

Chuu chat
DOSSIER
DE
PROJET
PROFES
SIONNEL

* Introduction	3
Présentation Personnelle	3
Présentation du projet en Anglais	3
Compétences couvertes pas le projet	4
* Cahier des charges	4
Analyse de l'existant	4
Les utilisateurs du projet	4
La Livraison	5
Les fonctionnalités attendues	5
Application mobile:	5
Application web:	5
Contexte technique	6
* Conception du projet	6
Architecture	6
Architecture globale	6
Schéma des d'architectures du projet	7
Architecturale du back-end	7
Architecturale du front-end	7
Architecture Logicielle	7
Choix de développement	8
Les langages	8
Les frameworks	9
Logiciels et autres outils	9
* Organisation du travail d'équipe	10
Méthode agile	10
Méthodologie des tickets	10
Méthodologie de Git et des branches	11
Méthodologie Git	11
Méthodologie des branches	11
Trello	12
Tableau de conception	12
Tableau concernant la partie back	12
Tableau concernant la partie front	12
Tableau concernant les détails du front	12
* Conception backend de l'application	13
La base de données	13
Conception de la base de données	13
Modèle conceptuel de données	13
Modèle logique de données	14
* Conception du front-end de l'application	16
Arborescence du projet	16
Moodboard	17
Charte graphique	17

Maquettage	18
Wireframe	18
Haute fidélité	18
* Développement du backend de l'application	18
Organisation	18
Fonctionnement de l'API	19
Middleware	20
Routage	20
Connexion à la base de données	22
Structuration du code	23
MVC	23
Controller/Model	23
Sécurité	23
Injection SQL	24
Les tables arc-en-ciel	24
JWT	24
Gestion des Droits	26
Recherches Anglophone	27
Exemple: Ajout d'un utilisateur à une room depuis l'API	27
Tests	31
Documentation	32
* Développement du front-end de l'application	32
Arborescence	32
Pages et composants	33
Sécurités	35
Secure Store	35
Problématiques rencontrées	36
Socket.io	36
Socket.io-client	37
Navigation imbriquée	38
* Conception de l'espace administrateur	39
User Story	39
Langage	40
Conception du frontend du site web	40
Charte graphique	40
Maquettage	40
Conception du back end du site web	40
* Axe d'amélioration	40
* Conclusion	41
* Annexes	42

★ Introduction

Présentation Personnelle

Je m'appelle Naomi Monderer. Je suis en cours de cursus Coding School à La Plateforme Marseille dans la promotion 2021-2022. Je suis actuellement en deuxième année et je me prépare au titre de Concepteur développeur d'application web et mobile.

J'ai découvert le code sur Openclassroom.com en essayant des formations gratuites puis je me suis orientée vers une formation qui proposait un cadre et du présentiel pour encrer ma formation dans un apprentissage collectif. J'ai collaboré avec Laura Savickaite, Laura Scognamiglio, Dorian Palace et Daouda Sarr sur ce projet.

Présentation du projet en Anglais

Since a few years, a style of music became more and more popular. I speak about Kpop music. Kpop started to exist in South Korea in the 90's and we can see his evolution from this time until today. It is a cultural and strong object which is impregnated about so much of Korean identity. In Europe we are flooded of Korean culture. This style of music has crossed many borders and now we are listening to this music in France. That's why people are more and more interested into Kpop.

Chuu is a cute way to say "Kiss" in Korean. Based on idols and groups, The intention behind Chuu was to create a platform where people with similar taste in music could meet and communicate with each other. The chat is divided into different "rooms" corresponding to Kpop groups and a single "common" room that would help people to present and integrate themselves. The user can choose which rooms they want to get into, to chat about their favorite group with other fans, without creating any discord, which you can usually find in other platforms such as Twitter for example. The discords are created by one or several administrators and they can ban them. The disruptors from the room to avert any "trolls", which is a common phenomenon on the internet. So the goal was to keep the best in this culture without the violence we can find into chat applications or forums.

What is the tech stack ?

We focused on Javascript as it is a modern and flexible language. We used Node.js coupled with Express.js for API development, MySQL for the communication with the database and the React Native Framework for the development of the mobile application. The web application was developed in Javascript native. Additionally, WebSocket technology is employed to enable real-time communication and instant messaging functionality for the user.

experience of the chat. We also implement the JSON WEB TOKEN to securate the authentication of users on the app.

Compétences couvertes par le projet

Ce projet couvre les compétences du titre suivantes :

- Maquetter une application
- Développer des composants d'accès aux données
- Développer la partie front-end d'une interface utilisateur web
- Développer la partie back-end d'une interface utilisateur web
- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données
- Concevoir une application
- Développer des composants métier
- Construire une application organisée en couches
- Développer une application mobile
- Préparer et exécuter les plans de tests d'une application
- Préparer et exécuter le déploiement d'une application

★ Cahier des charges

Analyse de l'existant

Depuis quelques années déjà, un genre particulier de musique s'est popularisé, ramenant avec lui de nombreux admirateurs tout autour du globe. En effet, la Kpop (musique pop coréenne) s'est répandue au-delà de ses frontières. Basé sur un principe d'idols et de groupes, notre application a pour objectif d'exploiter ce matériau afin de mettre en lien les différentes communautés touchées par cette pratique musicale. Tisser du lien entre les fans renforce la communauté tout en la faisant s'épanouir et quoi de mieux qu'un réseau de chat en ligne pour offrir ces moments de partage ?

Les utilisateurs du projet

Ce projet se décompose en deux parties : une application mobile et une interface web.

La partie application mobile sera utilisée par la communauté de fan de Kpop ou communément appelés "selca". Le compte utilisateur est créé par l'utilisateur lui-même. Cet espace est géré par l'utilisateur, permet de s'ajouter ou de se retirer d'une room à laquelle il s'est inscrite et de discuter en temps réel avec les autres participants.

La partie site web est utilisée uniquement par les personnes ayant des droits

administrateur. Il s'agit d'un panel administrateur. Dans cet espace, il est possible d'organiser toute la gestion et la modération du chat.

La Livraison

Les livrables attendus comprennent une application de chat mobile avec un pannel admin pour la gestion des messages, des salles et des utilisateurs. Les utilisateurs pourront créer et personnaliser leurs profils, envoyer des messages sur des canaux publics et participer à des conversations enrichissantes. Le projet implique la création de maquettes Figma (wireframes et haute fidélité) pour visualiser le design de l'application.

Les fonctionnalités attendues

Application mobile:

Page “Inscription” :

Sur cette page, l'utilisateur doit remplir un formulaire et renseigner son adresse email, le pseudonyme qu'il souhaite utiliser, et un mot passe. Il est ensuite redirigé sur la page de connexion et peut se connecter à l'application.

Page “Connexion”:

Lorsque l'utilisateur lance l'application mobile , il est redirigé vers une page d'accueil sur laquelle il doit se connecter avec l'username et le mot de passe qu'il aura choisi lors de son inscription.

Page “Les canaux de discussions”:

Sur cette page on trouve tous les canaux de discussion qui donnent accès aux différents chat. Chaque canal représente un groupe d'artistes de K-Pop. Pour accéder aux chats, il faut s'ajouter en cliquant sur les pastilles qui représentent ces groupes d'artistes.

Page “Ma liste de chat ”:

Cette page représente en quelque sorte la liste des canaux de discussion auxquels l'utilisateur participe déjà et permet en cliquant sur un élément de la liste, d'accéder au contenu du chat et de rejoindre le fil de discussion.

Page “Chat”:

Cette page représente le cœur du projet. C'est ici qu'il est possible de poster et recevoir les messages des autres membres de ce canal de discussion.

Page “Modification du profil”:

Cette page permet à l'utilisateur de modifier ses données personnelles. Il peut donc mettre à jour son email, son pseudonyme et son mot de passe.

Application web:

Page “index”:

Il s'agit de la page d'accueil. vous pouvez commencer à vous déplacer dans les pages de gestion via le header.

Page “users”:

Sur cette page, les administrateurs peuvent changer le pseudonyme des utilisateurs mais aussi changer le rôle des utilisateurs selon trois rôles différents. Un administrateur a le droit d'interdire l'accès à un utilisateur à un ou plusieurs canaux de discussion, mais il peut aussi lui faire accéder au statut d'administrateur.

Page “messagesByRoom”:

Cette page invite l'administrateur à choisir un canal de discussion pour accéder aux messages contenus dedans.

Page “messages”:

L'administrateur peut supprimer des messages d'utilisateurs s'il juge ceux-ci comme étant inappropriés, insultants ou discriminants.

Page “rooms”:

L'administrateur peut mettre à jour le nom des canaux de discussion, peut supprimer un ou des canaux de discussions et en ajouter de nouveaux selon les besoins.

Contexte technique

L'application mobile devra être accessible sur les systèmes d'exploitation IOS et Android. L'application web devra être accessible sur tous les navigateurs. Comme décrit plus haut les choix des outils et des technos sont :

★ Conception du projet

Architecture

Architecture globale

Notre projet se base sur une **architecture 3-tiers** (ou plus généralement appelée **multi-couches**). Les différentes couches de l'application sont séparées:

- **La couche client** matérialisée par l'application mobile et web
- **La couche applicative** (couche de logique métier) matérialisée par l'API
- **La couche de persistance des données** matérialisée par la base de données.

Cette architecture 3-tiers est également une architecture “**client-serveur**”. Plus précisément, il s'agit de l'environnement dans lequel nos applications de machines clientes (les applications web et mobile) communiquent avec notre application de machines de type serveurs (l'api).

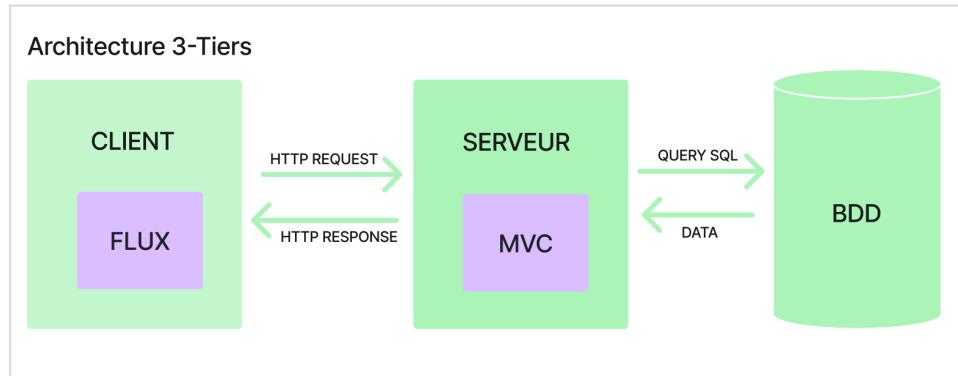


Schéma des d'architectures du projet

Architecturale du back-end

Elles se rapprochent de l'architecture MVC bien que nous n'ayons pas conçu de répertoire modèle qui représente la “couche data”. Notre api est séparée en différents répertoires :

- Le router
- La couche applicative (controller/modèle)

Architecturale du front-end

L'architecture Flux dans un projet React Native est un modèle de conception de flux de données unidirectionnel. Elle comprend des composants React Native pour l'interface utilisateur, un état centralisé géré par un store, des actions déclenchées par les composants, des réducteurs qui mettent à jour l'état en réponse aux actions, et des souscriptions aux changements d'état pour réagir et mettre à jour l'interface utilisateur en conséquence. Cela permet de maintenir un flux de données clair et prévisible dans l'application.

Architecture Logicielle

Les utilisateurs auront accès à l'application mobile mais les administrateurs auront accès à la l'application web et mobile. L'application mobile et le site web utilisent la même API. L'API communique avec ma base de données MySQL.

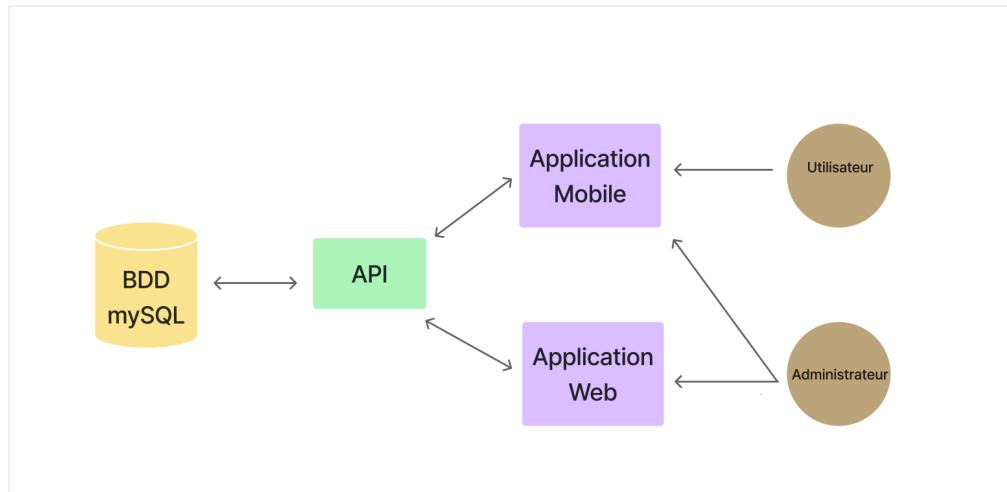


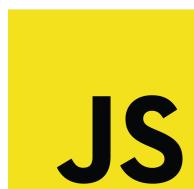
Schéma de l'architecture logiciel

Choix de développement

Les langages



Avec Node Js comme environnement de développement.



Pour réaliser ce projet, nous avons décidé d'utiliser uniquement du javascript.

Nous avons fait ce choix pour plusieurs raisons: Javascript est un langage riche avec de nombreux concepts, nous permettant une montée en compétences. C'est un langage beaucoup utilisé par les géants du web. Ces derniers ont créé de nombreuses bibliothèques de code open source facilitant ainsi le développement de certaines fonctionnalités. Il est présent dans toutes les applications web mais aussi mobiles. Il n'existe à ce jour plus aucunes pages web qui n'utilisent pas cette technologie pour dynamiser son contenu.

Nous avons utilisé SQL comme langage pour interagir avec notre base de données et MySQL comme gestionnaire de base de données.



Les frameworks

Pour la création de notre API nous avons choisi d'utiliser Express.js qui est un framework de NodeJs.

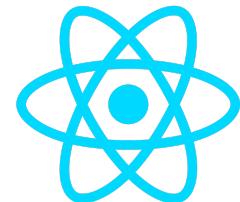
Express.Js

Express.Js est le framework le plus populaire pour NodeJs. C'est un framework minimaliste permettant de garder un certain contrôle dans le développement du projet et apporte peu de surcouche permettant ainsi de garder des performances optimales et une exécution rapide.



développer, de façon rapide et efficace, une API. Ce qui correspondait parfaitement à notre besoin.

Nous avons choisi Express pour son cadre d'exploitation libre et gratuit. Il comporte un ensemble de paquets amenant des fonctionnalités et des outils qui aident à simplifier le développement. Ayant un calendrier à respecter, Express Js permet de



React Native

Pour la création de notre application mobile, nous avons utilisé le framework react native car il est écrit en javascript et qu'il permet le développement d'un seul code pour les plateformes iOS et Android.

Expo

Expo est une plateforme de développement d'applications mobiles multiplateformes qui facilite la création d'applications pour iOS, Android et le web. Elle est basée sur React Native, un framework populaire pour le développement d'applications mobiles. Cette plateforme nous a permis de mettre en place notre environnement de développement côté front. Il inclut également un serveur de développement qui permet de voir les modifications en temps réel pendant le développement.

Logiciels et autres outils

Dans le cadre de ce projet, nous avons dû utiliser d'autres outils:

- **Visual Studio Code** pour écrire notre code;
- **Postman** pour effectuer les requêtes API;
- **Git Bash** et **GitHub** pour le versionning de notre code;
- **Trello** pour organiser notre travail;
- **NPM** pour installer les paquets;
- **Figma** pour la création de nos maquettes;
- **Figjam** pour la conception de la base de données (MCD/MLD).
- **Photoshop** pour la création de plusieurs backgrounds de l'application.
- **Socket.io** pour l'interaction dans le chat en temps réel.
- **JWT** pour l'authentification des utilisateurs et la sécurité

★ Organisation du travail d'équipe

Méthode agile

Les méthodologies de gestion de projet dites méthodes agiles ont toutes un point commun: elles sont inspirées du Manifeste Agile édité en 2001 par des développeurs de logiciels bien décidés à améliorer leur process et à réduire leur taux d'échec.

Nous avons suivi la méthode **Agile SCRUM** pour suivre la méthodologie de travail déjà appliquée dans nos alternances respectives.

C'est un cadre de gestion de projet qui divise les projets en plusieurs phases dynamiques appelées "**sprints**". Après chaque "sprint", nous parlions des éléments qui pourraient être améliorés pour le prochain, en désignant les éléments bloquants et les éléments ralentisseurs. Cette pratique rentre dans la phase du **rétro-planning**. Cela nous a permis une certaine flexibilité dans l'évolution du projet, nous permettant de réadapter nos tickets et notre planning en fonction des besoins.

Les points et moments d'échanges étaient aussi journaliers dont l'appellation "**daily**" faisant partie de la méthode scrum. Bien sûr, cette méthode est chapotée par un **scrum master**. Cette personne a pour rôle de garantir que les processus scrum soient appliqués. Dans notre cas, nous avons décidé de manière collégiale qu'un membre de notre groupe de travail serait chargé de cette fonction.

Méthodologie des tickets

Pour le front comme pour le back les tickets sont créés sur trello et découpés en 3 parties:

- Une description de la fonctionnalité vue par l'utilisateur
- Une description plus technique pour les développeurs
- Une description axée sur la sécurité de la fonctionnalité

Le but de cette liste de tickets était de permettre à l'équipe de s'auto-attribuer des tâches en respectant le nommage des tickets de son propre nom. Les colonnes du tableau indiquent l'état d'avancement des tâches: à faire, en cours, recette, terminé.

Pour le front une capture d'écran précise de la maquette est intégrée au ticket de manière à imager le travail à effectuer.

The screenshot shows a Trello card with the following details:

- Title:** 05. API/ Route POST user to a room LISA
- Label:** Dans la liste SPRINT/1
- Members:** A placeholder icon with a '+' sign.
- Notifications:** A dropdown menu showing 'Suivie'.
- Description:** A large text area containing:
 - As a prospect : je veux m'inscrire à une room afin de participer au chat de ce dernier. // à discuter si ce champs est dispo dans le profil
 - Description : Une route qui permet de s'inscrire à une room
-> post le tableau participants.
 - Sécurité :
 - route accessible UNIQUEMENT aux users connectés.
 - vérifier que l'utilisateur est connecté.
 - SI PROFIL - vérifier que c'est bien le profil de l'user connecté.
 - vérifier que rôle est à 1 et pas à 0 si à 0 return (vide en front).
 - < supérieur à 0.

Détail d'un ticket créé dans trello pour ajouter un utilisateur à une chatRoom.

Méthodologie de Git et des branches

Afin de structurer notre projet, nous avons défini des conventions de nommage des branches et une méthodologie Git. Les règles s'appliquent autant à la partie API qu'à la partie Application.

Afin d'éviter les potentiels "merge conflicts", nous avons décidé de mettre en place une gestion spécifique des branches et des conventions de noms afin d'organiser le répertoire.

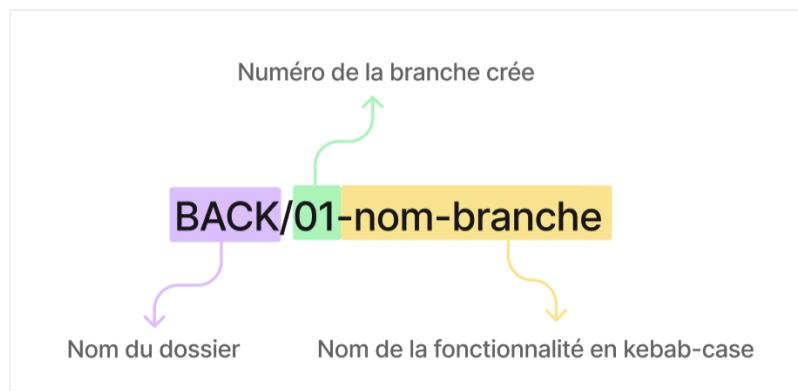
Méthodologie Git

Nous avions une branche par ticket qui correspondait au développement d'une fonctionnalité. Une fois la fonctionnalité complètement développée, nous fusionnions (merge) la branche correspondante vers une branche de "pré-production" - il y a eu plusieurs branches "pré-production" qui portent simplement le numéro du sprint en cours. Il a été néanmoins un peu compliqué pour nous de déterminer les fins des sprints et certaines difficultés nous ont bloqué à en déclencher de nouveaux. Enfin nous avions une branche dite de production (la branche "main") qui recevait principalement les merges considérés comme "finaux" - en d'autres termes, lorsque les différents merges sur un sprint fonctionnaient ensemble, on fusionnait la branche finie sur "main".

Méthodologie des branches

Il y avait une nomenclature spécifique pour les branches ainsi que pour les commits.

Branches : il devait y avoir la notion *feature* (si on ajoutait un nouvel élément) ou *fix* (si on réglait un bug sur une base déjà existante)/FRONT ou BACK en fonction du répertoire où l'on se trouve - le numéro du ticket créé dans l'ordre croissant - le nom donné à la fonctionnalité/au fix.

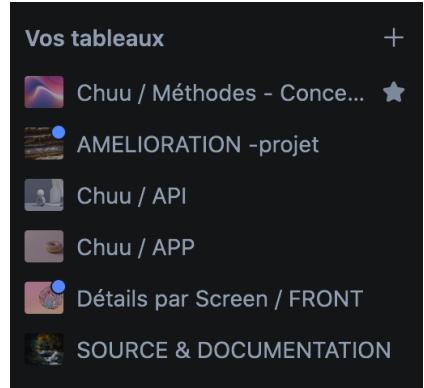


Commit : [FRONT ou BACK/numéro du ticket] ce qui a été fait dans le commit.

Trello

Le découpage des tâches a pu être réalisé grâce à l'outil Trello, organisé selon 4 grosses thématiques sous la forme de tableau (voir en annexe page 43) :

- La Conception
- Le Back/API
- Le Front
- Les détails du front page par page



Capture d'écran depuis trello

Tableau de conception

Dans ce tableau, nous avons détaillé les normes de travail liées aux tickets, aux branches et à Git. Ainsi chaque membre avait la possibilité de les consulter ou d'en créer de nouvelles.

Nous y avons aussi ajouté les tâches qui incombent à la conception de la base de données (méthode merise), aux maquettes (wireframe, haute fidélité) et aux idées de moodboard, à l'architecture, aux méthodologies de travail (gitflow, standardisation de la forme des tickets).

Tableau concernant la partie back

Dans ce tableau, nous avons listé les tâches de notre futur API. Chaque ticket représente une fonctionnalité et devra contenir le chemin de la futur route à établir entre la base de données et notre application.

Tableau concernant la partie front

Comme pour le tableau précédent nous avons défini un ticket par fonctionnalité réalisable par l'utilisateur qui serait géré dynamiquement dans le front.

Tableau concernant les détails du front

Dans ce tableau, nous avons tenu à soigner les détails de front non essentiel au bon fonctionnement de l'application mais qui rendent l'expérience utilisateur plus agréable.

★ Conception backend de l'application

La base de données

Ce projet nous a demandé de penser à la manière dont nous allions stocker les données générées par l'application. En effet nous allions devoir pérenniser l'accès aux Messages, aux canaux de discussions et aux informations des utilisateurs. Nos recherches sur le framework express nous ont fait découvrir que les bases de données les plus utilisées avec node Js sont les bases de données NoSql. N'ayant aucune contrainte concernant le choix de celle-ci, nous avons choisi d'utiliser une base de données relationnelle(SQL). Durant notre apprentissage, nous avons beaucoup travaillé sur les bases de données MySQL et nous souhaitions travailler cette technologie pour aller plus loin dans notre apprentissage de celle-ci.

Conception de la base de données

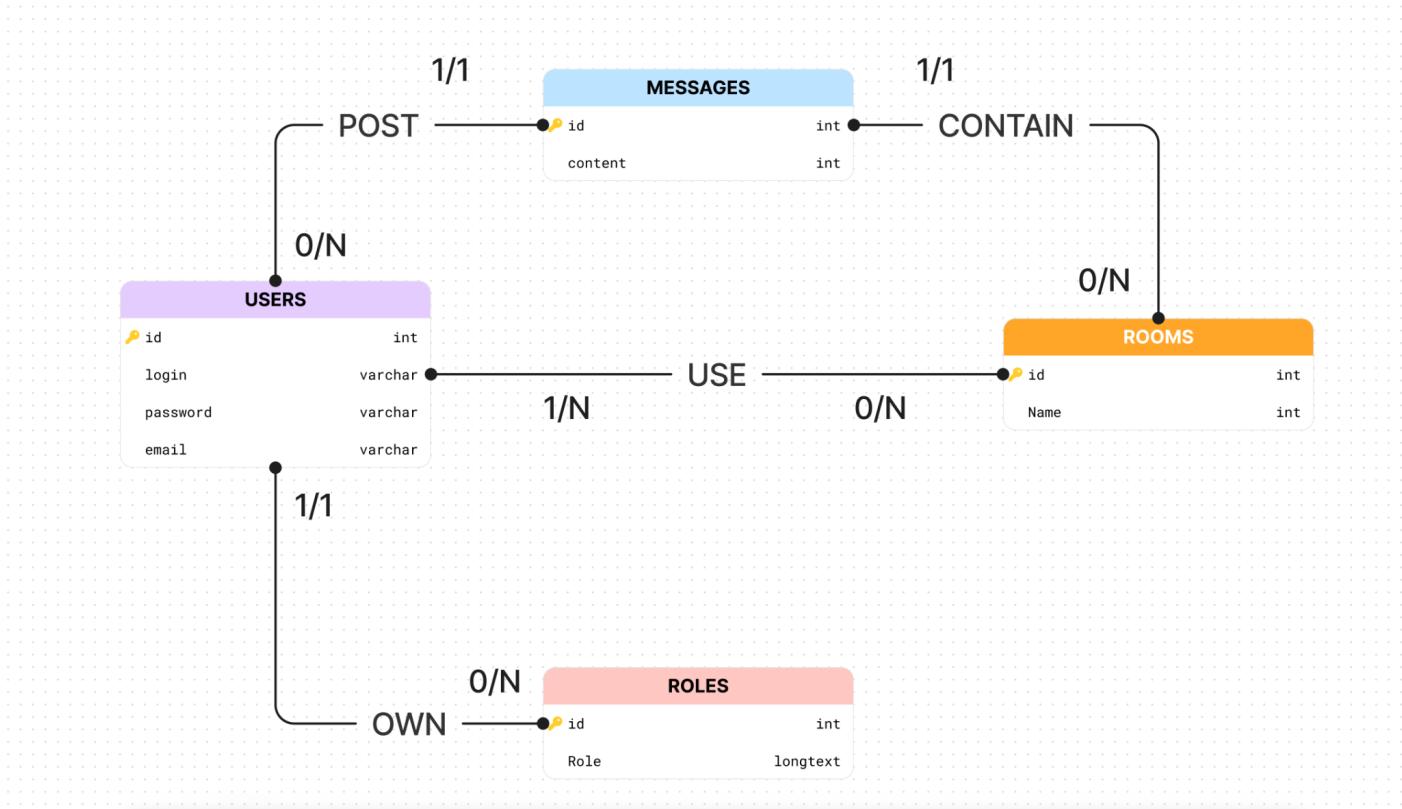
Pour concevoir ma base de données nous avons utilisé la méthode Merise. Dans un premier temps , nous avons fait un recueil de données .

Voici un aperçu des données pertinentes qui m'ont aidé à construire une représentation claire des besoins:

- Pour les messages: nous avons besoin de stocker le contenu , le nom de l'utilisateur qui a posté le message, la date de création du message, et la room dans laquelle le message a été posté.
- Pour les utilisateurs: nous avons besoin de stocker leur identifiant, un email, un mot de passe,.
- Pour les canaux de discussions: nous avons besoin du nom du canal uniquement.

Modèle conceptuel de données

La première étape a été la création du modèle conceptuel des données. Nous avons créé des entités en fonction du dictionnaire de données récoltés. Nous avons donc dessiné dans un premier temps nos entités en leur donnant un nom. C'est à cette étape de la conception que l'on introduit la relation verbale entre les entités.



Exemple :

Un utilisateur utilise au minimum 1 room et au maximum une infinité de rooms et une room est utilisée au minimum par 0 utilisateur ou au maximum par une infinité d'utilisateurs.

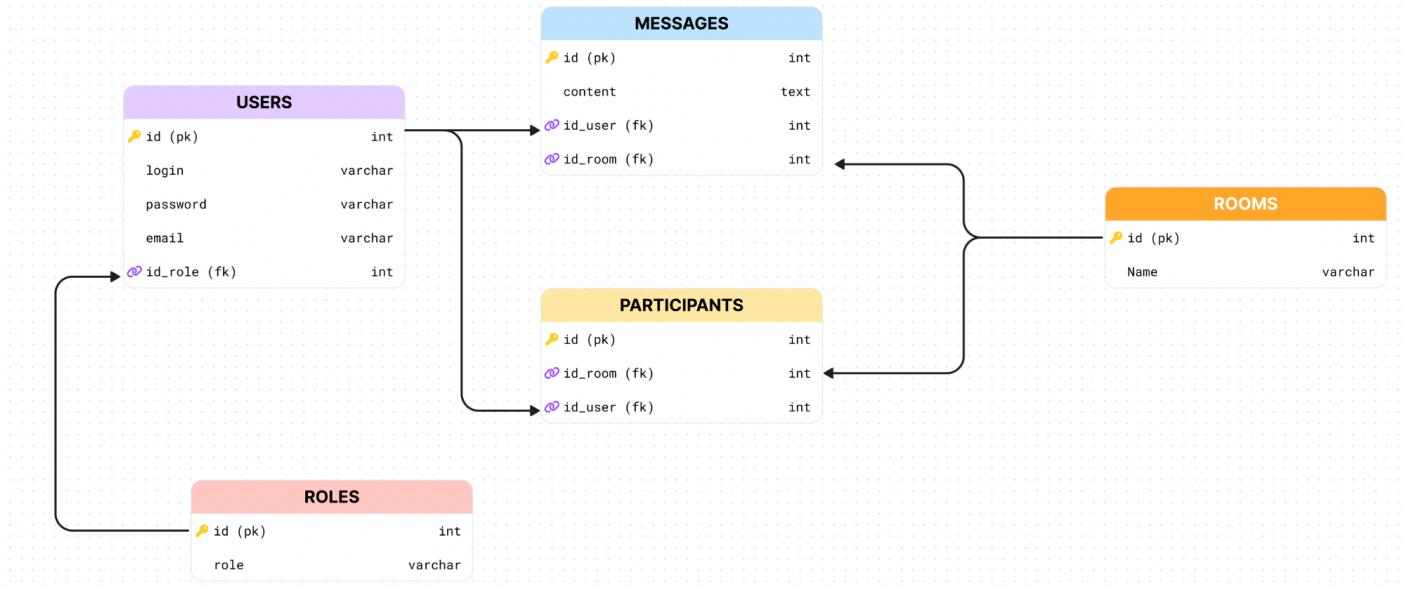
Modèle logique de données

Nous avons ensuite continué la création de ma maquette de base de données en créant le modèle logique de données.

Nous avons créé une clé primaire pour chaque entité qui permet d'identifier sans ambiguïté chaque occurrence de cette entité.

Ensuite, nous avons rempli chaque entité avec des attributs en reprenant les informations de notre recueil de données.

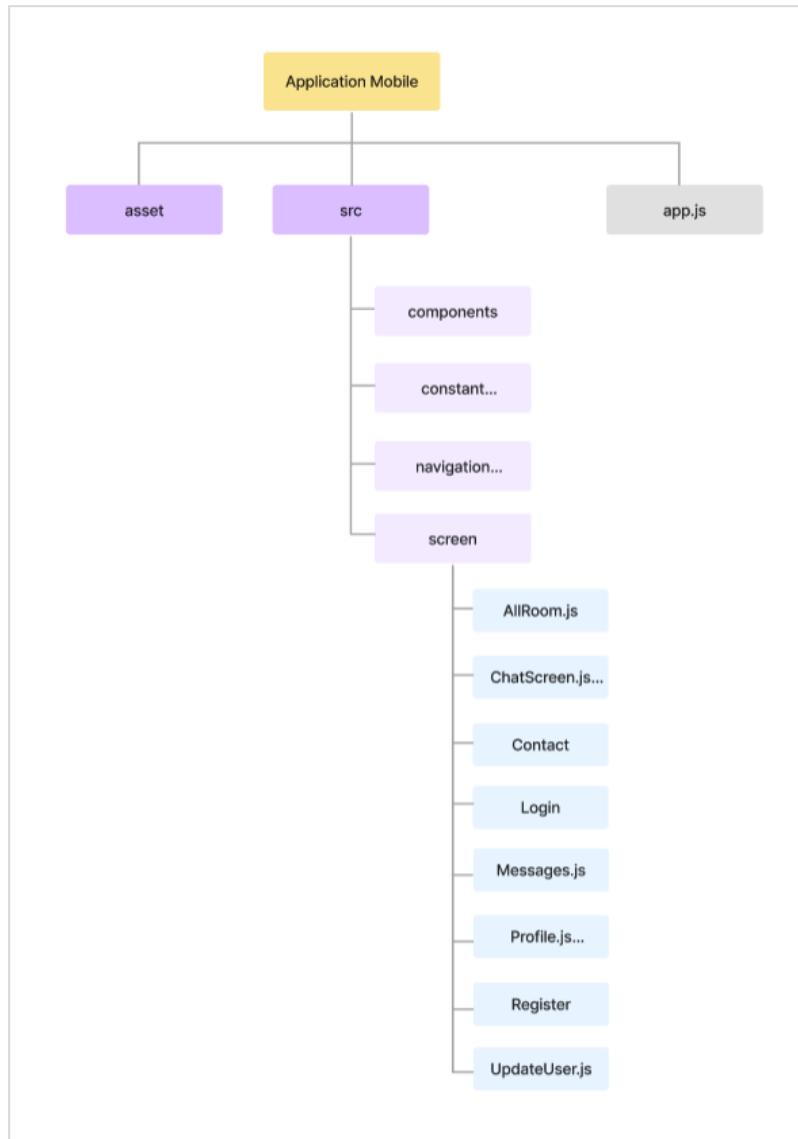
Nous avons procédé à la création des cardinalités de chaque entité.



Plan du Modèle Logique de Données

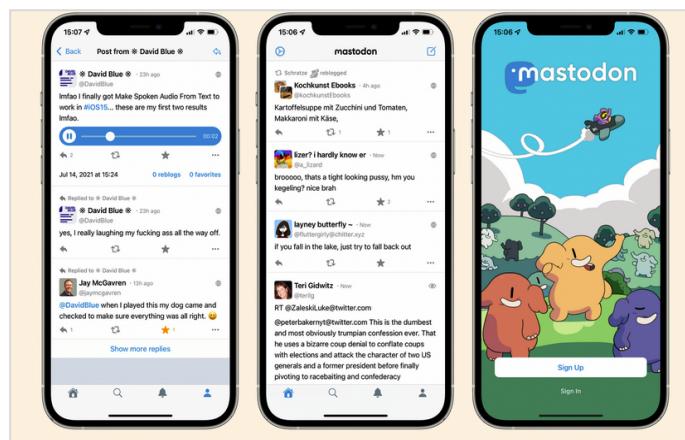
★ Conception du front-end de l'application

Arborescence du projet

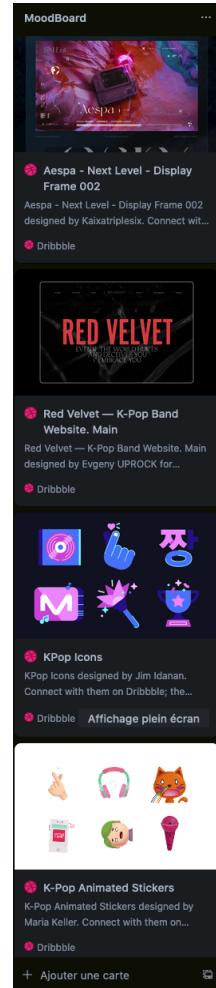


Moodboard

Pour guider la conception visuelle et trouver des inspirations, nous avions réalisé une petite collection d'images - un moodboard ainsi qu'une recherche explorative des autres applications de chat pour concevoir une UX/UI ancrée dans la réalité du marché.

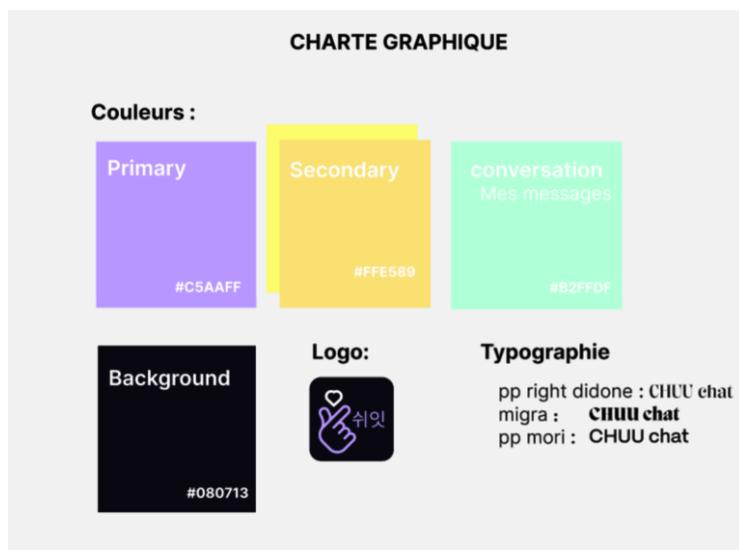


Interface application mastodon



Charte graphique

Après la création de l'arborescence du projet, nous avons établi la charte graphique et les couleurs. Nous avons ensuite commencé la création du logo à l'aide de photoshop et choisi les typographies



Maquettage

Les maquettes ont été créées à l'aide de Figma .
Elles se trouvent en annexe de ce dossier.

Wireframe

Avec toutes ces données récoltées, nous devions former le “squelette” de notre application, c'est-à-dire le wireframe aussi dit la maquette “low fidelity” (“low-fi”). Il permet la visualisation complète de la structure de l'application. Il ne doit pas être détaillé mais doit comprendre l'ensemble des éléments, leur agencement et l'articulation des uns avec les autres.

C'est une étape importante de la conception UX (user experience) qui permet de réfléchir sur le côté instinctif de l'application et de son accessibilité. Ainsi que d'avoir une première vue d'ensemble sur les différents composants à réaliser, ceux qui seront réutilisables par d'autres plus grands etc...c'est utile si l'on code en React.

Étant une application mobile, la conception a été, de manière logique, mobile first (ça aurait été le cas également si l'application pouvait être web).

Haute fidélité

Lorsque la maquette low-fidelity est validée, la réalisation de la maquette high-fidelity peut commencer. Cette étape consiste à merger l'UI (user interaction) et l'UX ensemble - afin que cette maquette puisse ressembler le plus possible au produit final. Elle permet un appui aux développeurs et un guide notamment pour la partie “front”.

La maquette finale devrait également comprendre un design system graphique. Ce dernier a pour objectif de réunir tous les éléments visuels, graphiques etc (couleurs, typographie, espacements entre les éléments...) dans un même système standardisé. Ce dernier peut, bien évidemment, évoluer mais il permet aux développeurs de construire des normes stylistiques à réutiliser dans tout le site/l'application afin d'avoir une homogénéité.

★ Développement du backend de l'application

Organisation

Notre back end a pour but d'être utilisé à la fois pour notre application web et notre application mobile. Nous avons décidé de créer une API afin de ne pas coder deux fois ma logique métier.

Dans le but de rendre notre backend plus efficace , nous nous sommes concentrés sur la logique et l'optimisation de notre code. Nous avons donc fait des recherches dans ce sens. Nous divisons donc nos programmes en différents modules, ainsi cela augmente la lisibilité du code et devient plus facile à maintenir pour les prochaines versions. J'évite les répétitions en créant des fonctions et des services.

Fonctionnement de l'API

Nous avons tenté de nous approcher au maximum d'une API RESTful, bien que nous n'ayons pas respecté totalement certaines conventions.

Une API REST (REpresentational State Transfer) est un interface de programmation d'application qui respecte les contraintes de l'architecture REST: **La séparation entre client et serveur, l'absence d'état de sessions, L'uniformité de l'interface, L'architecture en couches...**

Dans notre application, lorsque le client envoie une requête sur notre API, le routeur interprète l'URL et la méthode HTTP de cette dernière. Selon la route contenue dans la requête, une fonction (callback) du model/controller est appelée. Avant d'appeler le model/controller, la requête peut faire appel à des services techniques appelés "middleware" (également appelée en callback). Le model/controller exécute le code SQL. Puis la DB renvoie les données au model/controller, qui repassent par le routeur pour transmettre la réponse HTTP au client.

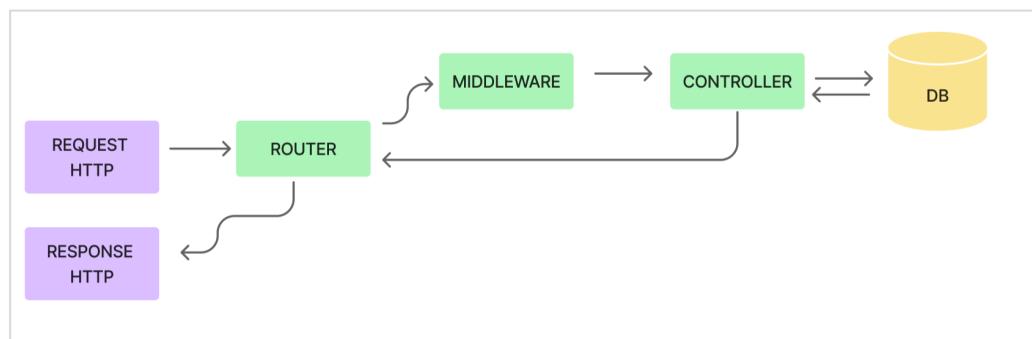


Schéma du fonctionnement de l'API

Les différentes méthodes HTTP utilisées dans ce projet :

- GET : Pour la récupération de données.
- POST : Pour l'enregistrement de données.
- PUT : Pour mettre à jour l'intégralité des informations d'une donnée.
- PATCH : Pour mettre à jour partiellement une donnée.
- DELETE : Pour supprimer une donnée.

Les différents statuts de réponse utilisés dans ce projet sont :

- 200 : OK
Indique que la requête a réussi à accéder aux ressources demandées.
- 201 : CREATED
Indique que la requête a réussi et qu'une ressource a été créée.
- 204 : NO - CONTENT
Indique que la requête a bien été effectuée et qu'il n'y a aucune réponse à envoyer.
- 400 : BAD REQUEST

Indique que le serveur ne peut pas comprendre la requête à cause d'une mauvaise syntaxe.

- 401 : UNAUTHORIZED

Indique que la requête n'a pas été effectuée car il manque des informations d'authentification.

- 404 : NOT FOUND

Indique que le serveur n'a pas trouvé la ressource demandée.

- 500 : INTERNAL SERVER ERROR

Indique que le serveur a rencontré un problème lors de l'exécution.

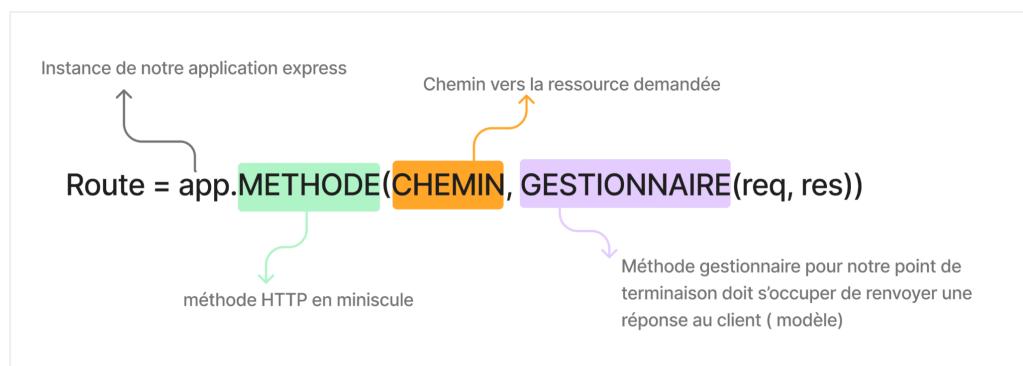
Middleware

Un middleware est une couche logiciel entre deux couches de logiciels. C'est une simple fonction qui a un rôle particulier. Dans le cas du framework Express c'est une fonction entre la requête et la réponse. Cette fonction a accès aux paramètres de la requête et de la réponse et donc il permet d'effectuer de nombreuses actions. Un de ses objectifs est la vérification des données envoyées par exemple dans le body des requêtes. Un middleware peut être appliqué à une unique route ou à plusieurs. De base, le framework possède quelques middleware mais il est possible d'en créer au besoin du projet.

Routage

Afin de mettre en place le routage, nous avons utilisé le routeur de Express.Js.

Express.Js met à disposition un middleware qui gère facilement le routage du projet. Pour déclarer une nouvelle route à express , on peut le résumer à cette exemple :



Comme expliqué plus haut notre but a été de rendre le code facilement modifiable et maintenable, nous avons donc divisé le routeur en 6 parties afin

de regrouper et trier les routes en faisant démarrer le chemin de ces routes par les segments d'url appropriés à nos besoins.

Voici l'implémentation de celles-ci:

```
//Users route
app.use("/users", users);
//Participants route
app.use("/participants", participants);
// Verify route
app.use("/connected", signIn, users);
//Admin route
app.use("/admin", [signIn, isAdmin], admin);
//Message route
app.use("/chat", chat);
//Rooms route
app.use("/rooms", rooms);
```

Extrait de code: implémentation des route

Le middleware use de cet exemple permet de spécifier le routeur qui doit être utilisé en fonction d'une route appelée.

Comme on peut le voir dans cet exemple, si l'URL '/users est appelée , notre routeur users est utilisé.

App est une instance de express et la fonction use permet d'associer un routeur dans lequel on trouve toutes les routes pour cette ressource.

Dans les routeurs une classe express.Router est instanciée. **C'est un middleware niveau routeur.** Un middleware niveau routeur fonctionne de la même manière qu'un middleware.

Grâce à la méthode router() de celui-ci le schéma de la route est défini. Ainsi je peux utiliser les méthodes HTTP: get, post, put, patch et delete afin d'effectuer différentes actions.

```
// Une route qui retourne toutes les infos de tous les utilisateurs
router.get('/all', getAllFromUsers);
//[BACK/05-post-user-to-a-room]: Permet d'ajouter un user à une room.
router.post('/room/:idRoom', signIn, addUserToRoom);
//[BACK/07] get the details from 1 user
router.get('/details/:userId', signIn, getUserDetails);
//[BACK/08-update-user]: Une route qui met à jour les info des utilisateurs
router.post('/update', signIn, updateUser);
//une route qui supprimer un utilisateur
router.delete('/supressAccount', signIn, supressAccount);
```

*Routes correspondant aux queries exécutables sur les utilisateurs
back/src/routes/users.js*

Dans l'exemple ci-dessus, nous avons instancié la classe Router de express. Grâce à cette classe, nous pouvons faire appel aux différentes méthodes HTTP.

Le premier paramètre de la fonction est l'URL, le deuxième est un middleware permettant la vérification de notre JWT token et le troisième est un gestionnaire qui est une fonction callback.

Connexion à la base de données

Nous avons créé un module dans le fichier database.js à la racine de l'API, qui établit la connexion à la base de données. Celui-ci est exporté et appelé dans toutes les fonctions exécutant du code SQL sur la base de données.

Dans cet exemple j'utilise un paquet npm 'mysql' qui permet de lier notre serveur Node.js à Mysql et ainsi pouvoir exécuter des requêtes.

Il existe plusieurs paquets permettant une connexion à une base de données, nous avons choisi mysql. C'est la solution la plus simple et rapide à mettre en place pour interagir avec une base de données MySQL en Node.Js.

Pour ce faire, il faut installer un paquet grâce à la commande suivante:

```
npm install mysql
```

Ensuite pour se connecter à la base, on appelle la fonction CreateConnection() qui permet d'indiquer l'host, le login de la bdd, le mot de passe et le nom de la base.

```
const mysql = require('mysql');
var connection
connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'root',
  database: 'chat',
  port: '3306',
});
```

Enfin, nous utilisons la fonction connect() pour nous connecter à la base de données, une exception (ou erreur) sera envoyée en cas d'erreur.

```
connection.connect(function(err) {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log(`Connected to database on port ${connection.config.port}!`);
});
module.exports = connection
```

Structuration du code

MVC

Le MVC (model, view, controller) est un design pattern permettant de structurer la manière de coder. Le modèle (côté serveur) détient les données, il permet de les récupérer de la base de données et de les gérer afin d'être envoyés vers le controller. Le controller, lui, traite ces données et les manipulent, servant ainsi d'intermédiaire entre le modèle et la vue.. Il fait référence à la logique du code. Une fois ces données maniées, le controller peut laisser place à la view (côté client) qui va présenter ces données à l'utilisateur.

Controller/Model

La logique et les queries de notre API se trouvent dans le même dossier. Il s'agit du dossier que nous avons nommé "controller". Ce dossier contient le code du modèle et du controller. Ce model/controller permet les interactions entre les routes et la base de données. On y trouve des fichiers qui permettent de découper les queries par entité afin d'organiser le code et de s'y retrouver. Il existe aussi un autre fichier qui permet la gestion des droits en fonction du rôle des utilisateurs.

Sécurité

Les API sont un moyen d'appeler des informations provenant de la base de données ainsi, il existe de nombreux risques.

Pour rendre notre API sécurisée nous avons mis en place plusieurs actions que je vous décris dans ce chapitre ainsi que les axes d'amélioration qui pourraient grandement augmenter la sécurité.

Les différents risques sont :

- Injections SQL : Les injections consistent à insérer dans les codes un programme un autre programme malveillant permettant ainsi d'attaquer directement une base de données et y prendre le contrôle. L'attaquant peut alors se servir librement de toutes les informations récoltées.
- Tables arc-en-ciel : méthode d'attaque cryptographique utilisée pour inverser des fonctions de hachage non salées, principalement utilisées pour stocker des mots de passe. Elle est utilisée dans le but de trouver un mot de passe correspondant à un hachage donné. (générant sur le hash)
- JWT tampering : fait référence à une attaque dans laquelle un JSON Web Token (JWT) est modifié de manière malveillante par une partie non autorisée. Cette altération peut se produire lorsqu'un attaquant modifie le contenu du JWT, tel que les données de l'utilisateur, les revendications ou d'autres informations stockées dans le token.(jwt.verify)

Injection SQL

Les queries sont sécurisées grâce aux placeholders ("?") qui permettent de remplacer les variables par des valeurs envoyées en argument de la méthode query. On dit que les paramètres sont liés (bind) aux placeholders.

```
const verifyParticipation = `SELECT id_room FROM participants WHERE id_user = ? AND id_room = ?`;
db.query(verifyParticipation, [req.user.id, req.params.idRoom], function (error, dataIdRoom) {
```

Les tables arc-en-ciel

Les mots de passe des utilisateurs ne sont pas stockés en dur dans la base de données . Pour sécuriser les mots de passe, j'utilise la dépendance bcrypt. Nous avons utilisé la méthode **genSalt()** pour générer une chaîne de caractère avant le hachage. Ensuite nous utilisons la fonction **hash()** afin de chiffrer le mot de passe nous récupérons le mot de passe en base, puis nous vérifions la correspondance du mot de passe saisi dans le formulaire de login avec celui qui est haché grâce à la méthode **compareSync()**.

```
// Hash password
const salt = await bcrypt.genSalt()
const hash = await bcrypt.hash(password, salt);

if(bcrypt.compareSync(password, results[0].password)) {
```

Attaques par dictionnaires

Une attaque par dictionnaire (dictionary attack) est une méthode utilisée pour tenter de deviner un mot de passe en utilisant un ensemble de mots couramment utilisés, des mots du dictionnaire ou des combinaisons de mots (regex)

```
const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)[a-zA-Z\d]{8,}$/;
//minimum 8char, 1maj, 1minuscule ett 1 chiffre
```

JWT

Sécurité du JWT

1- Le token permet de valider des requêtes sans utiliser de mot de passe pour vérifier l'utilisateur en faisant circuler celui-ci de nombreuses fois entre le client et le serveur, ce qui exposerait l'application et la base de données à des failles.

2- Durée de validité limitée : Les JWT peuvent avoir une durée de validité limitée, définie par une date d'expiration. Cela limite la fenêtre d'opportunité pour une utilisation abusive ou la récupération d'un JWT volé.

3- Intégrité des données : La signature du JWT garantit l'intégrité des données. Si le contenu du JWT est modifié après avoir été émis, la signature devient invalide et le token est considéré comme non fiable.

4- Authentification de l'utilisateur grâce au payload: ces informations peuvent être vérifiées par le serveur à chaque demande pour s'assurer que l'utilisateur est authentifié et a les autorisations nécessaires pour accéder à certaines ressources.

Qu'est ce que le JSON Web Token?

Le JSON Web Token est une technologie qui peut être mise en place pour répondre à différents besoins. Il doit être pensé dès les plans de conception car certaines fonctionnalités seront intégrées très tôt dans la réalisation du projet. Le JWT résout un problème lié à la scalabilité d'une application et la sécurité des données utilisateurs.

Son anatomie se découpe en 3 parties:

- **le header** : identifie quel algorithme a été utilisé pour générer la signature.
- **le body/ le payload** : est la partie qui contient les informations de l'utilisateur, sous forme de chaîne de caractères encodée en base 64.
- **la signature**: Elle est créée à partir du header et du payload générés et d'un secret. Une signature invalide implique systématiquement le rejet du token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

La signature du jeton a une importance fondamentale, il sert à vérifier que les informations connues sont inchangées.

Le JWT est un jeton qui permet l'échange des informations sur l'utilisateur de manière sécurisée. C'est une méthode de communication entre deux parties. Il permet donc:

- l'authentification des clients au service mis à disposition.

- d'éviter la montée en charge des serveurs en vérifiant l'état de connexion des clients sans devoir partager cet état de connexion entre les différents serveurs. C'est ce qui permet à notre API d'être "stateless". Sans le JWT et afin de vérifier cet état, il faudrait exécuter un certain nombre de requêtes redondantes. En effet, l'état de connexion est stocké directement dans le Token. Il n'est donc pas nécessaire de le stocker en mémoire dans une base (distante ou locale), ni dans des cookies de sessions.

Comment fonctionne le JSON Web Token?

Dans le but d'effectuer une authentification sécurisée sur notre projet nous avons mis en place le Jeton Web Token.

Le token et le refresh token sont alors générés lors de cette l'authentification à la connexion d'un utilisateur sur l'application.

Le refresh token est un jeton qui expire selon une fréquence donnée, dans notre cas il s'agit d'un mois. Tous les mois l'utilisateur doit se reconnecter à l'application pour en générer un nouveau.

Lorsqu'un utilisateur essaie de se connecter à son espace , une demande est envoyée au serveur. Si les informations envoyées sont correctes , le serveur renvoie une réponse sous format JSON dans lequel se trouve le jeton . Celui-ci contient des informations concernant la personne connectée (son id , son mail, son username et son rôle).

Le client enverra ce jeton avec toutes les requêtes (modifier un profil, poster un message dans le chat) qui suivront.

Le refresh token

En ce qui concerne le refresh token, notre implémentation est à repenser. Lorsque nous envoyons le refresh au back nous vérifions s'il est toujours valide en le décodant avec jwt.decoded() or c'est le token qui devrait être décodé. Le token expire avant le moment où il est censé être régénéré par le refresh token

Gestion des Droits

Le middleware isAdmin nous permet de vérifier le rôle de l'utilisateur qui envoie la requête côté client à l'aide du payload de son token unique. Le contenu du payload est récupéré et comparé avec le nombre entier "2" qui est défini en base de données comme étant le rôle des administrateurs. Le middleware est appelé sur toutes les routes nécessaires à une requête de ou des administrateur-s.

```
var admin = require('./src/routes/admin');
```

back/index.js

```
//Admin route
app.use('/admin', [signIn, isAdmin], admin)
```

back/route/admin.js

```

exports.isAdmin = (req, res, next) => {
  if (parseInt(req.user.id_role) === 2) {
    return next();
  } else {
    return res.status(400).json({ message: "You are not an administrator" });
  }
}

```

back/middleware/isAdmin.js

Recherches Anglophone

Afin de simuler un environnement de pré-production pour développer notre application mobile, nous avons installé expo en suivant la documentation qui est exclusivement en anglais:

The screenshot shows the official Expo documentation website. It features two main sections: '1 Initialize a new project' and '2 Start the development server'. Each section includes a terminal-like interface for running commands. A note in the first section provides an alternative command using the --template option. The second section notes that the Metro Bundler starts an HTTP server for serving the app.

1 Initialize a new project

To initialize a new project, use `create-expo-app` to run the following command:

```
Terminal Copy
# Create a project named my-app
- npx create-expo-app my-app
# Navigate to the project directory
- cd my-app
```

ⓘ You can also use the `--template` option with the `create-expo-app` command. Run `npx create-expo-app --template` to see the list of available templates.

2 Start the development server

To start the development server, run the following command:

```
Terminal Copy
- npx expo start
```

When you run the above command, the Expo CLI starts [Metro Bundler](#). This bundler is an HTTP server that compiles the JavaScript code of your app using [Babel](#) and serves it to the Expo app. See how [Expo Development Server](#) works for more information about this process.

Capture d'écran de la documentation sur le site d'expo

Exemple Fonctionnalité: Ajout d'un utilisateur à une room depuis l'API

Pour ajouter un utilisateur à une room sur notre API, il faut envoyer une requête HTTP de type POST à la route :

"{hostname}/users/room/:idRoom" : **hostname** désignant le nom de domaine de l'API et **:idRoom** l'id de la chatroom auquel l'utilisateur veut s'ajouter. Il faudra alors préciser dans le body (via la clé **id_room**) l'id de la room dans laquelle l'utilisateur veut se rajouter.

```
var users = require('./src/routes/users');

//Users route
app.use("/users", users);
```

Screen 1 et 2 : index.js - Point d'entrée de l'API

On fait donc appel au middleware niveau routeur de Express. En appelant cette fonction, le routeur associe le segment "/users/" de l'URL au fichier JS './src/routes/users'.

```
//[BACK/05-post-user-to-a-room]: Permet d'ajouter un user à une room.
router.post('/room/:idRoom', signIn, addUserToRoom);
```

screen 3 : users.js - Suite du routeur

C'est dans ce fichier que sont décrites toutes les routes concernant les utilisateurs. La fonction ci-dessus appelle la méthode router.post() qui indique qu'il s'agit d'une requête POST. Les 3 paramètres représentent :

- La suite de la route (à partir du segment "/users"). La variable idRoom de la route est indiquée par " : ".
- La fonction middleware signIn() (appelée en callback) qui sera exécutée dans un premier temps afin de vérifier l'état de connexion de l'utilisateur.
- La fonction du controller **addUserToRoom** qui effectue les vérifications puis l'insertion en base de données.

```

exports.signIn = (req, res, next) => {
    const tokenToUse = req.headers.token1;
    const tokenRefresh = req.headers.refreshtoken;
    try {
        const mySecret = "mysecret";
        const decoded1 = jwt.verify(tokenToUse, mySecret);
        req.user = decoded1;
        const decoded2 = jwt.decode(tokenRefresh)
        var now = new Date().getTime() / 1000;
        try {
            const decoded2 = jwt.verify(tokenRefresh, mySecret)
            var now = new Date().getTime() / 1000;
            if (now > decoded2.exp) {
                /* expired */
                // le token est disponible ds le scope grace
                // au callback dans le refreshToken du usersController
                return refreshToken(decoded1.id, token => {
                    res.status(401).send(token)
                });
            }
            next();
        } catch (err) {
            return refreshToken(decoded1.id, token => {
                res.status(401).send(token)
            })
        }
    } catch (err) {
        // console.log('auth.js : ', err);
        return res.status(401).send(err);
    }
};

```

Screen 4 : auth.js - Module de vérification des token de connexion

A l'appel de la fonction middleware signIn(), les headers de la requête HTTP (token1 et refreshtoken) sont vérifiés à l'aide de la fonction jwt.verify, ainsi que du secret. Les scénarios envisagés sont :

- Si verify ne parvient pas à vérifier le token1, une exception levée, catch, puis renvoyée en réponse avec le statut 401 (Unauthorized) ce qui signifie que l'utilisateur n'est pas autorisé à poursuivre.
- Si c'est le refreshtoken qui n'a pas pu être vérifié, ce dernier est renvoyé en réponse avec le statut 401.
- Sinon, on compare la date d'expiration du tokenrefresh (qui est contenue dans son payload, une fois décodé) avec la date d'aujourd'hui.
- Si le token est expiré, on renvoie le token en réponse avec le statut 401.

- Sinon, la fonction next() est appelée, ce qui termine l'exécution du middleware afin d'appeler la fonction suivante appelée dans le routeur (addUserToRoom).

```
const addUserToRoom = (req, res) => {
  const verifyRoles = `SELECT role FROM users INNER JOIN roles
  ON roles.id = users.id_role WHERE users.id = ?`
  db.query(verifyRoles, [req.user.id], function (error, data) {
    if (data.length !== 0) {
      if (data[0].role !== "ban") {
        const verifyParticipation = `SELECT id_room FROM participants WHERE id_user = ? AND id_room = ?`
        db.query(verifyParticipation, [req.user.id, req.params.idRoom], function (error, dataIdRoom) {
          if (dataIdRoom[0] == undefined) {
            var insertUser = insertToRoom(req.body.id_room, req.user.id);
            res.status(200).send({ message: 'Request succeed.' })
          } else res.status(400).send(
            { message: 'The id user '+[req.user.id]+' is already related to the id room '+[req.body.id_room]+'. ' }
          );
        })
      } else res.status(400).send({ message: 'You were ban of this room.' })
    }
  })
}
```

Screen 5 : userController.js - Vérification de la requête HTTP et exécution des query SQL

A l'appel de la fonction **addUserToRoom()** du **userController**, une première query SQL de vérification du rôle de l'utilisateur permet de savoir si ce dernier est banni à l'aide d'une jointure interne entre la table role et la table user.

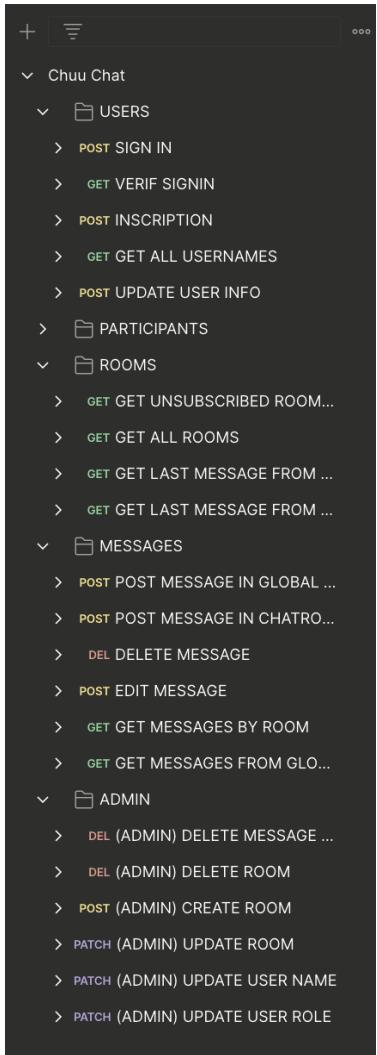
- Si l'utilisateur est banni, l'API retourne une réponse de statut 400 avec un message indiquant que l'utilisateur est banni.
- Sinon, une seconde query SQL de vérification est exécutée afin de vérifier que l'utilisateur n'est pas déjà présent dans la room dans laquelle il essaie de se rajouter.
- Si il est déjà présent, un message lui notifie dans une réponse avec le statut 400

```
const insertToRoom = (id_room, id_user) => {
  const sql = `INSERT INTO participants (id_room, id_user) VALUES (?,?)`
  db.query(sql, [id_room, id_user], function (error, data) {

    if (error) {
      throw (error)
    }
    else {
      return data
    }
  })
}
```

- Sinon, on appelle la fonction auxiliaire insertToRoom() qui exécute la query SQL d'insertion du couple (id_user,id_room) dans la table participants.

Tests



Le testing représente une étape importante (voire obligatoire) du développement. En effet, plusieurs types de tests existent afin de s'assurer que le développement de nouvelles fonctionnalités se déroule comme prévu, et n'impacte pas négativement les fonctionnalités précédemment développées :

- les tests unitaires : consistent à tester de manière individuelle et isolée chaque fonction du projet en comparant le résultat effectif et le résultat attendu en fonction d'un jeu de données test prédéfinies.
- les tests fonctionnels : consistent à tester une fonctionnalité complète afin de vérifier un scénario d'utilisation de bout en bout (ex = connexion d'un utilisateur à l'application)
- les tests de non régression : consistent à spécifiquement vérifier que la version en cours n'a pas généré de régression par rapport à la version précédente.

Quelque soit le type de test, ces derniers peuvent être manuels (c'est à dire exécutées un à un par une personne) ou automatisés via des outils qui vont suivre un scénario précis, valider ou non le test selon des critères qu'on aura prédefinis, puis générer un rapport de test de chacunes des fonctions, ou fonctionnalités selon le type de test.

Au vu de la charge de travail et du planning de notre projet, nous avons finalement été contraints de ne pas utiliser d'outils de test automatisés. Nous avons donc procédé à un testing fonctionnel manuel de notre API grâce à l'outil **Postman**.

Dans un premier temps, il a fallu créer la collection Postman qui recense l'ensemble des appels possibles vers notre API.

Dans le cadre de tests manuels, il est quasi indispensable de tenir à jour un cahier de recettes. Il s'agit d'un document qui recense l'ensemble des tests, leur résultats ainsi que des informations complémentaires sur le contexte et l'exécution des tests.

Afin de fluidifier le processus de test, nous avons créé 2 environnement Postman contenant le nom de domaine de l'API ainsi que les informations de l'utilisateur connecté. Cela permet alors de passer d'un utilisateur à un autre sans avoir à modifier la collection .

L'outil "Test" de Postman nous a ainsi permis de créer un script qui enregistre les tokens de connexion de l'utilisateur dans les variables de

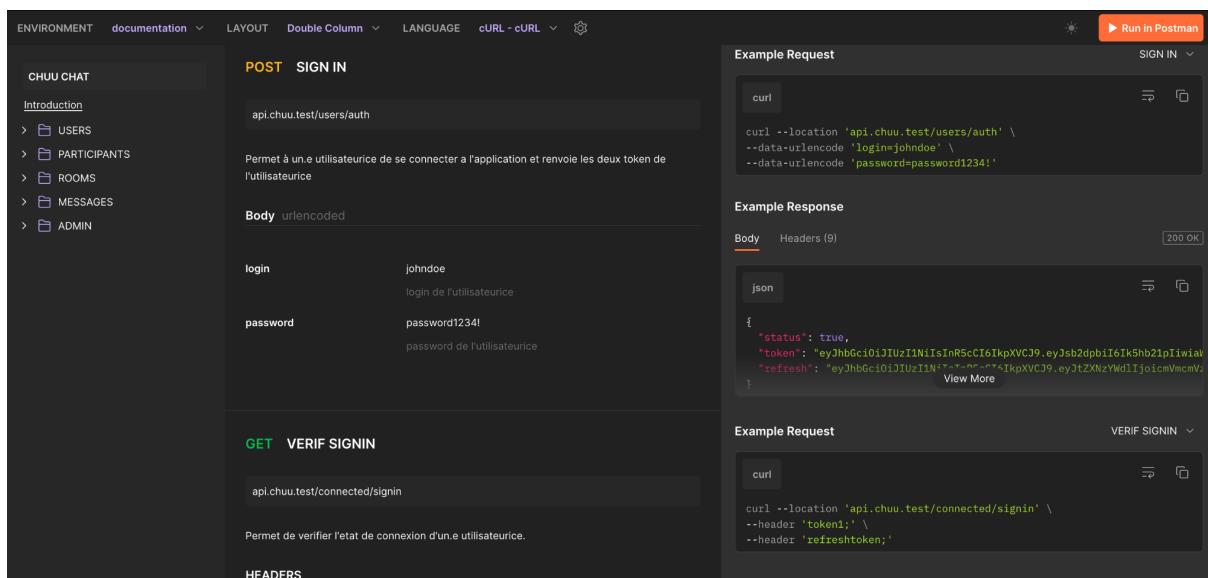
l'environnement (voir en annexe page ...?). Ce script est donc configuré pour être automatiquement exécuté après chaque appel de la requête de connexion.

Une fois tout cela mis en place, il a fallu lancer chaque requête une à une, puis comparer le résultat de la requête (response + contrôle des données en base) au résultat attendu, puis le consigner dans le cahier de recettes.

Documentation

La documentation de l'API a également été générée grâce à Postman. Une fois la collection créée, il a été assez rapide de générer une documentation à partir de cette dernière. Il a donc fallu rajouter une brève description pour chaque requête ainsi que pour leurs paramètres et les headers nécessaires. Nous avons également enregistré un exemple de réponse pour chaque appel à l'API afin que Postman les mette en forme dans la documentation. Enfin, nous avons généré la page web de documentation qui recense les requêtes ainsi que toutes ces informations dans un document. Ce document fournit également un exemple de code qui permettrait d'envoyer ces requêtes dans plusieurs langages (par exemple : cURL, Ajax, Fetch.js, etc...)

La documentation de notre API est disponible à cette URL :
<https://documenter.getpostman.com/view/8533229/2s93mBxJxX>

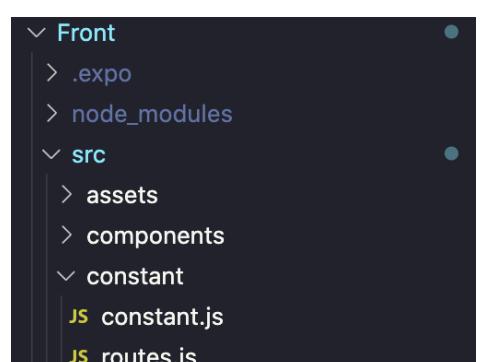


Capture d'écran d'une page de la documentation de l'API

★ Développement du front-end de l'application

Arborescence

- Dans le dossier src du projet on y trouve tous les dossiers essentiels au



projet.

- Dans le dossier assets, on trouve des sous dossiers qui contiennent les icônes du projet et les images qui habillent le front.
- Dans le dossier components on y trouve tous les composants réutilisables et d'autres qui ne le sont pas.
- Dans le dossier constant, on trouve le fichier de constant.js qui nous permet de stocker les constantes du projet ainsi que le fichier routes.js qui permet d'établir le nom des routes pour la navigation.
- Dans le dossier navigation se trouve la navigation.
- Dans screen , se trouve tous les écrans de notre application mobile.

Pages et composants

Un composant permet de pouvoir découper une page en éléments indépendants et réutilisables.

Conceptuellement, les composants sont comme des fonctions JavaScript. Ils acceptent des entrées quelconque (appelées « props ») et renvoient des éléments React décrivant ce qui doit apparaître à l'écran.

Le principe de react native est d'écrire un maximum de composants réutilisables. Ainsi on évite de dupliquer le code.

Voici un exemple d'un composant réutilisable. Il permet l'affichage d'un bouton qui envoie le texte émis par un utilisateur dans le bon canal de chat. Pour éviter les répétitions, j'ai créé un composant nommé ButtonMessage.

```

export default function ButtonMessage(props) {
    const handleSubmit = () => {
        SecureStore.getItemAsync('token1').then((rest) => {
            SecureStore.getItemAsync('refreshToken').then((res) => {
                if (res) {
                    if (props.text) {
                        axios.post(`${API}/chat/${props.idRoom}`,
                            JSON.stringify({content: props.text,}),
                            {headers: {
                                'Content-Type': 'application/json',
                                token1: rest,
                                refreshToken: res
                            }
                        }).then(res => {
                            props.setText('');
                        }).catch(e => {
                            console.log('handleSubmit:', e)
                        })
                    }
                }
            })
        })
    }
}

```

Front/componants/buttonMessage.js

Je passe en paramètre de ma fonction ButtonMessge() ce que je vais vouloir envoyer à notre API pour enregistrer les informations en base de données comme illustré ci-dessus. Nous utilisons la méthode **axios** de la **librairie Axios**. Nous plaçons en premier paramètre de cette fonction le chemin vers l'API et dans le second paramètre, sous le format Json, le contenu de l'input qui permet à un utilisateur de poster un message.

```

    return (
        <React.Fragment>
            <TouchableOpacity
                style={styles.container}
                onPress={() => {handleSubmit()}}
            >
                <Image
                    style={styles.img}
                    source={require('../assets/icons/right-arrow.png')}
                />
            </TouchableOpacity>
        </React.Fragment>
    )
}

```

Front/composants/buttonMessage.js

Ci-dessous, voici le retour de la fonction. Je récupère au niveau de l'attribut onPress le contenu de ma constante handleSubmit. Ci-dessous, un aperçu de mon composant depuis le device.



Le rond vert avec la flèche représente le composant ButtonMessage depuis le front de l'application.

Sécurités

Secure Store

Le secure store est une API fournie par Expo à installer côté client et permet d'utiliser les services afin de sécuriser les données sensibles qui transitent entre notre API et notre front. Le secure store stocke dans le keystore ou keychain (en fonction de l'OS) le token qui vient de l'API en le chiffrant. Le token est donc stocké en local sur le device de l'utilisateur grâce à la fonction `setItemAsync()`.

```
const connect = () => {
  if (login !== '' && password !== '') {
    axios.post(API + '/users/auth', {
      login: login,
      password: password
    })
      .then(function (response) {
        setLogin('');
        setPassword('');
        const token = response.data.token;
        const refresh = response.data.refresh;
        SecureStore.setItemAsync('token1', token).then(() => {
          SecureStore.setItemAsync('refreshtoken', refresh).then(() => {
            // console.log('co')
            navigation.navigate(ROUTES.HOME, { screen: rooms.length > 1 ? ROUTES.FEED : ROUTES.CHATROOMS })
          })
        })
      })
  }
}
```

stockage et chiffrement du token dans le secure store lors de la connexion des utilisateurs dans: Front/composants/login.js

On implémente donc le secure store au moment où la route de la connexion des utilisateurs est appelée dans le front. Pour récupérer le contenu du token on utilise la fonction `getItemAsync()` qui déchiffre les token et les renvoie au back au moment des call API.

```
useEffect(() => {
  SecureStore.getItemAsync('token1').then((token) => {
    SecureStore.getItemAsync('refreshtoken').then((refresh) => {
      axios({
        method: 'get',
        url: `${API + uri}`,
        headers: {
          'Content-Type' : 'application/json',
          token1: token,
          refreshtoken: refresh
        }
      }).then((response) => {
        setContacts(response.data)
      })
    })
  })
})
```

Exemple de récupération du token à chaque requête client dans Front/composants/ contact.js

Problématiques rencontrées

Socket.io

Afin de mimer la messagerie instantanée, nous avons mis en place Socket.io. Il dispose d'une connexion bidirectionnelle signifiant que le serveur peut pousser des informations (ici messages) aux clients connectés dans la chatroom.

Il faut commencer par installer socket.io via npm dans le projet.
Par la suite, nous avons dû faire une connexion du côté serveur.

```
const io = require('socket.io')(server)
```

```
io.on('connection', (socket) => {
  console.log('a user connected');
  socket.on('disconnect', () => {
    console.log('user disconnected');
  });
  socket.on('joinIn', (id_room) => {
    socket.join(id_room);
  })
});

server.listen(port, () => {
  console.log(`Socket.IO server running on port :${port}`);
});
```

Cette connexion de socket io se réalise à chacune des rooms qui sont identifiées par leur id grâce à “.join”.

```
// socket côté query avec la clef newMessage
```

Pour que le serveur connaisse les id des rooms, on appose à socket un “évènement” nommé “joinIn” - dans l’exemple au-dessus, il est écouté via le “.on”. Cet événement est émis du côté client dans notre composant ChatRoom qui détient l’ensemble des id des rooms. C’est également dans ce composant qu’on appelle celui des Messages à qui on fait passer la connexion socket en props.

Socket.io-client

```
import { io } from 'socket.io-client';
```

On importe la librairie côté client

```
const [socket, setSocket] = useState(io("http://localhost:3000"));

useEffect(() => {
  socket.emit('joinIn', route.params.id_room)
```

On instancie l’objet “io” et on lui passe par défaut l’adresse url du serveur. On stocke cet objet dans la constante “socket”. On accroche l’événement .emit à ce nouvel objet qui permet de joindre l’id de la room dans laquelle l’utilisateur se trouve au serveur afin d’insérer le message en base de données dans la bonne room.

```
<Messages style={{ height: '40%' }} idRoom={route.params.id_room} socket={socket} />
```

On passe les props au composant Message grâce aux attributs idRoom et socket. Puis, dans le composant Message, on écoutait l’événement “newMessage”.

```
props.socket.on('newMessage', message => setMessages(messages => [...messages, message]));
```

Ce dernier était émis depuis notre controller de messages dans la fonction qui s’occupait d’envoyer un message dans la base de données.

```
io.to(parseInt(req.params.roomId)).emit('newMessage', message)
```

Etant donné que nos messages étaient un useState, une fois qu’on récupérait l’entièreté des messages, on avait un setter qui en faisait un array qu’on pouvait parcourir et donc afficher sur l’écran par la suite.

```
getMessages(data => {
  setMessages(data);
  console.log('getMessages: ', setMessages);
})
```

Aussi, on disait à socket que sous l’événement “newMessage”, il devait re-set un nouveau tableau avec en plus l’information “fraîchement” envoyée. Ce qui faisait que lorsqu’on “map”-pait notre array de messages, on avait toujours également le dernier message envoyé dans la chatroom.

Socket s'occupait donc d'éviter un refresh qu'on aurait dû avoir en temps normal - de cette manière il avait un accès au côté serveur et envoyait en instantané à tous les clients participants à la room.

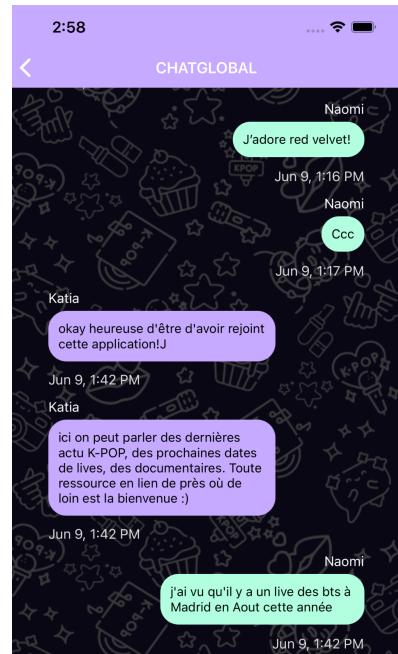
Navigation imbriquée

La navigation avec React Native fait référence à la gestion de la navigation entre les différentes vues ou écrans au sein d'une application mobile développée avec React Native. Nous avons choisi d'utiliser la bibliothèque de navigation React Navigation.

Pour mettre en place la navigation nous avons instancier les objets grâce aux méthodes `createBottomNavigator()` et `createStackNavigator()` et nous les avons stocké dans les variables Tab et Stack.

Voici un aperçu des concepts clés de la navigation avec React Native utilisés dans notre projet:

- Stack Navigator : Il permet de gérer la navigation en pile (stack) où les écrans sont empilés les uns sur les autres. Lorsqu'un nouvel écran est affiché, il est ajouté au sommet de la pile, et lorsque l'utilisateur appuie sur le bouton de retour, l'écran supérieur est retiré de la pile. Cette vue montre un exemple de navigation via à la flèche de retour vers le screen précédent. Pour empiler les vues nous avons englobé le composant `Stack.Navigator` autour du composant `Stack.Screen`



Afin d'augmenter l'expérience utilisateur nous avons implémenté une tab bar.

- Tab Navigator : Il permet de créer une barre de navigation avec des onglets en bas ou en haut de l'application. Chaque onglet est associé à un écran différent, et les utilisateurs peuvent passer d'un onglet à un autre pour afficher différents contenus.

```

const ChatRoomStack = () => {
  return (
    <Stack.Navigator>
      <Stack.Screen
        name = {ROUTES.CHATROOMS}
        component = {AllRooms}
        options={{{
          headerStyle: {backgroundColor: '#C5AAFF'},
          headerTintColor: 'white',
          headerBackTitleVisible: false,
          headerShadowVisible: false,
          headerLeft: ()=> null
        }}}
      />
    </Stack.Navigator>
  )
}

```

Le return de la fonction TabBar qui utilise la navigation en tabulation (Tab navigator) , inclue dans l'attribut "componant" le return de la fonction "ChatRoomStack" qui utilise elle-même la navigation en pile (stack navigator).

```

return (
  <Tab.Navigator {...{screenOptions}} >
    <Tab.Screen name={ROUTES.CHATROOMS} component={ChatRoomStack} options={{{
      tabBarIcon: ({focused}) => (
        <View style={{alignItems: 'center', justifyContent: 'center'}}>

```

★ Conception de l'espace administrateur

Comme énoncé plus haut , notre projet est composé d'une partie administration. Celle-ci est gérée grâce à un site web.

User Story

En tant qu'administrateur de l'application web de gestion de l'application mobile, je souhaite pouvoir accéder à l'application en utilisant mes informations d'identification afin d'effectuer différentes tâches de gestion. Je veux avoir la possibilité de supprimer les messages d'utilisateurs inappropriés ou indésirables pour maintenir un environnement sain dans les discussions. Je veux également pouvoir créer de nouveaux salons de discussion pour permettre aux utilisateurs de participer à des conversations sur différents sujets. En tant qu'administrateur, je devrais être en mesure de supprimer les salons de discussion existants si nécessaire. J'aimerais pouvoir renommer les noms des salons de discussion pour les mettre à jour ou les adapter en fonction des besoins. En plus de cela, je souhaite avoir la capacité de modifier les noms des utilisateurs en cas de changement de pseudo ou d'erreur dans

l'entrée des données. Je veux également pouvoir modifier le rôle des utilisateurs, par exemple, passer un utilisateur de "membre" à "banni" ou "administrateur". Enfin, en tant qu'administrateur, je devrais pouvoir modifier mon propre espace personnel, y compris mon nom d'utilisateur, ma photo de profil et d'autres informations pertinentes.

Langage

Afin de rester cohérent avec la stack techno mise en place, nous avons choisi de développer en javascript natif. L'avantage d'utiliser à la fois javascript pour le frontend et le backend se trouve également dans la fluidité du déploiement sur les différentes plates-formes. Cela simplifie la gestion de l'infrastructure et réduit les dépendances de déploiement, ce qui facilite la mise en production de l'application.

Pour passer nos call API, et effectuer des requêtes réseau puis gérer les responses, nous avons utilisé la méthode `fetch()` de l' interface JavaScript native Fetch API.

Conception du frontend du site web

Charte graphique

Dans l'optique d'être cohérent, notre site web utilise la charte graphique de l'application mobile présentée plus haut.

Maquettage

J'ai procédé au maquettage de notre application web de la même manière que pour notre application mobile. J'ai aussi utilisé Figma. Les maquettes sont consultables en annexe page 50.

Conception du back end du site web

Pour la partie back de ce site , le but était donc d'utiliser la même API que pour l'application mobile.

Des routes ont été créées spécialement pour la gestion côté administrateur comme par exemple la route `delete '/chat/:roomId/:id'` qui permet de supprimer une chatroom. Cette route est protégée, il faut avoir le rôle administrateur pour exécuter cette action.

★Axe d'amélioration

- Bien que la méthode agile ait été mise en place, la communication et l'écoute n'a pas toujours été évidente. Il a fallu faire preuve d'une grande souplesse pour s'adapter aux humeurs de chacun.e et d'une grande patience pour gérer les "crises" humaines de notre groupe de travail.

- Nous aurions aimé déployer l'application et la conteneurisé avec docker pour les problèmes de versionning liés aux systèmes d'exploitation et ainsi nous rapprocher au maximum d'un environnement de production.

★ Conclusion

Pour conclure, ce projet m'a permis de découvrir de nouveaux frameworks et aussi de pouvoir monter en compétence sur javascript.

Ce projet qui s'est déroulé sur l'année de formation m'a appris à organiser mon travail en équipe mais aussi seule . Le projet est composé de différents temps entre la conception et le développement. La conception représente 60% du temps total passé sur ce projet tandis que le développement représente lui 40%.

De plus, le projet présenté dans ce dossier m'a permis de voir que je pouvais m'adapter aux changements de situation et rebondir en fonction des imprévus.

★ Annexes

Trello

Tableau de Conception

The screenshot shows a Trello board titled "Chuu / Méthodes - Conception". The board has the following structure:

- CHARTE DE TRAVAIL** column:
 - BRANCH SYSTEM
 - COMMIT SYSTEM
 - TICKETS
 - + Ajouter une carte
- TO DO** column:
 - METTRE LA DB SUR PLESK POUR EVITER LES ERREURS
 - merci de bien mettre dans admin.js les tickets de votre route en commentaire
 - METTRE CLEFS UNIQUES SUR LES CHAMPS COMME LOGIN EMAIL
 - GESTION DES IMAGES ??? avec react Native
 - TO DO: mettre réaction en DB, possibilité de les gérer en socket mais très difficile
 - + Ajouter une carte
- IN PROGRESS** column:
 - UML lien en desc
 - + Ajouter une carte
- FINISH** column:
 - MCD/MLD
 - Créer la database
 - Cahiers des charges lien en desc
 - + Ajouter une carte
- FRONT** column:
 - Wireframe
 - 0/2
 - High fidelity
 - Mood Board
 - Charte graphique
 - + Ajouter une carte

Tableau de l'API

Chuu / API

Visible par l'espace de travail

Tableau

Power-ups

Automatisation

Filtre

Partager

À faire

SPRINT/1

Recette

Terminé

Ajoutez une autre liste

11. (optionnel) API/ Route envoyer un message privé à un user | socket io

TO DO: mettre des routes importantes directement ds l'index, à discuter

TO DO: faire un troisième dossier admin en techno web, car ça sera une app différente de celle en react native,

+ Ajouter une carte

01. API/ Route Inscription ROSE

02. API/ Route connexion JISOO

03. API/ Route GET all users JENNIE

04. API/ Route GET all users from 1 room JENNIE

05. API/ Route POST user to a room LISA

+ Ajouter une carte

29. API/Check expire date token

REFRESH TOKEN vérifier le temps entre les deux tokens pour avoir quelque-chose de "dynamique"

+ Ajouter une carte

Tableau du Front

Chuu / APP ⭐ | **Visible par l'espace de travail** | **Tableau** | **Power-ups** | **Automatisation** | **Filtre** |  **Partager** | **...**

MoodBoard

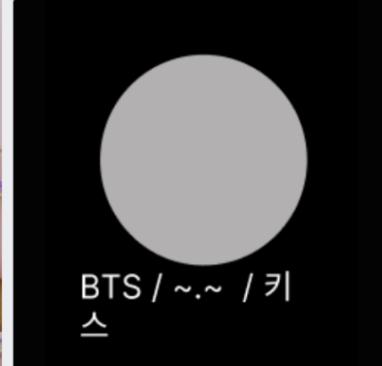
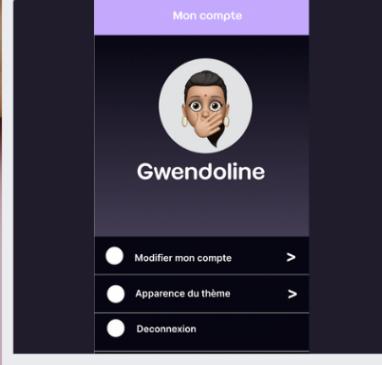
- 🔗 * on Twitter
 - 뉴진스 'OMG' 앨범 도착 🎵
 - pic.twitter.com/gAwII7kwUR— *
- Twitter Affichage plein écran
- 
- Aespa - Next Level - Display Frame 002

Aespa - Next Level - Display Frame 002
designed by Kaixatriplesix. Connect with...
- Dribbble
- 
- + Ajouter une carte

To DO

- Searches for new rooms...
- FRONT/08 searchbar v1
 - ☰ 1
- FRONT/11 array de groupes choisis à adhérer
 - ☰ 1
- FRONT/04 Chat Général | l'user doit être connecté
- FRONT/10 All rooms | optionnel voir toutes les rooms
 - Modifier mon compte >
- FRONT/13 check plus loin
 - ☰ 1
- + Ajouter une carte

En cours

- 
 BTS / ~.~ / 키스
- FRONT/15 block présentation groupe JENNIE
 - ☰ 1
- 
 Mon compte
 Gwendoline
 Modifier mon compte >
 Apparence du thème >
 Deconnexion
- + Ajouter une carte

Recette

- FRONT/10 navbar
 - ☰ 1
- FRONT/02 Connexion page 2 du figma JENNIE
 - ☰
- BTS

azet90 : Tu penses me rejoindre à la... . 4 janv.

 - ☰ 1 1
- FRONT/02 block tous les chats avec dernier message JENNIE
 - ☰ 1
- FRONT/05 Profil | l'user pourra update ses infos page 5 | ROSE
 - ☰ 1
- 
 + Ajouter une carte

Terminé

- LISTE DES VUES avec détails composants sur figma
 - ☒ 0/3
- Maquette High fidelity | <3 ·
- FRONT/00 squelette/archite pour une app react native
- Charte graphique
- Cahier des charges
- Mood Board
 - + Ajouter une carte

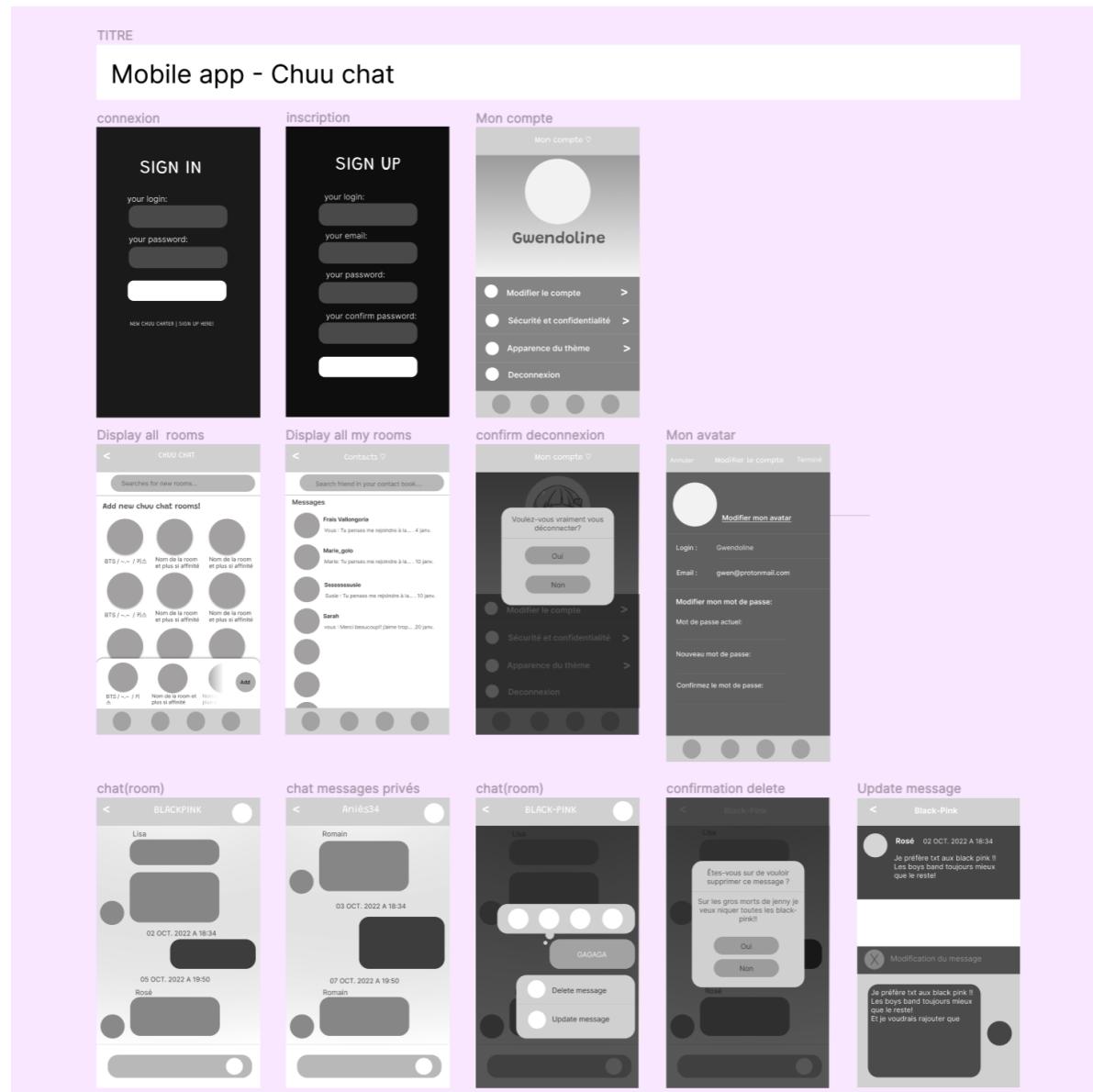
Détails du front

Détails par Screen / FRONT Visible par l'espace de travail Tableau Power-ups Automatisation Filtre Partager ...

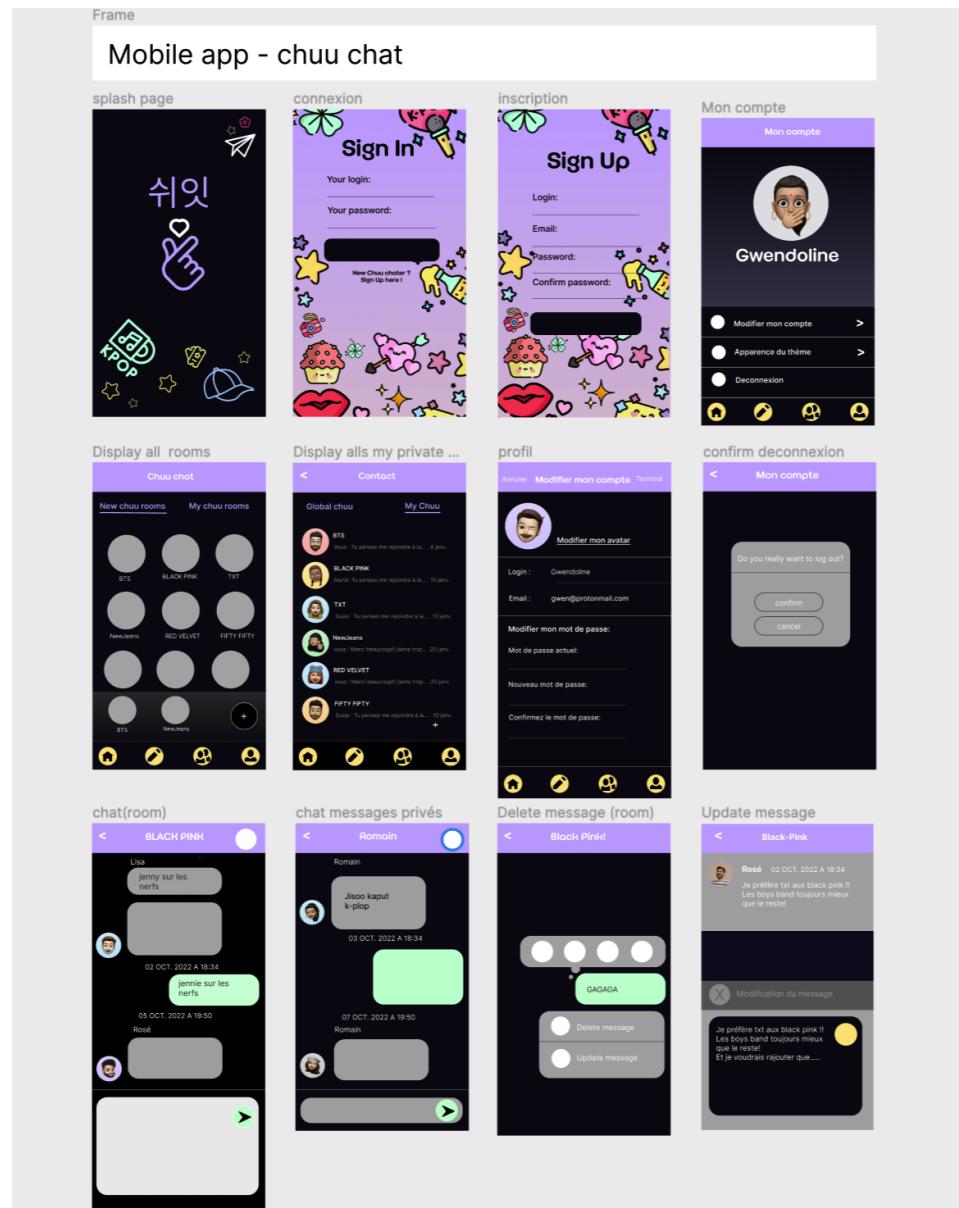
CHATSCREEN	AllRooms	Chuu-Rooms	commentaires	Connexion
trouver un moyen pour que les heures suivent leurs émetteurs dans l'affichage. (gauche ou droite selon qui parle)	Retirer le CHATGLOBAL de my chuu car le filtre Main Chuu sert déjà à trier les discussions.	J'ai remarqué que les filtres qui concernent nos propres discussions ou nos rooms ("my chuu-rooms" ou "My chuu chat") ne se présente pas du même côté sur l'écran. => mettre my chuu rooms à droites et more chuu rooms à gauche.	implémenter les typo. construire le dossier dans assets/Fonts	+ Ajouter une carte
Rajouter aussi le nom du destinataire du message comme sur la maquette	+ Ajouter une carte	Retirer le "titre" de la page chuu-rooms dans la marge en blanc en haut	Mettre en place l'image de chargement de l'appli et créer l'icone qui servira à lancer l'appli depuis le "bureau"	+ Ajouter une carte
adapter la taille des bulles en fonction de la largeur que le messages prends sur l'écran		+ Ajouter une carte		
le fond d'écran fait remonter les messages trop haut. ils sont mangés par la marge blanche du "titre" du screen				
rectifier l'heure d'envoi des messages et de reception des messages à l'heure locale (+1h)				
+ Ajouter une carte				

moyen-pour-que-les-heures-suivent-leurs-émetteurs-dans-l'affichage-gauche-ou-droite-selon-qui-parle

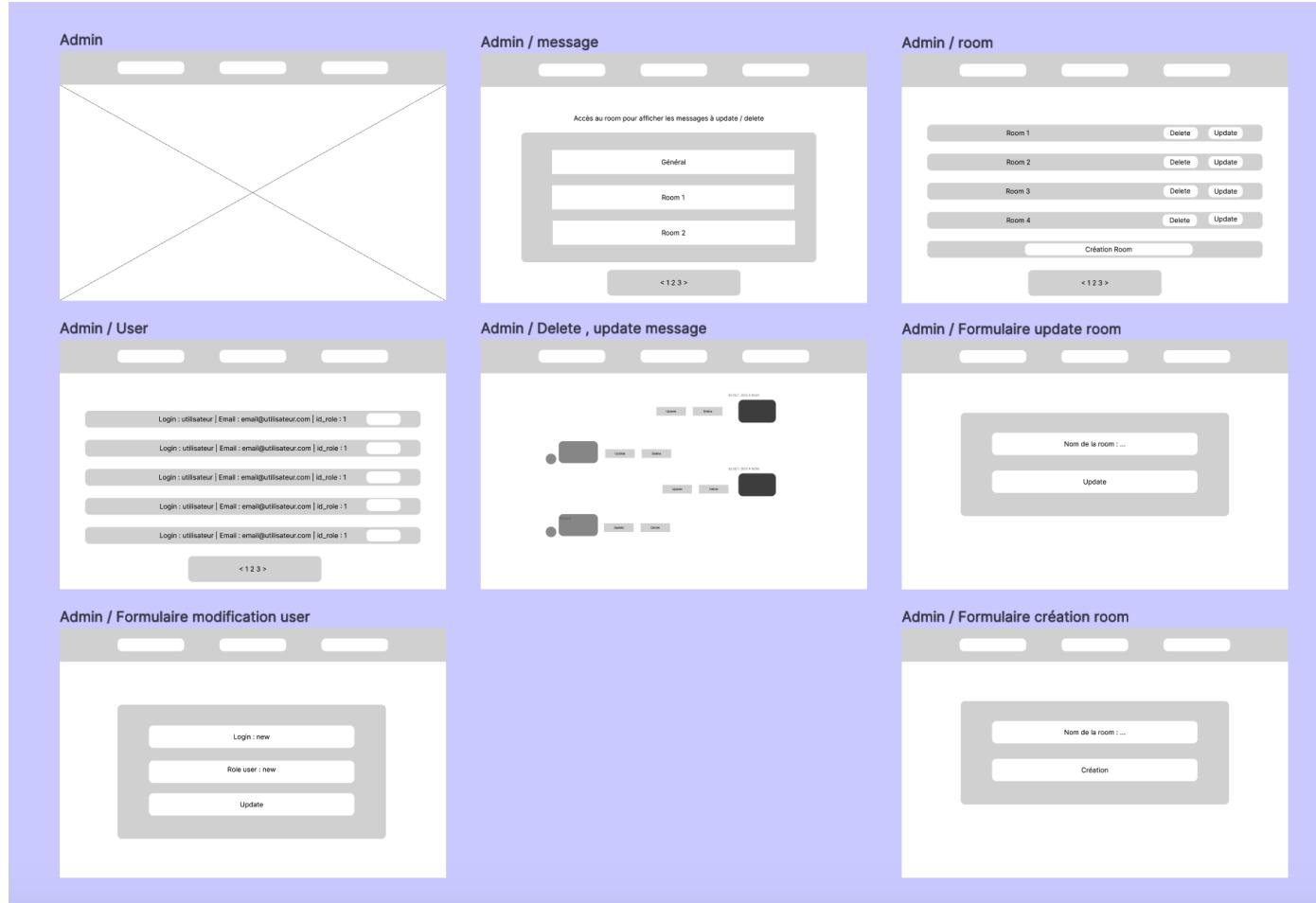
Maquettage Application Mobile Wireframe



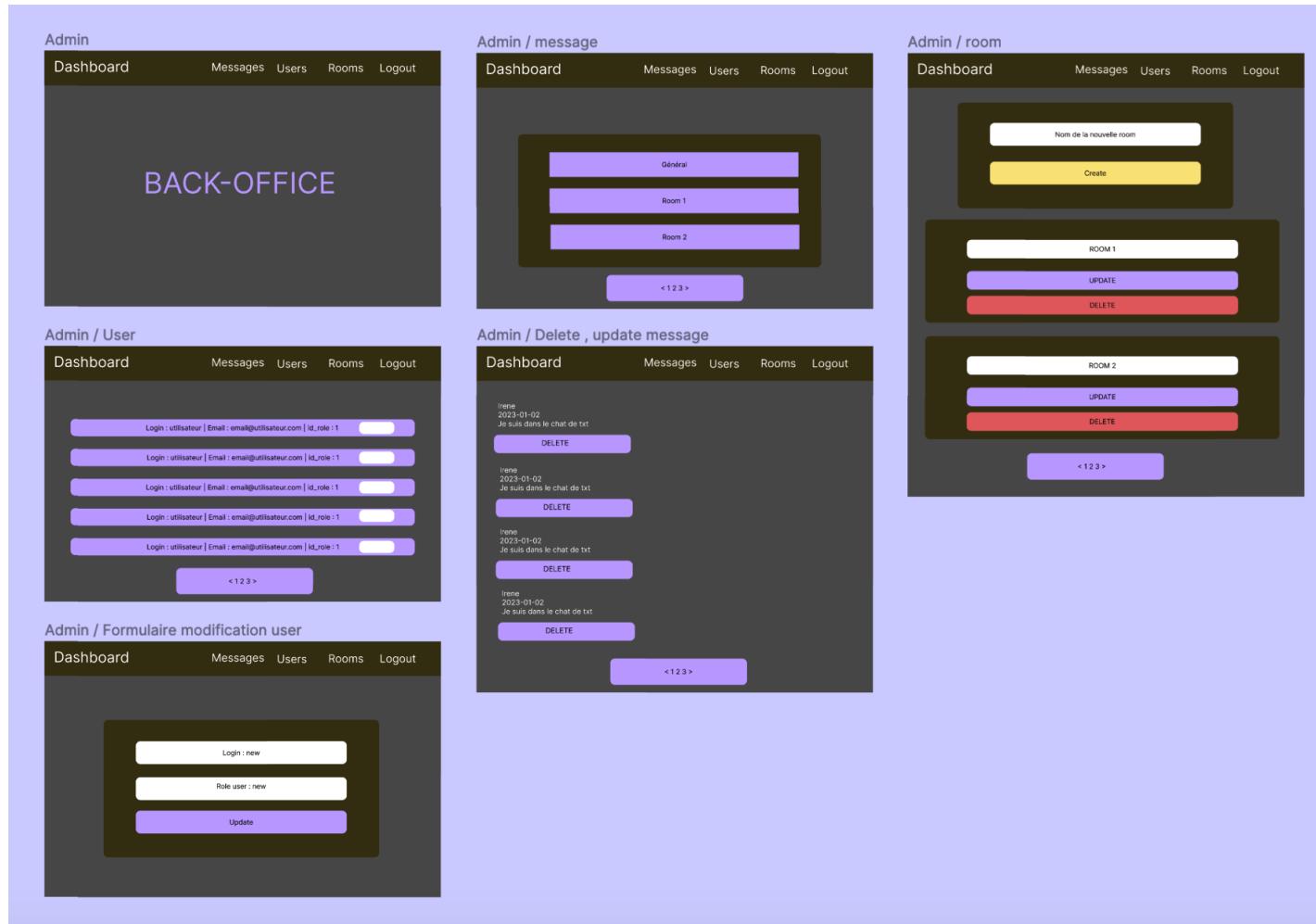
Maquette Haute fidélité



Maquettes Application web Wireframe



Haute fidélité



POSTMAN - Scripts d'enregistrement des tokens dans l'environnement

The screenshot shows the Postman application interface with the following details:

- Header Bar:** Home, Workspaces, API Network, Explore, Search Postman, Invite, Settings, Notifications, Upgrade.
- Sidebar:** Collections, Environments, History.
- Current Request:**
 - Method:** POST
 - URL:** {{baseUrl}}/users/auth
 - Body:** Contains a script to handle response tokens:

```

1 var token = pm.response.json().token;
2 pm.environment.set("authToken", token);
3
4 var rToken = pm.response.json().refresh;
5 pm.environment.set("refreshToken", rToken);

```
- Environment:** chuu localhost

Variable	Initial value	Current value
login	Naomi	Naomix
password	Naomi999	Naomi999
authToken		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e yJsb2dpbi6Ik5hb21peCisIm1hdCI6MTY4NjMxNDQzNCwidHlwZSI6ImF1dGh0b2tlbiIsImVtYWlsIjoiZW1haWxAdGVzdC5jb20iLCJpZCI6IjciLCJpZF9yb2x1I joxLCJpZF9yb29tcyI6WyIwIiwiMiJdLCJleHAIoje20TE00Tg0MzR9.r5Hi0GZhk2Xm7SnBXacmWSuihlV3MKUMIUOKYhCaDS8",
refreshToken		eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e
- Globals:** No global variables.
- Body Response:**
 - Status: 200 OK
 - Time: 138 ms
 - Size: 896 B
 - Save as Example

```

1 {
2   "status": true,
3   "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e  
yJsb2dpbi6Ik5hb21peCisIm1hdCI6MTY4NjMxNDQzNCwidHlwZSI6ImF1dGh0b2tlbiIsImVtYWlsIjoiZW1haWxAdGVzdC5jb20iLCJpZCI6IjciLCJpZF9yb2x1I  
joxLCJpZF9yb29tcyI6WyIwIiwiMiJdLCJleHAIoje20TE00Tg0MzR9.r5Hi0GZhk2Xm7SnBXacmWSuihlV3MKUMIUOKYhCaDS8",
4   "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e  
yJtZXNzYwdlIjoicmVmcmVzaCBUb2tlbiBpbmZvIiwiaWF0IjoxNjg2MzE0NDM0LCJ0eXB1IjoidG9rZW4iLCJlbWFpbCI6ImVtYWlsQHRlc3QuY29tIiwbG9naW4i0  
iJOYW9taXgiLCJpZF9yb29tcyI6WyIwIiwiMiJdLCJpZCI6IjciLCJpZF9yb2x1IjoxLCJleHAIoje20DcxNzg0MzR9.  
cLM1PfxWZ3hB71KFN9bCSBz2gxAD3KiyjEUX4VWds9g"
5

```
- Bottom Navigation:** Online, Find and replace, Console, Import Complete, Runner, Capture requests, Cookies, Trash, Help.

