "admin", c'est le joueur qui peut exécuter des actions supplémentaires comme bannir quelqu'un ou lancer la partie qu'il a paramétré au préalable. Pour pouvoir avoir ce fonctionnement j'ai implémenté un autre tableau global (appelé game_tab) qui contient l'ensemble des parties créées sur le serveur. Dans chaque ligne on retrouve toutes les informations nécessaires sur la partie mais aussi sur les joueurs qui la composent comme par exemple le nombre de personnes, l'objet game servant à l'exécution du maître du jeu etc... Au départ le tableau ne contenait que peu d'attributs par conséquent un dictionnaire était plus approprié. Puis il a fallu ajouter encore d'autres attributs rendant la liste associée à la clef beaucoup plus longues et donc la manipulation plus complexe. J'avais donc la possibilité de transformer cette variable en classe un peu plus structurée. Cependant le projet avait déjà bien avancé et faire un tel changement prendrait beaucoup de temps sans compter la quantité de bugs que cela allait amener. J'ai donc décidé de garder la structure en tant que telle en me disant qu'il ne fallait plus la compliquer davantage.

En se basant sur ce tableau j'ai donc pu implémenter les parties publiques et privées qui nécessitent un code pour être rejointes. La principale difficulté était la gestion des tableaux game_tab et connected en plus de devoir gérer correctement les différentes rooms.

Plus tard au cours du développement j'ai mis en place une room "générale" qui permet de regrouper au même endroit tous les joueurs qui viennent de se connecter ou qui viennent de quitter une partie. Elle permet de se repérer plus facilement pour la gestion des joueurs. Dans

l'implémentation il est possible d'utiliser le chat dans la room générale mais ce n'est pas une fonctionnalité qui a été retenue pour le projet final.

Implémentation de la fonction typing

La fonction que l'on appelle typing est la fonction qui permet de savoir si un utilisateur est en train d'écrire dans le chat. L'implémentation de cette partie a demandé un peu plus de réflexion sur la manière dont allait fonctionner l'algorithme. Au final j'utilise un système de timer dans le client qui permet d'appeler périodiquement une fonction si quelqu'un est en train d'écrire (touche pressée).

Cette fonction a demandé beaucoup d'attention car on modifie un tableau regroupant tous les joueurs écrivant dans le chat. Il était donc possible que cela ne fonctionne plus indiquant qu'il y avait un problème dans la gestion des rooms. Ce module a donc aussi servi involontairement au debug.

Implémentation du chat vocal

Nous avons déjà la possibilité de faire communiquer les joueurs à l'écrit grâce au chat de partie mais comme notre jeu est censé être plus vivant nous avons décidé d'ajouter la possibilité de communiquer oralement. Pour cela j'ai effectué des recherches sur l'éventuelle utilisation de l'api WebRTC. Il s'agit d'une interface de programmation assez utilisée pour les applications de communication par VoIP. Le fonctionnement est assez complexe mais semblait être utilisable pour notre projet. Cependant pour pouvoir l'intégrer au client il fallait une importante quantité de travail et mobiliser au moins deux personnes dessus. L'équipe ayant déjà des difficultés sur certaines parties à ce moment-là, il n'était pas judicieux d'augmenter la charge de travail pour ce que cela allait apporter.

Après réflexion nous nous sommes donc tournés vers un serveur discord géré par un "bot" que les joueurs pourraient rejoindre au début de la partie. Pierre a d'abord implémenté le fonctionnement avec les fonctions d'ouverture/fermeture de salon, d'activation du bot etc... Mon travail était de faire l'intégration de ce serveur dans notre jeu. Ce fût l'une des tâches les plus ardues car le serveur discord et notre serveur de jeu (fonctionnant avec uvicorn) nécessitent deux exécutions différentes mais nous avons besoin de les exécuter en parallèle en plus de les faire communiquer. J'ai donc d'abord réussi à les faire tourner en parallèle en utilisant des threads. Mais notre code n'étant pas optimisé pour fonctionner avec des threads nous avons eu de très gros problèmes d'exécution indéterministe du jeu.

Après de très nombreuses tentatives et un long travail, j'ai finalement décidé de procéder autrement. Les deux serveurs s'exécutent sur deux processus différents et la communication serveur de jeu vers serveur discord se fait via des webhooks, dans l'autre sens elle se fait via des lectures/écritures dans des fichiers. Ce n'est pas la meilleure solution mais au final elle nous permet d'avoir un chat vocal sans toucher à notre serveur de jeu.

Écriture de l'histoire pour la partie communication

Pour la partie communication de notre projet j'ai écrit l'histoire avec Paul et participé à l'enregistrement. J'ai aussi aidé Pierre dans le montage en préparant un certain nombre de bruits d'ambiance qu'il a pu compléter.

Pour l'ensemble du développement des modules sur lesquels j'ai travaillé j'ai utilisé mon propre "faux client" écrit en js pur. Plus fonctionnel qu'esthétique, il m'a permis de faire des tests et

de prendre des décisions sans gêner ou devoir attendre le développement de certaines parties du client "réel".

Karim KHATER

Mon rôle

Dans ce projet, je faisais partie de l'équipe front-end.

Ce que j'ai fait

J'ai commencé par travailler sur le design et l'interface du jeu. J'ai choisi la palette de couleurs et les premières wireframes. Ensuite, je suis devenu responsable de la page de configuration qui est la barre latérale responsable du démarrage du jeu, du bannissement des joueurs, et du choix du nombre de joueurs, des infectés, de l'infirmière, du crazy doctor, et des anti-masques.

Les contenues de page de config

- un slider pour choisir le nombre d'infectés
- 3 checkboxes pour l'antimasque, l'infirmière et le crazy-doctor
- un select box qui contient la liste des joueurs
- un bouton ban pour commit le bannir d'un joueur en utilisant le select box mentionné avant "ban"
- des logos des infectés, infirmière, crazy-doctor et antimasque
- un bouton de commencement de jeux "start"
- un bouton de quitter la partie "leave"

Sur la page de configuration, j'ai commencé par concevoir le panneau sans fonctions dedans et tester son interactivité avec le reste de la page. Ensuite, j'ai ajouté les cases à cocher et les logos des paramètres qui dépendent de la valeur des cases à cocher puis j'ai ajouté le bouton de démarrage et les ai connectés avec le backend à l'aide de Ahmed. Ensuite j'ai fait une autre version de la page pour les joueurs non-administrateurs qui n'ont pas de bouton de bannissement ou de bouton de démarrage et ne peuvent pas modifier les paramètres. A ce stade, tout fonctionnait bien jusqu'à ce que je doive implémenter le bouton de bannissement et sa boîte de sélection. L'implémentation d'une boîte de sélection qui reçoit des entrées de façon dynamique était un gros problème, surtout parce que je suis nouveau dans VueJS. J'ai dû récupérer la liste depuis le backend, puis attribuer un v-for aux options et sauvegarder cette liste à l'aide de 'localStorage' pour éviter que la liste ne disparaisse à chaque fois que je ferme le sidebar. Ensuite, la phase d'intégration continue est arrivée, et j'ai dû assigner tout ce que j'ai fait avec le vrai backend. C'était un peu difficile, car je devais comprendre les fonctions mises en œuvre et adapter mon travail à ces fonctions, et déterminer si une erreur était due à ma partie ou à celle de l'autre personne.

Au final, je pense que nous avons bien travaillé, et le résultat était bon, c'était une bonne première expérience pour nous aider à apprendre comment travailler en groupe.

Lafdhal AHMED

Mon rôle

Au début du projet et durant 3~4 semaines j'étais dans l'équipe du backend j'ai donc commencé par me documenter sur les technologie qu'on allait utiliser par la suite (fastapi, python socketio, docker ..etc) et j'ai participé au développement du chat côté backend. Après qu'on ait constaté un retard au niveau du frontend, on m'a mis dans l'équipe du frontend pour qu'on puisse avancer. Je me suis rapidement mis à commencer le développement en commençant par la documentation de vue.js. Ensuite j'ai lu la spécification du mj faite par Pierre ainsi son code pour mieux comprendre la spécification.

Ce que j'ai fait

Mon travail était de faire l'interface de la page du jeu. Voici les grandes phases de développement :

- premièrement j'ai commencé par la mise en place des joueurs. C'était la phase la plus compliquée, surtout faire en sorte que les joueurs soient repositionnés à chaque fois qu'un joueur se connecte ou s'en va. Au début j'ai mis la taille des joueur en fonction du taille de l'écran en utilisant des des tailles relatives au lieu des pixels mais quand en implémentant les icônes du vote, je me suis aperçu que je ne pouvais pas les positionner au bon endroit parce que ce n'était pas les même pourcentages et j'ai donc été contraint de tout changer.

- la mise en place des logos pour chaque joueur selon son rôle.

- la mise en place du timer: j'ai eu des difficultés lors de l'implémentation du bouton de validation. Comme on pouvait passer à la phase suivante avant la fin du temps imparti, il fallait que le timer puisse se mettre à jour. J'ai donc dû implémenter un second timer qui alterne premier timer entre les phases.

- la mise en place du l'icone contenant le nombre des votes: pas des problèmes en changeant la position du joueur

- la mise en place de Didier Ratoul: changement des couleurs en noir lorsque l'on passe la souris sur un joueur si on veut le tuer, changement vers la couleur initiale si on veut le sauver.

- mettre la bande du maire: pas de problèmes après le changement du positionnement des joueurs.

- le bouton de validation

- le pop-up de fin du jeu et remettre les joueurs sur la page précédente

- l'envoi des données au backend et le traitement des données reçu du backend

- la chasse aux bugs avec Pierre que lui de sa part répond rapidement et corrige rapidement

- faire une part de l'intégration

- j'ai aussi participé à une séance d'enregistrement de l'histoire

Difficultés durant le projet

Personnellement, je trouve que le plus difficile a été de tester. Pour jouer il faut au minimum cinq joueurs. Cela implique qu'il fallait ouvrir cinq pages différentes, renseigner cinq fois le nom et rejoindre cinq fois une partie dès que l'on voulait tester quelque chose.

Ce que j'ai appris dans ce projet

Ce projet m'a appris beaucoup de choses, avant je n'aimais pas tout ce qui était web et sockets, mais grâce à ce projet j'ai appris à mieux les comprendre. J'ai aussi appris comment travailler en équipe, puisque c'était ma première expérience avec git. J'ai aussi appris l'importance de bien nommer ses fonctions et ses variables.

Paco KLEITZ

Mon rôle

J'ai choisi de travailler sur la partie backend, ayant déjà de l'expérience dans ce domaine. J'ai principalement travaillé sur l'architecture et la partie gestion d'utilisateur/authentification du backend, mais j'ai également récupéré la partie authentification/routing du client, un autre membre de l'équipe nous ayant quitté.

Mise en place de l'api pour l'authentification

J'ai commencé par mettre en place une API REST pour l'authentification. J'ai utilisé pour ça un framework web avec lequel j'avais déjà travaillé, je n'ai donc pas eu de mal à réaliser cette tâche. J'ai utilisé Docker depuis le début du développement du backend, la base de données est donc conteneurisée tout comme le serveur qui regroupe l'api et la communication socket. Le lancement du serveur et de la base de données se fait en une commande grâce à docker-compose qui déploie les containers et leur permet de communiquer via un réseau local propre à Docker.

Persistance de l'authentification

Nous voulions que l'utilisateur n'ait pas à se reconnecter à chaque session. Nous avons pour ça utilisé les Json Web Tokens qui sont un moyen de donner à l'utilisateur un objet lui permettant de prouver son identité sans avoir à se reconnecter. J'ai donc dû améliorer l'authentification de base pour renvoyer un token au lieu de connecter directement l'utilisateur.

Ajout de socket.io au serveur d'authentification

L'ajout de la communication socket au serveur a été compliquée, j'ai passé énormément de temps sur un problème d'upgrade de connexion http vers websocket. Après de nombreuses tentatives, j'ai appris que le serveur que nous utilisions ne permettait pas d'avoir un balanceur de charge qui lui transmet son trafic, car il n'était pas compatible avec certains headers http. Heureusement une version dérivée du serveur que nous utilisions supporte ces headers, le changement n'a donc été que de quelques lignes de configuration.

J'ai également perdu beaucoup de temps avec une erreur CORS dû à un header en double, l'application socket.io étant monté par dessus l'application fastapi, nous avons donc un double CORS. La documentation de python-socketio n'étant pas très complète, j'ai eu du mal à trouver comment désactiver le CORS.

Fariza LARBI :

En premier temps je me suis occupée d'une partie de design du jeu. Mon travail consistait à limiter l'utilisateur à envoyer un message toutes les deux secondes. Ceci a pour but d'empêcher à ce que le système de communication entre les joueurs soit abusé et rendre le jeu désagréable. Avant tout cela j'ai dû me familiariser avec l'espace de travail que mon groupe a décidé d'utiliser pour mener à bien ce projet. En premier temps, j'ai dû installer l'application Docker, cette application m'a

permis d'accéder aux fonctionnalités du système L'Unix ceci m'était crucial puisque le seul système d'exploitation que je possède est Windows. Ensuite, on avait besoin d'une application qui permet de rendre notre travail plus élégant et productif ce qui nous a mené à installer l'application NPM et enfin le dernier outil dont j'avais besoin était Python avec tous ses dépendances. Le plus grand défi auquel j'ai dû faire face c'est de relire et comprendre le travail de mes prédécesseurs pour que ma fonction puisse s'incruster sans impacter le travail des autres. Grâce la méthode prédéfini par JavaScript intitulé timeout, elle a pu effectuer le comportement dont je m'attendais, son travail consiste à désactiver pendant quelques secondes le bouton envoyer, ensuite grâce à la fonction appendMessage préécrite par ma collègue, la fonction est censé d'avertir l'utilisateur la raison pour laquelle il ne peut pas envoyer un message en l'appelant ainsi AppendMessage(« System », «attention vous envoyez énormément message) seulement-si le Bottom « envoyer » est désactivé (if (btn.disable)) mais ce message de notification n'a pas pu être implanté parce qu'il y a eu un soucis lorsqu'un collègue a implanté sa propre fonction du coup ceci fut reporter. Ensuite je me suis occupé de la partie de communication de notre produit. Pour faire part de notre application au monde entier j'ai créé une page les réseaux sociaux pour potentiellement attirer des nouvelles clientèles, ou de communiquer les derniers mise à jour des fonctionnalités mise à disposition.