

Laura Scognamiglio

2 mai 2023

Dossier De Projet Professionnel

Titre: Conception d'application
web et mobile.

Chuu Chat

Titre: Conception d'application web et mobile.	1
I. Contexte	4
1. referentiel de competences :	4
2. Résumé en anglais:	5
3. Cahier des charges:	7
Technos, langages et outils:	9
II. Organisation de travail	10
1. Trello:	10
2. Scrum (méthodologie Agile):	12
3. Méthodologie par ticket:	13
4. Git et GitHub:	14
5. Méthodologie des branches/Gitflow:	15
6. Veille et documentation:	17
III. Conceptions générales	17
1. Multicouches et MVC	17
2. Database relationnelle et requêtes préparées :	19
3. Modèle Merise (MCD, MLD, MPD) :	20
4. Schéma de conception API et définition	22
5. Schéma et conception JWT	24
7. Conception front du projet	30
IV. Fonctionement	35

1. Fonctionnement de l'API	35
2. Routing	36
3. Middleware	38
4. Controller	40
5. Sécurité	41
II. Fonctionnement du front	44
1. Arborescence	44
2. Composants	45
III. Développement	46
1. Extrait de code	46
2. Tests	54
3. Test en alternance	56
V. Conception de la partie administration	59
VI. Problématiques rencontrées	61
VII. Recherches anglophones et documentation	61
Conclusion	62

I. Contexte

Je suis Laura Scognamiglio, élève à La Plateforme. J'ai trente ans et je suis actuellement en alternance chez Enovacom, une entreprise qui crée des logiciels médicaux et interopérables à des fins de recherche ou de gestion pour des hôpitaux. Je suis issue d'un parcours artistique que j'exploite dans mon nouveau quotidien digital. Après une reconversion réussie dans le domaine du web, j'ai obtenu le titre RNCP de développeur d'applications/web et mobile l'année dernière. Pour ce projet scolaire, nous avons été 4 étudiants à travailler à partir du mois de janvier 2023 : Naomi Monderer, Laura Savickaite, Dorian Palace et moi.

Chuu Chat est une application mobile qui utilise une API. Une web app utilise également cette API pour les différents CRUD requis par la partie ADMIN. Plus généralement, Chuu Chat est une application développée en React Native & Node.js, offrant la possibilité de créer un compte, d'utiliser un chat, de choisir et de participer à une room. Autour de l'univers musical de la K-pop, Chuu Chat propose à ses utilisateurs un espace d'échanges autour de leurs groupes préférés. Elle exploite également les technologies web et les websockets pour permettre aux utilisateurs de communiquer en temps réel. L'application repose sur une base de données pour stocker les messages et met en œuvre des mécanismes d'authentification pour garantir sa sécurité.

1. référentiel de compétences :

- Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité

- Maquetter une application
 - Développer une interface utilisateur de type desktop
 - 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 et développer la persistance des données en intégrant les recommandations de sécurité
 - Concevoir une base de données
 - Mettre en place une base de données
 - Développer des composants dans le langage d'une base de données
-
- Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité
 - Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
 - 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

2. Résumé en anglais:

Chuu Chat is a mobile application developed using React Native and Node.js, with the Express.js framework being used for the backend. The application uses an API for communication between the mobile app and the data. Additionally, WebSocket technology is employed to enable real-time communication. The API is also used by a web app, it's the admin part where we can find all the CRUD operations needed.

The core functionality of Chuu Chat revolves around creating user accounts, providing a platform for users to engage in chat conversations, and facilitating the selection and participation in chat rooms. The application's database models are designed following the Merise methodology, ensuring a structured and efficient data management system.

For the security and to protect user interactions, Chuu Chat implements authentication mechanisms using JSON Web Tokens (JWT). This ensures that only authenticated users can access the application and its features, providing a secure environment for communication.

Overall, Chuu Chat is a feature-rich chat application made for K-pop fans, offering them an interactive space to discuss and connect with others who share similar interests. With its utilization of React Native, Node.js, Express.js, WebSocket technology, and JWT-based authentication, Chuu Chat delivers a seamless and secure chatting experience for its users.

3. Cahier des charges:

Portée du projet :

Chuu Chat est un projet d'application mobile visant à développer une plateforme de chat complète pour les passionnés de K-pop. Le projet sera réalisé par une équipe composée de Laura Savickaitė, Naomi Monderer, Dorian Palace et Laura Scognamiglio, au nom du client: "La Plateforme".

Planning :

Le projet devrait être achevé dans un délai d'environ 1 mois et aussi à raison d'une semaine toutes les 2 semaines en entreprise, de février à juillet. L'équipe adoptera une méthodologie Agile et utilisera des outils de gestion de projet tels que Trello pour gérer efficacement les tâches du projet.

Budget :

Le budget estimé pour le projet est compris entre 20 000 et 35 000 euros. Le coût final dépendra de divers facteurs, notamment de la complexité du design, des fonctionnalités souhaitées et du niveau de confidentialité et de sécurité requis pour l'application.

Livrables :

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

Public cible :

Le public cible de Chuu Chat est principalement constitué d'adolescents et de jeunes adultes ayant un vif intérêt pour la K-pop. L'objectif est de fournir un espace joyeux et inclusif où les utilisateurs peuvent partager leur passion pour la K-pop avec d'autres personnes partageant les mêmes centres d'intérêt. Des plateformes existantes telles que Discord, Twitter, Mastodon et WhatsApp serviront de référence pour les attentes des utilisateurs.

Contraintes légales et techniques spécifiques :

Le projet doit respecter les exigences légales telles que la CNIL et le RGPD. Par exemple, bien qu'un administrateur puisse supprimer des messages et des salles, les utilisateurs doivent avoir la possibilité de supprimer leur propre compte. Les réglementations sur la confidentialité et la protection des données seront strictement respectées.

Livrailles attendues :

Le projet vise à fournir une application de chat clé en main avec un panneau administratif, des pages de profil utilisateur, des fonctionnalités d'inscription et de connexion, ainsi que la possibilité de mettre à jour les informations personnelles. L'équipe utilisera des technologies telles que React Native, Node.js, Git/Github, MySQL, la méthodologie Merise, WebSocket et l'authentification JWT. L'intégration avec des API pertinentes sera établie pour améliorer la fonctionnalité de l'application.

Le ton et la personnalité de l'application de chat seront alignés sur l'image de marque, le design visuel et les préférences du client. L'application comprend une salle de chat commune accessible à tous les utilisateurs, si le temps le permet plusieurs salles dédiées aux différents groupes de K-pop.

Le processus de développement suivra la méthodologie Merise, notamment la création du Modèle Conceptuel de Données (MCD) et du Modèle Logique de Données (MLD). Figma sera utilisé pour la conception des maquettes, avec des pages clés comprenant l'inscription, la connexion, l'annuaire des utilisateurs, les détails du profil utilisateur et l'interface de chat. L'application comprend plusieurs routes API, dont des routes d'accès public, d'accès utilisateur authentifié et des routes exclusivement réservées à des fins admin.

Technos, langages et outils:

- **Node.js** : Plateforme JavaScript côté serveur qui permet d'exécuter du code JavaScript en dehors du navigateur. Elle offre une grande évolutivité et facilite le développement d'applications Web rapides et performantes.
- **Express.js** : Framework web minimaliste et flexible pour Node.js. Il facilite la création d'API et de serveurs web en fournissant des fonctionnalités utiles pour la gestion des routes, des requêtes et des réponses.
- **Expo** : Plateforme de développement d'applications mobiles multiplateformes qui simplifie le processus de création et de déploiement d'applications React Native. Elle fournit des outils et des services pour faciliter le développement et les tests.
- **React Native** : Framework JavaScript utilisé pour développer des applications mobiles multiplateformes. Il permet de créer des interfaces utilisateur réactives en utilisant des composants réutilisables. Les applications React Native sont écrites en JavaScript mais sont compilées en code natif pour chaque plateforme cible.
- **WebSocket.io** : Bibliothèque JavaScript qui permet la communication bidirectionnelle en temps réel entre les clients et les serveurs via des connexions persistantes. Elle facilite la mise en place de fonctionnalités de chat et de notifications en temps réel dans les applications Web.
- **JWT (JSON Web Token)** : Standard ouvert permettant de représenter de manière sécurisée des informations sous forme de jetons au format JSON. Ils sont utilisés pour l'authentification et l'autorisation dans les applications Web et mobiles.
- **Merise/MCD/MLD/MPD** : Méthodologie de conception de bases de données relationnelles. Merise est une méthode d'analyse, de conception et de gestion de projets

informatiques. Le MCD (Modèle Conceptuel de Données), le MLD (Modèle Logique de Données) et le MPD (Modèle Physique de Données) sont des étapes de la méthodologie Merise pour décrire la structure et les relations des données entre les entités.

- **Trello** : Outil de gestion de projet en ligne basé sur la méthode Kanban. Il permet de créer des tableaux pour organiser les tâches, les suivre et collaborer efficacement au sein d'une équipe.

- **Figma** : Outil de conception d'interface utilisateur (UI) et d'expérience utilisateur (UX). Il permet de créer des maquettes interactives, de collaborer avec d'autres designers et de partager facilement les designs.

- **Postman** : Outil de développement d'API qui permet de tester, de déboguer et de documenter les API. Il offre une interface conviviale pour envoyer des requêtes HTTP, visualiser les réponses et gérer les collections d'API.

- **npm (Node Package Manager)** : Gestionnaire de paquets pour Node.js qui permet d'installer, de mettre à jour et de gérer les dépendances des projets. Il offre un vaste écosystème de packages et simplifie la gestion des bibliothèques utilisées dans les applications Node.js.

- **GitHub/Git** : Plateforme de développement collaboratif et de gestion de versions de code. GitHub permet aux développeurs de partager, de collaborer et de contrôler les versions de leur code source grâce à Git, un système de contrôle de versions distribué.

II. Organisation de travail

1. Trello:

Trello est un outil de gestion de projet en ligne qui permet de suivre et d'organiser les différentes tâches et activités d'un projet. Nous avons fait le choix d'avoir plusieurs trello par section:

- **API** tout ce qui traite de celle-ci
- **APP** tout ce qui va concerner features front
- **Méthodes/Conception** là où on retrouve nos instructions pour nommage de branches etc,
- **Screen/Front**, il est dédié à la finalisation maximale de Chuu Chat

Tableau Api: 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 App: 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 Screen, 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.



légende: Trello de la partie API

2. Scrum (méthodologie Agile):

Scrum est une méthodologie de gestion de projet Agile pour le développement logiciel en groupe. Elle se base sur des cycles de développement appelés « sprints », dans mon entreprise ils durent 3 semaines. Pendant chaque sprint, l'équipe se concentre sur la réalisation de tickets, une sprint review a lieu en fin de cycle pour présenter les avancées. Le Scrum Master donne ensuite une retro ou l'on parle du cycle en encourageant la collaboration, la transparence et l'esprit d'équipe. Les rôles clés dans Scrum sont le Scrum Master, qui facilite le processus, le Product Owner, qui définit les besoins du client, et l'équipe de développement, qui réalise les fonctionnalités. Nous nous sommes inspirés de nos alternances et nous avons donc mis en place des daily réguliers et des réunions pendant les sprints.

Dans notre projet nous avons puisé dans cette méthodologie pour créer notre système. Ainsi nous avons fait des daily meetings où chacun parlait de ce qu'il avait réussi ou non à

faire la veille et ce qu'il comptait faire aujourd'hui. Nous n'avons pas appliqué cette méthode jusqu'au totallement, mais elle a été un véritable succès pour l'encadrement et la réalisation de l'API et du BACK de l'application.

3. Méthodologie par ticket:

En n'ayant pas de Jira mais des Trellos nous avons essayé d'adopter la méthodologie du ticket sans le système de points pour gérer les problèmes et les tâches d'un projet à l'aide de tickets. Chaque ticket représente une unité de travail, qu'il s'agisse d'une fonctionnalité à développer, d'un bug à corriger ou d'une demande de modification. Les tickets contiennent des informations telles que la description de la tache à faire. Ils sont choisis par les membres de l'équipe qui sont responsables de leur résolution, et leur statut est mis à jour au fur et à mesure de l'avancement dans les colonnes associés. Là aussi une forte inspiration de nos alternance est visible.

Dans le trello regroupent les informations on retrouves les informations résumé par le lead dev comme suit:

TICKETS

Dans la liste CHARTE DE TRAVAIL

Notifications

⌚ Suivre

Description Modifier

- chacun est libre de créer un ticket
--> structure (As a prospect / Description / Sécurité)
- un ticket peut-être lié au côté front/back, ne pas oublier de le lier dans les commentaires !
- lorsqu'on s'assigne un ticket -> marquer son pseudo dans le titre (ex : 14. API/Route xxxxxxxxxxxx ROSÉ)

FLUX DE TRAVAIL :

"à faire" le ticket est prêt à être développé.
"en cours" le ticket est en cours de développement.
"recette" le ticket est mergé dans sprint, tout le monde peut le tester / je poste en commentaire du ticket que c'est mergé en sprint et je rajoute le nom de la branche.
"terminé" le ticket est passé en dev/main

Activité

Masquer les détails

Automatisation ⓘ

+ Ajouter un bouton

Actions

→ Déplacer

Copier

Écrivez un commentaire...

laura savickaite a ajouté cette carte à CHARTE DE TRAVAIL
3 janv. à 12:18

4. Git et GitHub:

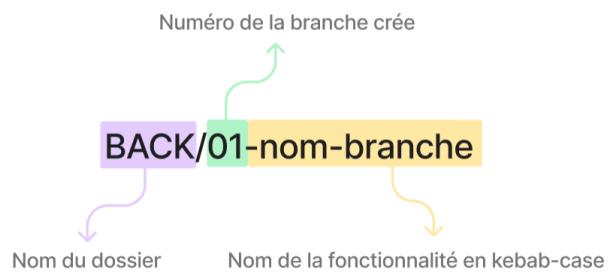
- Git est un système de contrôle de version / versioning, utilisé dans le développement de logiciels. Il permet de suivre et de gérer les modifications apportées au code source d'un projet. Git enregistre l'historique complet des modifications, ce qui facilite la collaboration et la gestion des différentes versions du code. Il permet aux développeurs de travailler sur des branches distinctes, d'effectuer des commits pour enregistrer les modifications et de fusionner les branches une fois les fonctionnalités

développées et testées. Git facilite également la résolution des conflits lors de la fusion des branches.

- GitHub, quant à lui, est une plateforme en ligne qui utilise Git comme système de contrôle de version. Elle offre des fonctionnalités supplémentaires telles que l'hébergement de dépôts Git, la collaboration avec d'autres développeurs, le suivi des problèmes et des demandes de fusion, ainsi que des outils pour faciliter le processus de développement logiciel en équipe.

5. Méthodologie des branches/Gitflow:

La méthodologie consiste à travailler sur des branches distinctes. Chaque branche représente une fonctionnalité. Nous avons créé une branche à partir de la branche principale ("master"), développé la fonctionnalité sur cette branche nouvelle, nous l'appelons puis la fusionnent /mergé avec la branche principale une fois terminée. Cette approche permet de travailler de manière isolée sur des fonctionnalités indépendantes, de faciliter la collaboration et de réduire les risques de conflits lors de la fusion. Pour ce faire nous avons aussi essayé d'appliquer des règles de noms pour les branches, des commits détaillés et réguliers.



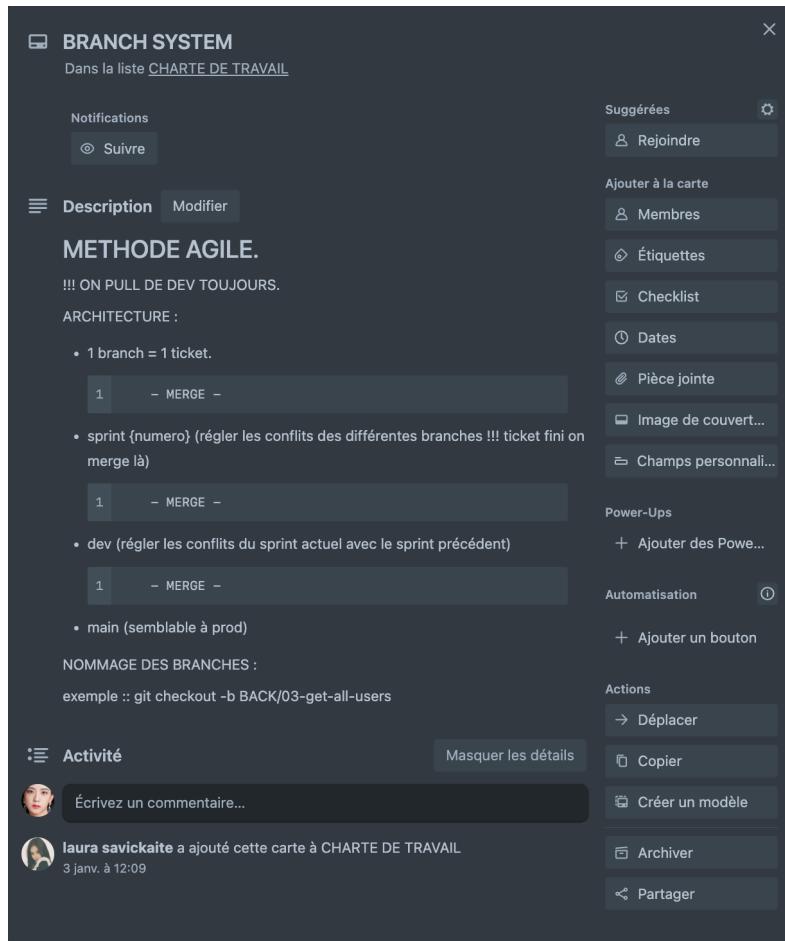
légende: schema structure de branche back chuu chat



Suppression d'un message qui appartient à un chat spécifique	9 Jan 2023 16:29	dorian-palace	28f7df04
ajout du détail des ticket et des routes dans tous les routers	9 Jan 2023 16:16	naomie-mon...	f90a23a6
[BACK/26] correction to the ticket global chat	9 Jan 2023 16:14	Laura Savick...	f3737787
[BACK/28] create a room for admin done	9 Jan 2023 15:56	Laura Savick...	7155b224
resolve merge conflict BACK/05-post-user-to-a-room	9 Jan 2023 15:26	naomie-mon...	2163edc9
BACK/05-post-user-to-a-room verif sur insertToRoom ok	9 Jan 2023 15:18	naomie-mon...	0570bce2
Merge BACK/27-get-message-chat-room	9 Jan 2023 15:16	dorian-palace	1b568983
[BACK/27] Route get message d'un chat ou l'utilisateur est inscrit à la room	9 Jan 2023 15:10	dorian-palace	fe2c3a1e
sprint/1 merged BACK/17	9 Jan 2023 14:41	Laura Savick...	0c73923a
[BACK/17] update message by user done	9 Jan 2023 14:38	Laura Savick...	71d4e5f8
Merge branch 'BACK/26-route-admin-get-all-messages' into sprint/1	9 Jan 2023 12:06	[laura-scogn...	4b625dbc
BACK/26-route-admin-get-all-messages fix auth problems, get all messages now fu...	9 Jan 2023 12:04	[laura-scogn...	13b973b4
fix auth problems, get all messages now fonctionnal	9 Jan 2023 11:55	[laura-scogn...	2b362ccd
sprint/1 getUsersDetails == added details in route	9 Jan 2023 11:55	Laura Savick...	6d596b87
check conflits non montrés BACK/18	9 Jan 2023 11:50	Laura Savick...	bce2b370
sprint/1 merged BACK/18	9 Jan 2023 11:47	Laura Savick...	0360663e
[BACK/18] delete message done	9 Jan 2023 11:44	Laura Savick...	fa4f5158
sprint/1 clean	9 Jan 2023 11:21	Laura Savick...	5048396d
merge sprint/1 sur BACK/05	8 Jan 2023 22:18	naomie-mon...	cd8e5f02
fix accolade problem in UserController	8 Jan 2023 16:46	[laura-scogn...	b5ce1254
[BACK/08]update-user	6 Jan 2023 18:06	dorian-palace	723077c7

légende: extrait du gitGraph - plugin VsCode - du projet

Nous pouvons voir que nous avons appliqué le processus suivant pour nommer et gérer nos branches en rapport aux tickets choisis comme le démontre ce screen de notre trello.



BRANCH SYSTEM
Dans la liste CHARTE DE TRAVAIL

Notifications
Suivre

Description Modifier

METHODE AGILE.

!!! ON PULL DE DEV TOUJOURS.

ARCHITECTURE :

- 1 branch = 1 ticket.
- sprint {numero} (réglé les conflits des différentes branches !!! ticket fini on merge là)
- dev (réglé les conflits du sprint actuel avec le sprint précédent)
- main (semblable à prod)

NOMMAGE DES BRANCHES :

exemple :: git checkout -b BACK/03-get-all-users

Activité Masquer les détails

Écrivez un commentaire...

Actions

Déplacer Copier Créer un modèle Archiver Partager

Power-Ups

Suggérées Rejoindre Ajouter à la carte Membres Étiquettes Checklist Dates Pièce jointe Image de couvert... Champs personnali...

Automatisation

Ajouter des Pow... Ajouter un bouton

laura savickaite a ajouté cette carte à CHARTE DE TRAVAIL
3 janv. à 12:09

6. Veille et documentation:

La veille et la documentation sont des pratiques essentielles dans le développement de logiciels. La veille consiste à se tenir à jour des dernières avancées technologiques, des nouvelles bibliothèques et des meilleures pratiques du secteur. Cela permet de rester informé des tendances et des évolutions de l'industrie et d'adapter les choix techniques en conséquence. La documentation, quant à elle, consiste à créer et à maintenir des documents décrivant le fonctionnement du code, les procédures d'installation, d'utilisation et de maintenance, ainsi que les décisions architecturales prises. La documentation facilite la compréhension et la collaboration au sein de l'équipe, et constitue une référence précieuse pour les jeunes développeurs.

III. Conceptions générales

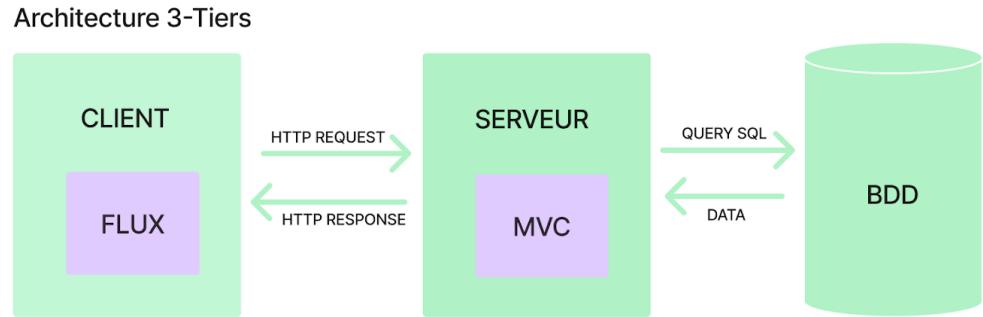
1. Multicouches et MVC

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.

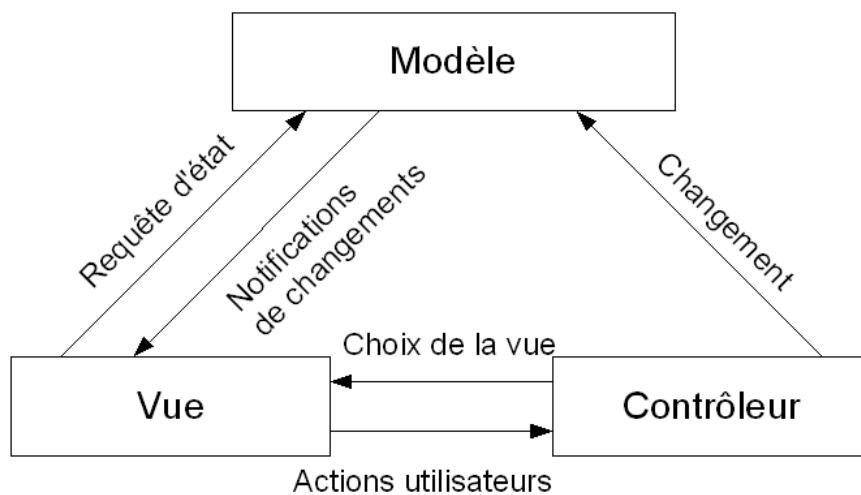
Cette architecture 3-tiers est également une architecture “**client-serveur**”. Plus précisément, il s'agit de l'environnement dans lequel nos applications de machines clientes

(les applications web et mobile) communiquent avec notre application de machines de type serveurs (l'api).



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 view. 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.



Dans notre cas, nous n'avons pas suivi ce design à la lettre. En effet, alors que nous avons bien un controller, nous avons trouvé cela plus facile de ne pas séparer le model du controller et donc avoir un seul dossier. La view est, quant à elle, gérée par le front.

Avoir un modele et un controller, aurait signifié découper cette petite fonction et en extraire la requête sql pour la placer dans le modele. Cela nous semblait un peu moins pratique pour une API de notre gabarit - nous avons donc préféré tout garder dans un même fichier.

2. Database relationnelle et requêtes préparées :

Dans notre projet, nous utilisons une base de données relationnelle (MySQL) car nous sommes aussi à l'aise avec le procédé qui nous permet de structurer notre projet et ainsi le développer dans de bonnes conditions. dossier et nous utilisons des requêtes préparées pour interagir avec celle-ci. Les requêtes préparées sont des requêtes SQL qui permettent d'améliorer les performances et la sécurité des opérations de base de données. En effet, elles offrent une protection contre les attaques par injection SQL en s'assurant que les données utilisateur sont échappées.

Nous avons choisi d'utiliser MySQL comme système de gestion de base de données (SGBD) et le langage SQL pour se connecter à la base de données. Cette combinaison est couramment utilisée dans le développement web avec Node.js en raison de ses avantages et de sa compatibilité avec les technologies associées.

Avantages de l'utilisation d'une base de données relationnelle avec Node.js :

- Structure bien définie : Les bases de données relationnelles, telles que MySQL, offrent une structure claire et organisée pour stocker les données. Cela facilite la création de tables et de relations entre les entités, ce qui est essentiel pour la modélisation de données complexe.

- Langage SQL : Le langage SQL est facile à apprendre et à utiliser. Il permet d'effectuer des requêtes complexes pour récupérer, modifier, insérer ou supprimer des données dans la base de données. Donc parfait pour les différents CRUD

- Intégrité des données : Les bases de données relationnelles permettent de définir des contraintes telles que les clés étrangères, etc.

Inconvénients potentiels de l'utilisation d'une base de données relationnelle avec Node.js

:

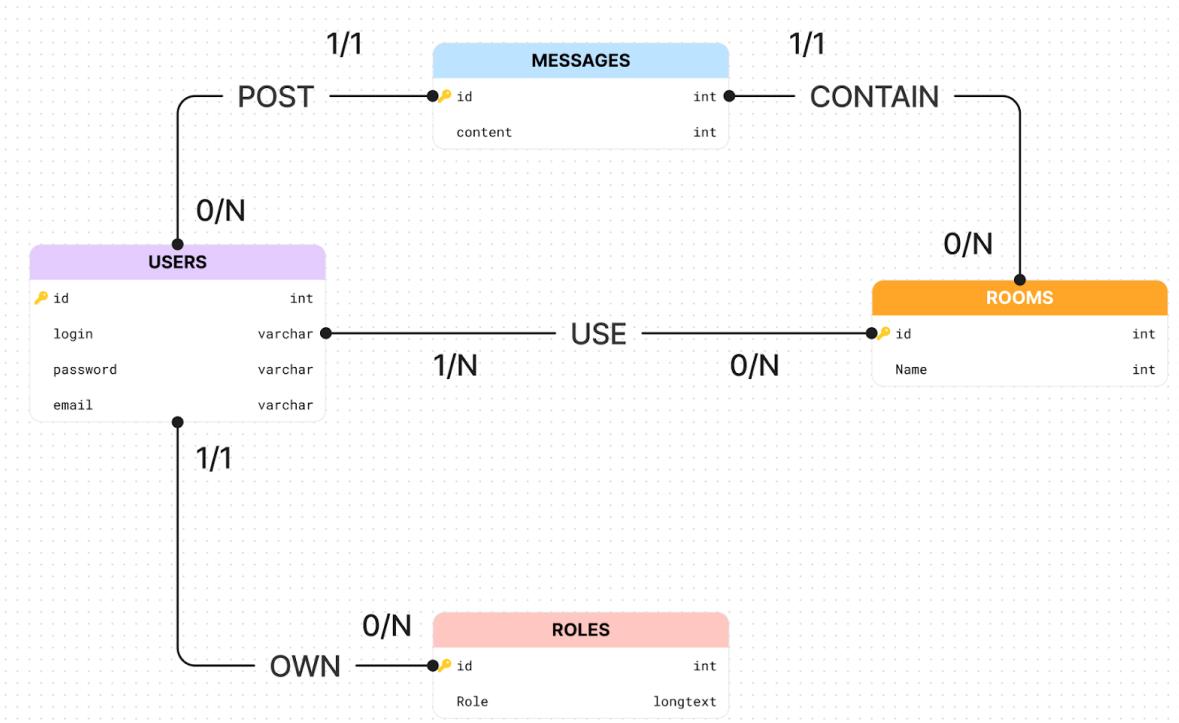
- Scalabilité : Les bases de données relationnelles peuvent présenter des limitations en termes de scalabilité lorsqu'il s'agit de gérer de grands volumes de données ou de faire face à une charge importante. Les jointures peuvent coûter cher en performances.
- Temps de conception : La conception et le développement d'une base de données relationnelle peuvent prendre plus de temps par rapport à d'autres types de bases de données, en raison de la nécessité de définir des schémas, des relations et des contraintes.

3. Modèle Merise (MCD, MLD, MPD) :

Nous avons adopté la méthodologie Merise pour la conception de la base de données de notre application Chuu Chat. Merise est une approche de modélisation des données qui permet de représenter les entités, les relations et les contraintes d'une manière claire et structurée.

- Modèle Conceptuel de Données (MCD) : Le MCD est utilisé pour décrire les entités principales de notre système, ainsi que les relations entre elles. Dans notre cas, nous avons identifié les entités suivantes : users, participants, rooms et messages. Le MCD définit les liens entre les entités avec:

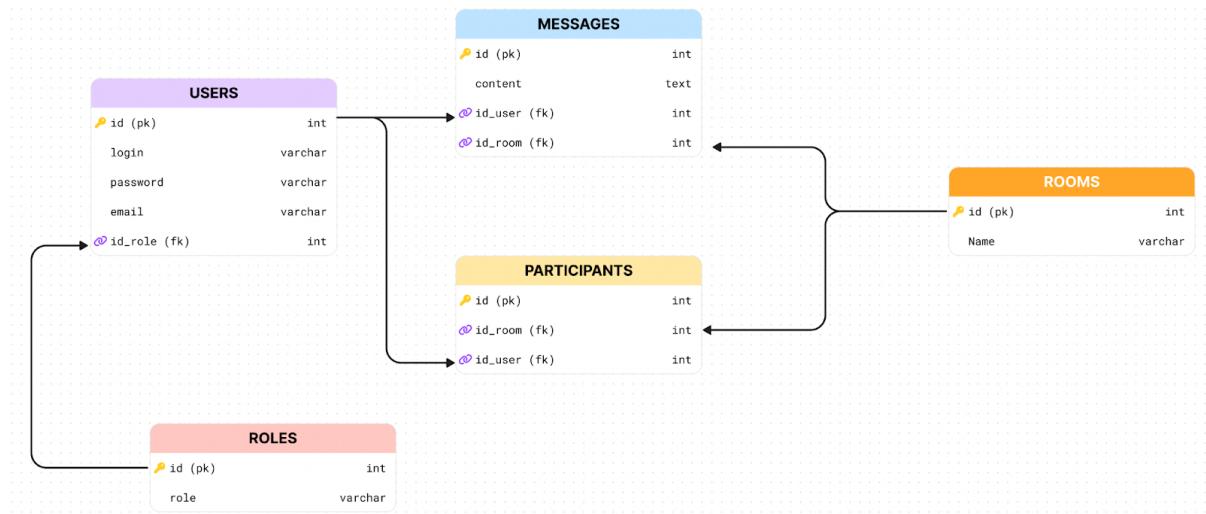
- des verbes d'action à l'infinitif,
- des cardinalités
- **PAS** de clés étrangères
- **PAS** de table de liaisons



légende: Premier **MCD** de chuu chaat

- Modèle Logique de Données (MLD) : Le MLD est une étape intermédiaire entre le MCD et le Modèle Physique de Données (MPD). Il permet de convertir les entités et les relations du MCD en tables et en schémas relationnels dans la base de données. Le MLD spécifie:

- typages des clés primaires / secondaires
- les clés primaires et secondaires,
- **PAS** de verbe mais des flèches
- tables de liaisons



légende: Premier **MLD** de chuu chaat

Nous avons utilisé MySQL en tant que SGBD relationnel avec Node.js pour notre projet Chuu Chat. Nous avons conçu la base de données en suivant la méthodologie Merise, en utilisant le MCD pour représenter les entités et les relations, et le MLD pour convertir le modèle conceptuel en tables relationnelles. Etant donné que l'application mobile n'est pas terminée elle ne possède pas encore de MPD, modèle physique de données qui exprime le stade final de l'application avec:

- les cardinalités sous forme de pattes d'oies
- flèches
- plus de clés étrangères

Pas de MPD car le projet n'est pas terminé.

4. Schéma de conception API et définition

Dans notre architecture logicielle, nous avons prévu différents niveaux d'accès pour nos utilisateurs. Les utilisateurs standards pourront utiliser l'application mobile, tandis que les administrateurs auront accès à la fois à l'application web et mobile.

Pour assurer une bonne synchronisation des données, nous utilisons une API commune pour l'application mobile et le site web. Cette API sert d'intermédiaire entre nos interfaces utilisateur et notre base de données MySQL.

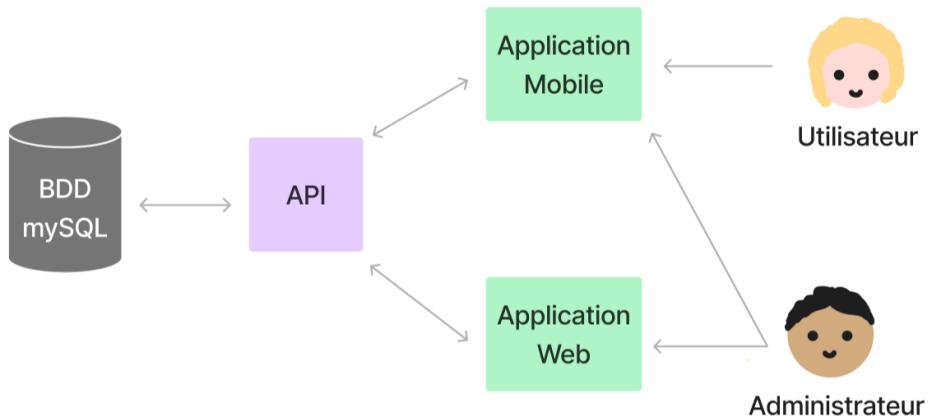
API REST

Basé sur le principe du protocole HTTP (hypertext transfer protocol), qui est un protocole de communication client-serveur basé sur une architecture de type requête (client)/réponse (serveur), une API REST (Representational State Transfer) permet à différents systèmes de communiquer entre eux en utilisant des méthodes HTTP standardisées comme GET (récupérer une donnée), POST (enregistrer une donnée), PUT (mettre à jour l'intégralité des informations d'une donnée), PATCH (mettre à jour partiellement une donnée) et DELETE (supprimer une donnée). Les données échangées entre les systèmes sont souvent formatées en JSON ou en XML. Les API REST sont généralement considérées comme plus légères et plus flexibles que les API SOAP.

Une API SOAP (Simple Object Access Protocol) est une API basée sur un protocole de transport plus lourd que HTTP, souvent basé sur XML et nécessitant des en-têtes spécifiques. Elles sont souvent utilisées dans des environnements d'entreprise où la sécurité et la fiabilité sont primordiales.

Ainsi, lorsque nous, en tant qu'utilisateurs, interagissons avec l'application mobile ou le site web, nos demandes sont envoyées à l'API. L'API se charge alors de communiquer avec notre base de données MySQL pour récupérer ou enregistrer les informations nécessaires.

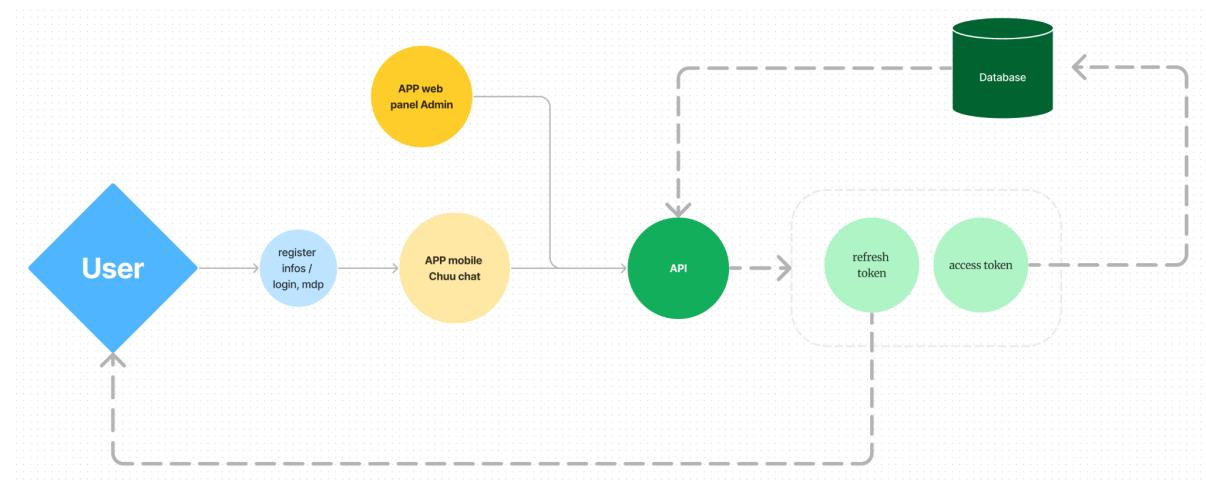
Ce choix d'architecture nous permet de gérer les données de manière centralisée et d'offrir une expérience utilisateur cohérente entre l'application mobile et le site web. De plus, cela facilite la maintenance et les mises à jour, car les modifications apportées à l'API sont automatiquement prises en compte dans les deux interfaces utilisateur.



5. Schéma et conception JWT

Au sein du projet, lors de sa conception initiale, nous n'avions pas envisagé l'application d'un système d'authentification basé sur le JWT (JSON Web Token) de manière approfondie. Cependant, nous avons décidé de re-imaginer notre gestion du JWT plus généralement une conception orientée token grâce à la documentation suivante: <https://auth0.com/docs/secure/tokens/token-best-practices>.

voici l' des premières versions de schématisation du token, nous verrons par la suite un meilleur schéma:



Pour commencer quelques définitions:

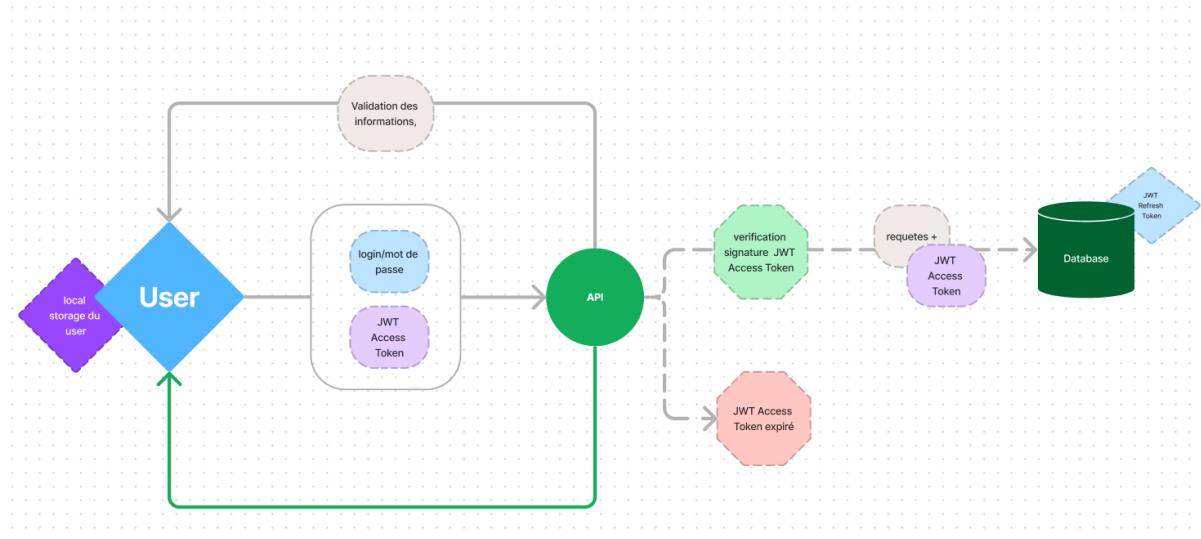
Le JWT est un format de token sécurisé utilisé pour l'authentification et l'autorisation dans les applications web et mobiles. Il permet de représenter des informations structurées sous forme de JSON, signées numériquement et chiffrées si nécessaire. Le JWT se compose de trois parties : l'en-tête (header), des données (payload) et la signature (signature). L'en-tête contient des informations sur le type de token et l'algorithme de signature utilisé. Le payload : les données spécifiques à l'application, telles que l'identifiant de l'utilisateur, son rôle et ses autorisations. La signature garantit l'intégrité des données et permet de vérifier l'authenticité du token.

Dans la vision d'appliquer la méthode token-oriented sur Chuu Chat, l'authentification et l'autorisation reposent principalement sur l'utilisation de ces JWT. Lorsqu'un utilisateur se connecte à l'application, il reçoit un JWT access token qui est ensuite utilisé pour accéder aux ressources et effectuer des opérations. Ce token est envoyé avec chaque requête vers l'API de l'application pour prouver l'authentification de l'utilisateur et autoriser ses actions. Si le JWT access token expire, un JWT refresh token peut être utilisé pour obtenir un nouveau JWT access token sans avoir besoin de demander à l'utilisateur de se reconnecter.

En mettant en place un système d'authentification basé sur le JWT, nous avons pu améliorer la sécurité de notre application en garantissant l'authenticité des utilisateurs et en contrôlant leur accès aux différentes fonctionnalités. De plus, cette méthode nous a permis de gérer efficacement les sessions utilisateur sans avoir recours à des mécanismes traditionnels tels que les cookies ou les sessions serveur, ce qui offre une plus grande flexibilité et une expérience utilisateur améliorée.

Le JWT est un format de token sécurisé utilisé pour l'authentification et l'autorisation dans les applications web et mobiles. Une application conceptualisée selon la méthode token-oriented se base principalement sur l'utilisation de JWT pour gérer l'authentification et l'autorisation des utilisateurs. En implémentant ce système d'authentification au sein de notre projet, nous avons renforcé la sécurité et la gestion des sessions utilisateur de manière efficace et flexible.

Voici un schéma de conceptualisation token oriented mais qui n'a pas pu être appliqué.
Il représente ce que nous aimerais actuellement refaire dans notre projet.



description du schéma par étape:

En tant qu'utilisateur créant un compte pour la première fois, voici les étapes du processus d'inscription sur Chuu Chat :

1. Je lance l'application Chuu Chat ou accède au panel admin web.
2. Je sélectionne l'option "Créer un compte" ou une fonction similaire.
3. Je fournis les informations requises telles que mon nom, mon adresse e-mail et mon mot de passe.
4. Les informations sont envoyées à l'API pour créer un nouveau compte utilisateur.
5. L'API vérifie la validité des informations fournies, notamment la disponibilité de l'adresse e-mail et la conformité du mot de passe.

6. Si mes informations sont valides, l'API crée un nouvel utilisateur dans la base de données en enregistrant les détails de mon compte.

7. Une fois mon compte créé avec succès, l'API génère un JWT access token contenant des informations d'identification et d'autres données pertinentes spécifiques à mon compte, comme mon rôle, mes autorisations et la durée de validité du token.

8. L'API renvoie le JWT access token à Chuu Chat.

9. Chuu Chat ou le panel admin stocke le JWT access token localement, par exemple dans le stockage sécurisé de mon appareil ou dans un cookie.(Dans notre cas ce sont les secure store qui ont été choisis)

10. Je reçois une notification ou un message indiquant que mon compte a été créé avec succès et je suis redirigée vers l'écran de connexion.

11. Je peux ensuite me connecter en utilisant les identifiants que j'ai créés (nom d'utilisateur et mot de passe).

12. Lorsque j'envoie une requête à l'API depuis Chuu Chat. pour accéder à des données ou effectuer des opérations, le JWT access token est inclus dans l'en-tête de la requête.

13. L'API vérifie l'intégrité et la validité du JWT access token en utilisant la clé secrète avec laquelle il a été signé lors de sa génération. Cela garantit que le token n'a pas été modifié et qu'il est authentique.

14. L'API extrait les informations du JWT access token pour m'identifier en tant qu'utilisateur et vérifier mes autorisations.

15. Si le JWT access token est valide et que j'ai les autorisations nécessaires, l'API traite ma requête et renvoie les données demandées à Chuu Chat.

16. Si le JWT access token est invalide (par exemple, s'il a été modifié ou a expiré), l'API renvoie une erreur d'authentification et Chuu Chat ou l'app web des admins doit me demander une nouvelle authentification en utilisant le JWT refresh token.

17. Lorsque j'ai besoin d'une nouvelle authentification, l'application mobile ou le site web envoie une requête à l'API avec le JWT refresh token pour obtenir un nouveau JWT access token.

18. Si le JWT refresh token est valide, l'API génère un nouveau JWT access token et le renvoie à l'application mobile ou au site web.

19. Ce processus d'authentification et d'utilisation du JWT access token se répète à chaque future requête, assurant ainsi une authentification sécurisée entre et l'API !!!

Mais dans ce schéma on peut voir que certains cas ne sont pas pris en compte et dans notre projet. Pour l'instant nous allons voir ensemble les cas que nous avons rencontrés et leurs solutions théoriques:

- Le compte de l'utilisateur est supprimé/bloqué/suspendu.
- Le mot de passe de l'utilisateur est modifié.
- Les rôles ou les permissions de l'utilisateur sont modifiés.
- L'utilisateur est déconnecté par l'administrateur.

voici les sources qui nous permis d'envisager des pistes de réponses:

<https://auth0.com/docs/secure/tokens/json-web-tokens>

<https://stackoverflow.com/questions/21978658/invalidating-json-web-tokens>

<https://developer.okta.com/blog/2017/08/17/why-jwts-suck-as-session-tokens>

donc à ces contraintes nous avons trouvé des solutions théoriques qui invalident les JWT côté serveur :

Si un utilisateur effectuait une action qui nécessitait une mise à jour de l'état du compte, comme la suppression, le blocage ou la suspension du compte, nous pourrions invalider le JWT associé à cet utilisateur en maintenant une liste blanche ou une liste noire des JWT valides/invalides. Ainsi, à chaque requête, nous vérifierons si le JWT fourni est toujours valide.

Nous pourrions envisager d'ajouter un "claim" supplémentaire au JWT pour spécifier des informations telles que l'état du compte de l'utilisateur ou la date de la dernière modification du mot de passe. De cette manière, lorsqu'une requête serait reçue, nous pourrions vérifier ce "claim" et prendre les mesures appropriées en conséquence. Exemples de "claims" courants: "exp" (expiration) qui indique la date et l'heure à laquelle le JWT expire, "iss" (issuer) qui indique l'émetteur du JWT, "aud", "sub", etc.

Pour limiter la fenêtre d'exposition en cas de changement d'état du compte ou de modification des autorisations, nous pourrions définir des durées de validité plus courtes pour les JWT. Cela nécessiterait une régénération plus fréquente des JWT, mais offrirait une meilleure réactivité en cas de changements importants.

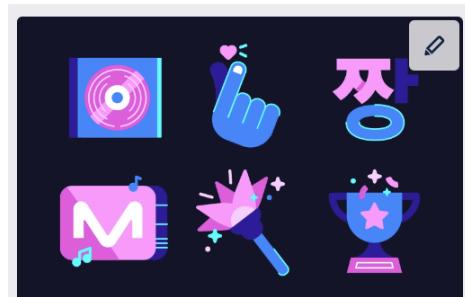
Nous pourrions envisager de mettre en place une stratégie de révocation des JWT qui permettrait d'annuler ou de révoquer un JWT spécifique avant son expiration normale. Cela pourrait être réalisé en maintenant une liste noire des JWT révoqués et en vérifiant si le JWT fourni est présent dans cette liste lors de chaque requête.

Nous pourrions également envisager d'utiliser des cookies sécurisés pour stocker les JWT, plutôt que de les stocker localement dans le stockage sécurisé du dispositif ou dans un cookie standard. Les cookies sécurisés sont moins vulnérables aux attaques de vol de JWT et offrent une meilleure gestion et révocation côté serveur. Mais ils doivent être sacrément plus

complexes à mettre en place étant donné que nous avons jamais eu recours au cookies jusqu'à lors.

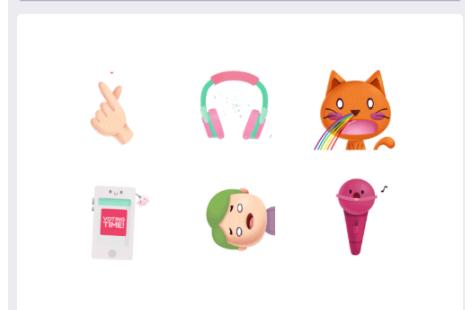
7. Conception front du projet

De par sa vocation, on voulait pour Chuu chat une présence visuelle assez dynamique et reflétant la jeunesse. L'inspiration première était les moyens de communications des générations plus jeunes : les emojis. En effet, ces derniers sont un bon moyen de refléter une certaine émotion et même la convier à l'interlocuteur - en créant/personnalisant nos propres emojis, on voulait inspirer une certaine familiarité : un endroit “fun”, amical et où il fait bon d'être. Il y a également des inspirations du pixel art et des esthétismes des années 90/début 2000 qui semblent refaire surface depuis quelques années.



● KPop Icons
KPop Icons designed by Jim Idan.
Connect with them on Dribbble; the...

● Dribbble Affichage plein écran

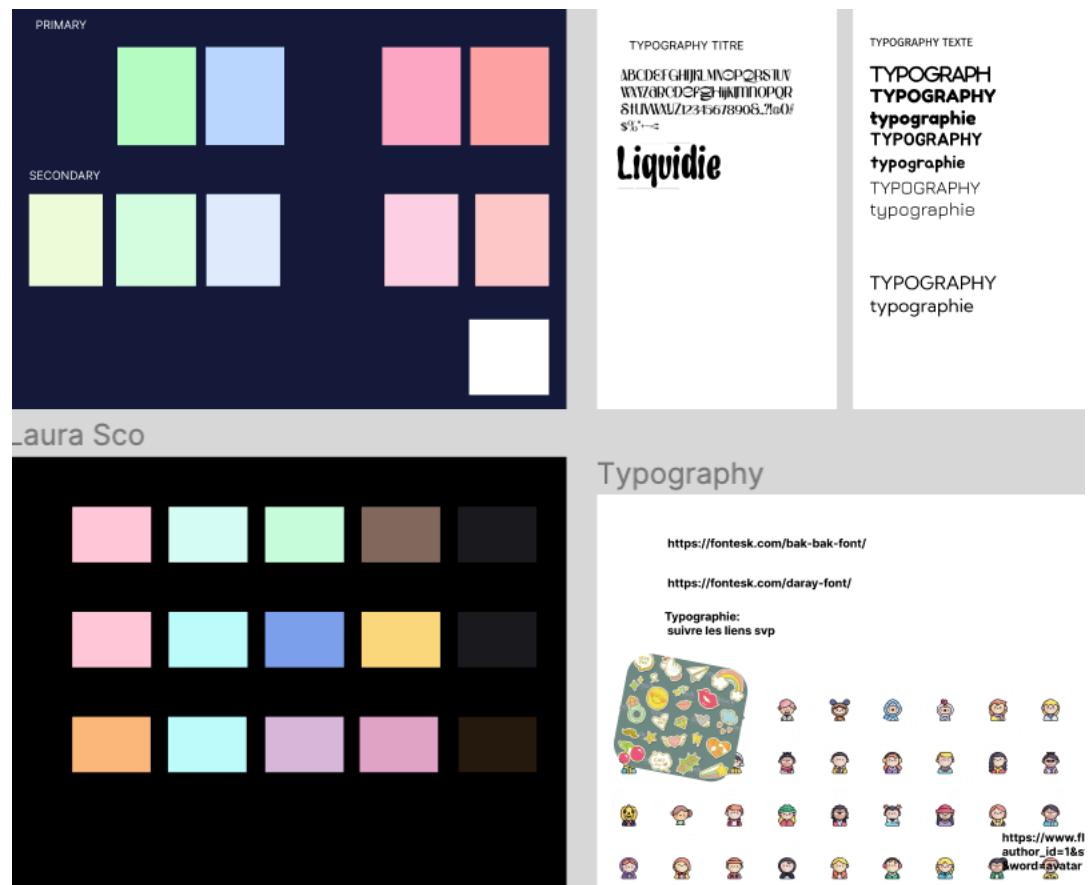


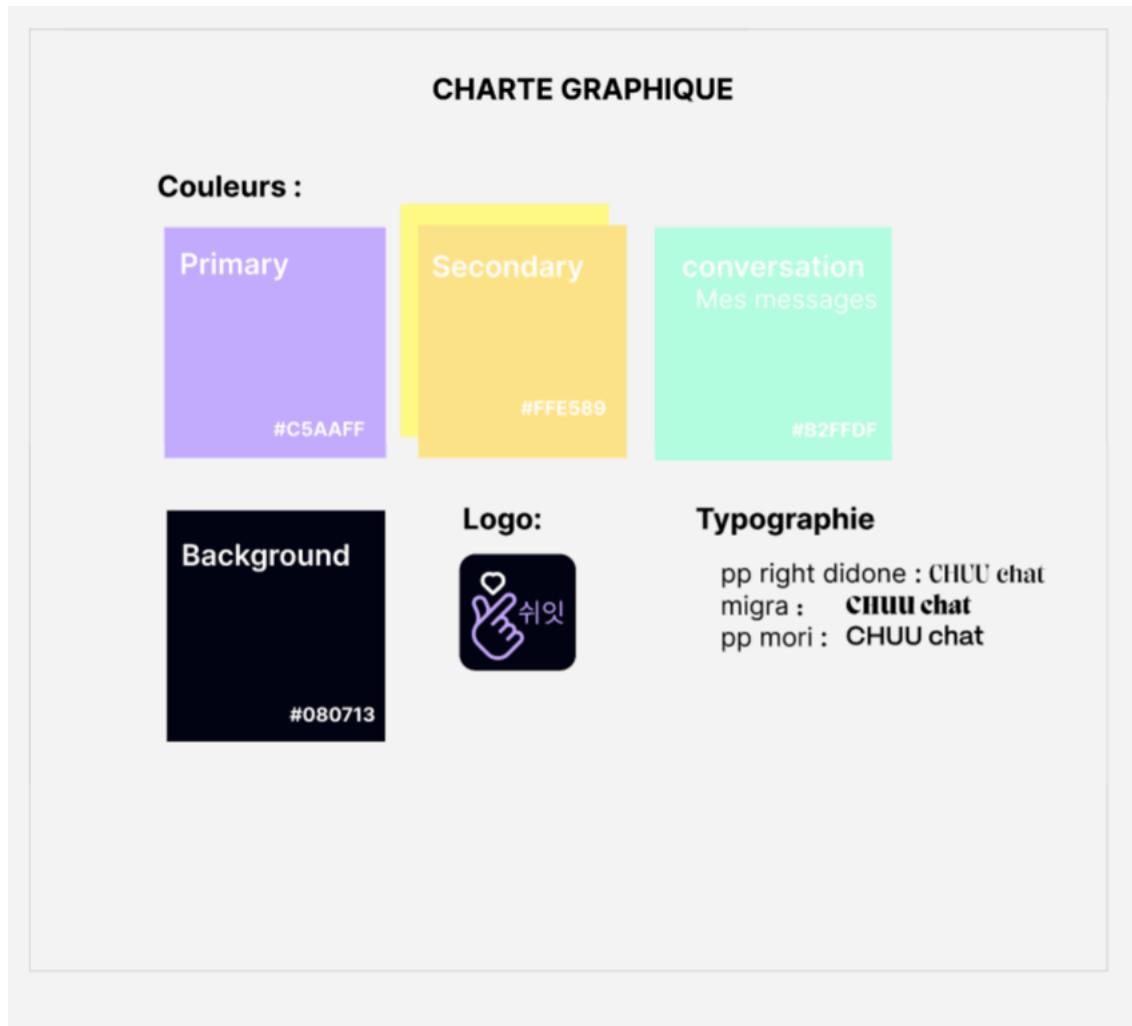
Moodboard

Pour guider la conception visuelle et trouver des inspirations, nous avions réalisé une petite collection d'images - un moodboard.

Charte graphique

Nous avons également réalisé une charte graphique pour nous guider dans la conception des maquettes. Nous y avons répertorié les couleurs que nous allions utiliser mais également les inspirations typographiques.





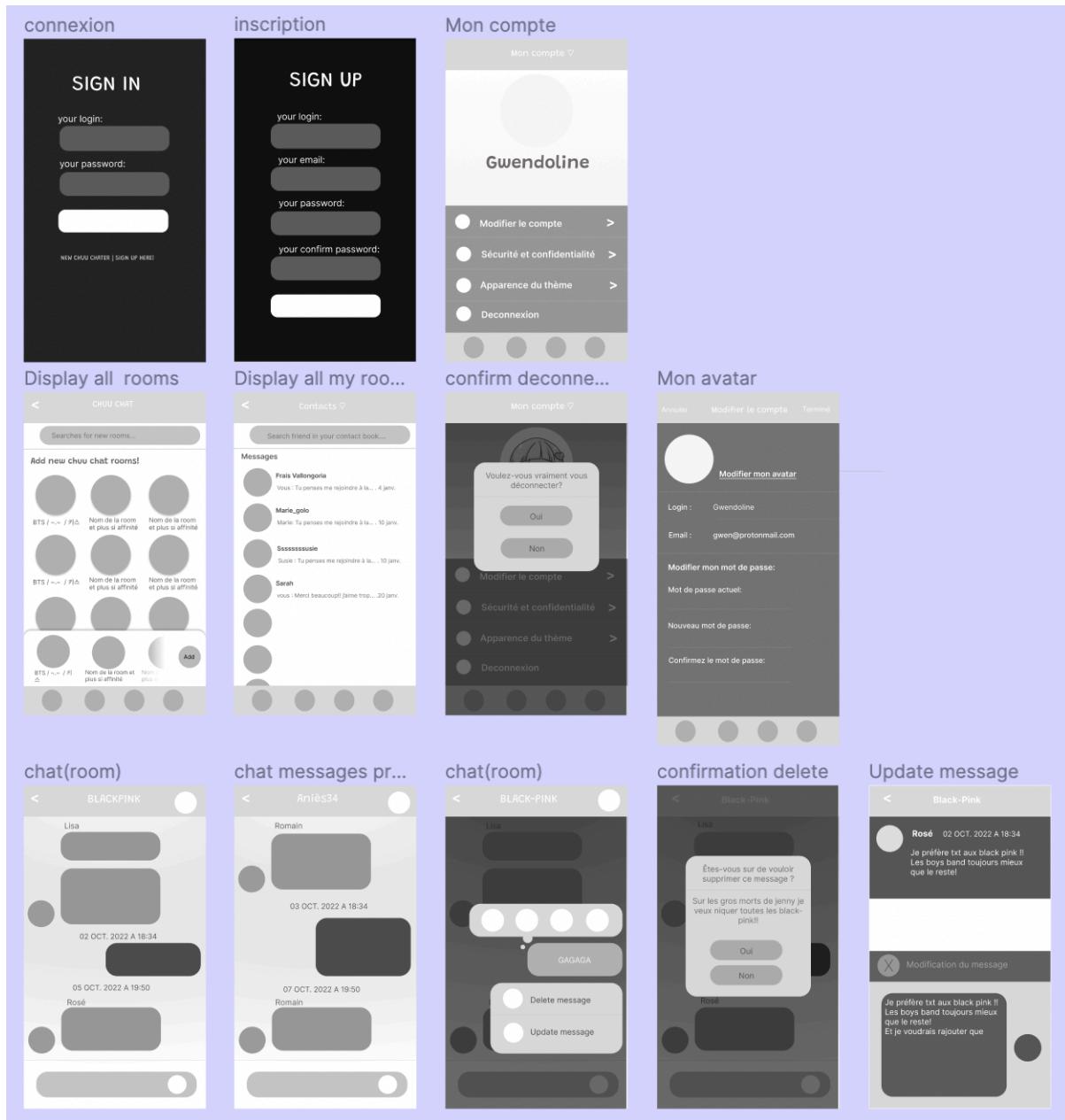
Wireframe

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.

Nous l'avons réalisé avec Figma mais il est tout à fait possible de le faire à l'aide d'un crayon sur une feuille de papier.

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).

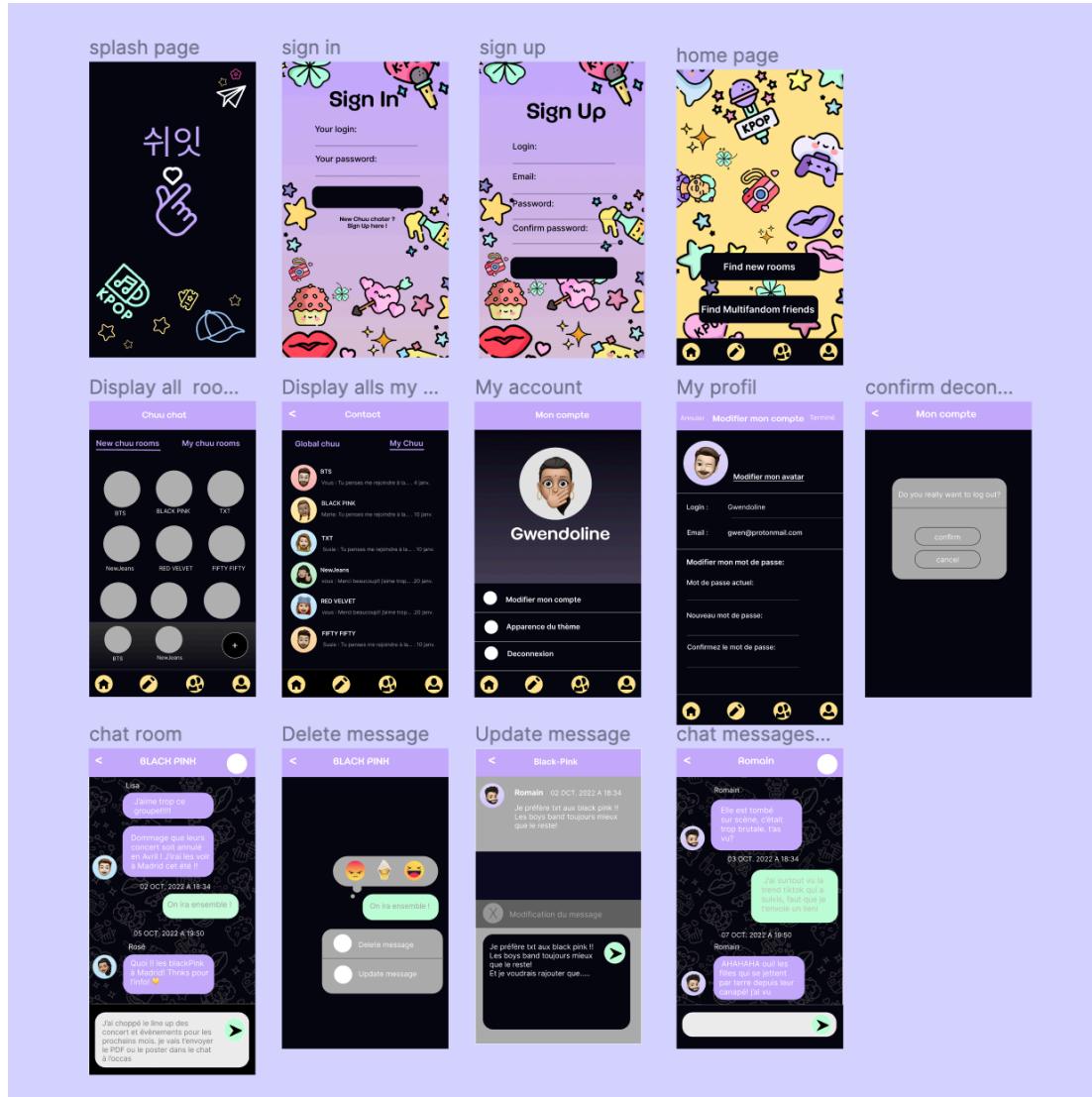


Maquette high-fidelity

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. Par exemple à l'aide de la mise en place de mixins.

Nous les avons réalisés à l'aide de Figma.



En ce qui concerne les choix technologiques du projet étant réalisé en React, nous avons voulu utiliser les naturelles dispositions du langage pour moduler notre front.

Composants

Une architecture basée sur les composants a été réalisée, puisque particulièrement populaire avec React, afin de créer des composants indépendants et réutilisables.

Les caractéristiques principales étaient d'avoir des composants qui puissent être :

- autonomes : une modification apportée à un composant n'influence pas ses compères.
- réutilisables
- remplaçables : par d'autres composants plus fonctionnels par exemple, dans le futur de l'application.
- maintenables

Chaque composant devait réaliser une tâche uniquement. Quelques composants, qu'on appelait composants génériques, étaient également codés pour servir d'autres composants (exemple, des boutons etc...).

Dans React Native, les architectures basées sur les composants sont généralement organisées en utilisant des dossiers pour regrouper les fichiers liés à chaque composant. Les dossiers peuvent contenir des fichiers de code source, des fichiers de test, des fichiers de style et des fichiers de documentation pour chaque composant.

Dans notre cas, nous avions le code source ainsi que son fichier de style.

Expo

Afin de simuler notre application, nous avons utilisé Expo. Il suffisait d'être connecté sur le même réseau, à la fois sur le mobile et l'ordinateur, pour que cela fonctionne.

IV. Fonctionnement

1. Fonctionnement de l'API

Le fonctionnement de l'API est basé sur une architecture de type requête/réponse. Le client envoie une requête à mon API via une url, dite requête HTTP (composées d'une ligne de commande, d'en-têtes et d'un corps de message) - cette dernière est analysée par le routeur qui répertorie toutes les requêtes possibles. En fonction de la route, un model/controller est appelé qui se chargera de récupérer ou d'envoyer une donnée.

Le serveur renvoie une réponse sous forme de JSON et également un statut.

Les différents statuts utilisés dans ce projet sont :

- 200 : OK

Indique que la requête a réussi

- 201 : CREATED

Indique que la requête a réussi et une ressource a été créée

- 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

- 403 : FORBIDDEN

Indique que le serveur a compris la requête mais ne l'autorise pas

- 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.

```
return res.status(400).send({ message: "The login or the password is invalid" });
```

2. Routing

Le routeur utilisé est celui d'express. Il se forme simplement d'un URI et d'une requête (get, post, etc) : "app.requête(route, fonction)" .

“App” est une instance d’express, la “requête” est la requête http, la route se présente sous forme d’uri et la fonction est le controller qui traite la donnée et renvoie une réponse au client.

```
//[BACK/03-get-all-users]: Une route qui
router.get('/', getUsers);
```

Nous avons dans un premier temps, un index.js qui recense l’ensemble des routes principales. La première ligne indique donc à l’application que pour toutes les routes ayant un URI commençant par “/users”, il faudra aller chercher la suite dans le dossier users.

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

```
//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);
```

Ce dernier comporte l’entièreté des routes, requêtes possibles, offertes pour la partie “users”. Cela nous évitait également de devoir préciser /user à chaque route users.

Cette requête commence par une instance du routeur d'express que nous appelons plus haut dans le fichier.

```
var router = express.Router();
```

La route permet donc de “get”, d’obtenir des données dès qu’on demandait /users (ici tous les utilisateurs). Lorsque cette requête est lancée, le routeur va trouver le controller correspondant avec la fonction indiquée, ici getUsers.

```
var {
  registerUsers,
  authUsers,
  connectedUser,
  addUserToRoom,
  getUsers,
  getUserDetails,
  getUserRole,
  updateUser,
  refreshToken,
  getAllFromUsers
} = require('../controllers/usersController')
```

```
const getUsers = (req, res) => {
  const sql = 'SELECT `login` FROM users'
  db.query(sql, function (error, data) {
    if (error) throw error;
    else res.send(data);
  })
}
```

Il est important de noter que la manière dont une requête est réalisée de la sorte, nous avons d’abord le port et l’URI qui suit.

<http://localhost:3000/admin/users/1/update/role>

3. Middleware

Le middleware est un logiciel qui permet une connectivité entre au moins deux applications ou des composants d’application. Dans notre projet, il prend la forme d’une fonction qui se dote également d’un rôle précis, fournit un service précis. Se situant entre deux couches logicielles, dans ExpressJS on le retrouve généralement entre la requête et la

réponse. De base, le framework possède quelques middleware mais il est possible d'en créer au besoin de l'application.

D'ordre général, il peut configurer et contrôler les connexions et les intégrations, gérer le trafic de manière dynamique via des systèmes distribués et sécuriser les connexions et le transfert de données. C'est principalement cette dernière fonctionnalité qu'on a utilisé pour notre projet.

Un middleware peut être appliqué à une unique route ou à plusieurs.

```
//[BACK/07] get the details from 1 user
router.get('/details/:userId', signIn, getUserDetails);
```

Ici, le middleware en question est signIn - avant de se diriger vers le controller attendu, la requête va être mise "en suspens" pour que des données nécessaires soient vérifiées.

On ne veut pas que n'importe qui ait accès aux détails d'un utilisateur, on veut seulement que les membres inscrits et connectés puissent le faire. SignIn nous permet de le faire en vérifiant le token de l'utilisateur (s'il y a bien un token).

```
const mySecret = "mysecret";
const decoded1 = jwt.verify(tokenToUse, mySecret);
req.user = decoded1;
const decoded2 = jwt.decode(tokenRefresh)
var now = new Date().getTime() / 1000;
```

Dans le cas où ce n'est pas le cas, la requête se verra automatiquement échouée.

```
} catch (err) {
  // console.log('auth.js : ', err);
  return res.status(401).send(err);
}
```

Cela permet également de rajouter une couche de sécurité et de pouvoir rediriger les utilisateurs de l'application vers les services disponibles selon leur statut.

4. Controller

Le rôle classique du controller est de gérer les réponses renvoyées par le model, les traiter et les envoyer sur la view. Il fait office d'intermédiaire entre les deux pôles, ainsi qu'entre le serveur et le client.

Dans notre application, nous avons décidé de nous passer de model et de tout rassembler dans le controller pour une raison de simplicité principalement. Alors que le model s'occupe de réaliser les demandes à la base de données d'ordinaire, nous avons intégré ceci dans le controller.

```
const getAllFromUsers = (req, res) => {
  const sql = 'SELECT * FROM users'
  db.query(sql, function (error, data) {
    if (error) throw error;
    else res.send(data);
  })
}
```

Aussi ce dernier détient et envoie non seulement la requête sql mais également s'occupe de la traiter s'il est nécessaire.

La plupart de nos fonctions sont similaires à celle présentée au-dessus, à savoir qu'il n'y a qu'une seule requête et un traitement minime (justifiant encore une fois notre décision d'utiliser que le controller). Il arrive que le traitement doit être plus spécifique comme ci-dessous.

```
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 this room.' })
        })
      } else res.status(400).send({ message: 'You were ban of this room.' })
    }
  })
}
```

Dans cette partie, un utilisateur peut s'ajouter à une room seulement s'il n'est pas banni de celle-ci. Dans un MVC classique, tout le traitement de la data aurait été dans le controller seulement.

5. Sécurité

Nous avons vu précédemment comment les middlewares permettaient une première sécurisation. En effet, ils permettent, principalement, dans notre application de vérifier les accès.

Il y a également une mise en place de rôles. En effet, il y en a trois disponibles : l'utilisateur lambda, l'administrateur et l'utilisateur banni - selon l'attribution de ce dernier, l'utilisateur aura différentes possibilités et droits au sein de l'application. Cela se présente sous forme d'un id_role qui est de 1 pour un utilisateur lambda, de 2 pour l'admin et de 0 pour un banni.

Il faut savoir que dans le cadre de l'administrateur, les deux points évoqués sont intimement liés :

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

Effectivement, pour ce dernier, nous avons mis en place un middleware spécifique.

```
exports.isAdmin = (req, res, next) => {
  console.log(req.body)
  console.log('req id role admin', req.body.id_role_admin)

  if (parseInt(req.body.id_role_admin) === 2) {
    return next();
  } else {
    return res.status(400).json({ message: "You are not an administrator" });
  }
}
```

Ici on vérifie l'id du rôle.

Évidemment, de manière classique nous avons les vérifications au niveau de la connexion : login et mot de passe correspondants.

```

const authUsers = (req, res) => {
  const login = req.body.login;
  const password = req.body.password;

  db.query(`SELECT users.id, users.login, users.email, users.id_role, users.password, GROUP_CONCAT(participants.
    console.log("results1: ", results)
    console.log(results.length);
    if (results.length > 0) {
      console.log('compare bcrypt:', bcrypt.compareSync(password, results[0].password))
      if(bcrypt.compareSync(password, results[0].password)) {
        const rooms = results[0].rooms?.split(',')

```

Mais il y a également, la mise en place de bcrypt pour le password notamment. Bcrypt permet le hachage du mot de passe en base de données : au lieu d'avoir le mot de passe directement rentré dedans, il est transformé pour éviter la fraude par force brute.

jennie \$2b\$10\$7tMGWzG2VhbwZW8HAfDFr.phidwV^k44pgFA3RqDxS^Zo...

Aussi, il est important de le déchiffrer lors de la vérification du password lors du login.

Enfin, la nouvelle instance de sécurité emmenée est le JWT aussi dit Json Web Token. Le Json Web Token (JWT) est un standard ouvert (protocole de communication) qui échange des informations entre deux parties sous forme d'objet JSON. L'information est signée de manière digitale via un algorithme HMAC ou RSA.

Le jeton fourni est composé de trois parties :

- Un en-tête (**header**), utilisé pour décrire le jeton. Il s'agit d'un objet JSON.
- Une charge utile (**payload**) qui représente les informations embarquées dans le jeton. Il s'agit également d'un objet JSON.
- Une **signature** numérique.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
 eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
 4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fu4

On peut le décoder grâce au site jwt.io.

Ce jeton est généré lors de la connexion de l'utilisateur : ce dernier entre ses informations et une demande est envoyée au serveur afin de fournir le token. Ce token sera alors vérifié lors des actions réalisées par l'utilisateur : il contient ses renseignements (son id, login, son rôle, ses rooms) et permet de s'assurer que l'utilisateur a des droits d'accès sur certaines rooms etc.

Il a également une durée de vie - nous étions donc obligés de le régénérer.

```
const tokenToUse = req.headers.token1;
const tokenRefresh = req.headers.refreshtoken;
```

Généralement le token et son refresh se trouvent dans le header de la requête.

```
const mySecret = "mysecret";
const decoded1 = jwt.verify(tokenToUse, mySecret);
req.user = decoded1;
var now = new Date().getTime() / 1000;
```

Une fois récupérés, on vérifie le token (const decoded1 - s'il y a une erreur on atterrit dans le catch directement) puis on s'occupe principalement du refreshToken : voir s'il a expiré ou pas.

```
//verification avec la date actuelle, si l'expiration
const decoded2 = jwt.verify(tokenRefresh, mySecret)
var now = new Date().getTime() / 1000;

//mettre decoded2.iat et pas .exp
if (now > decoded2.exp) {
    /* expired */
    //le token est disponible ds le scope grâce au callback ds refreshToken du UsersController
    return refreshToken(decoded1.id, token => {
        // console.log('first')
        res.status(417).send(token)
    });
}

next();
```

Dans le cas où le refresh est périmé, on relance un nouveau refreshToken.

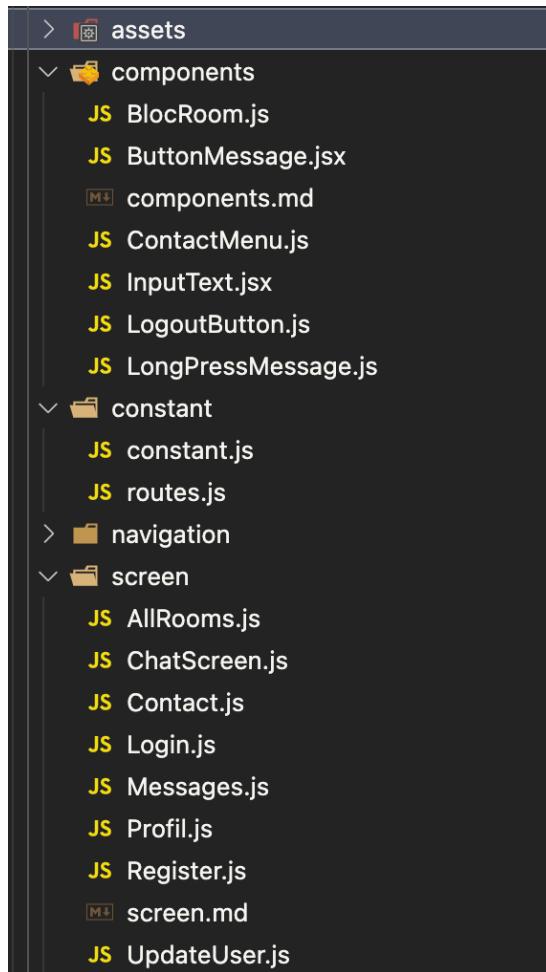
Le token a été une nouvelle problématique avec laquelle on devait jouer - il y a eu malheureusement beaucoup de problèmes pour le mettre en place et l'appliquer correctement. Cependant, voici quelques pistes d'amélioration qu'avec le recul, nous allons pouvoir mettre en place afin d'utiliser le JWT de manière un peu plus correcte :

- Si un utilisateur effectuait une action qui nécessitait une mise à jour de l'état du compte, comme la suppression, le blocage ou la suspension du compte, nous pourrions invalider le JWT associé à cet utilisateur en maintenant une liste blanche ou une liste noire des JWT valides/invalides. Ainsi, à chaque requête, nous vérifierons si le JWT fourni est toujours valide.
- Pour limiter la fenêtre d'exposition en cas de changement d'état du compte ou de modification des autorisations, nous pourrions définir des durées de validité plus courtes pour les JWT. Cela nécessiterait une régénération plus fréquente des JWT, mais offrirait une meilleure réactivité en cas de changements importants.

II. Fonctionnement du front

1. Arborescence

Dans le dossier src, nous avions divisé le travail en plusieurs sous-dossiers :



Notre dossier components avait les composants réutilisables. Navigation comprenait la navigation de l'application et le screen les “pages”.

2. Composants

Notre front a été pensé en React, ce qui induit donc l'utilisation de composants. Un composant est un bout de code indépendant et réutilisable. Il marche généralement comme une fonction. Il existe des composants en class mais aussi en fonctions - dans le cadre de notre application, nous avons utilisé les derniers, étant recommandés puisque préférés dernièrement dans le développement.

Nous avons donc divisé nos composants en deux catégories : la première comprenait les composants réutilisables (tels que des boutons etc...), la seconde comprenait les composants plus grands (les screens) qui utilisaient donc les premiers et transmettaient les informations nécessaires via des props.

```

export default function BlocRoom({name, room, id, pressRoom, deletePress, specialClass, touchable, tb, disabled, ...}) {
    const blocRoomPress = () => {
        // console.log('room id', room.id)
        pressRoom({'id':room.id, 'name':name})
    }

    const deleteRoom = () => {
        deletePress(id)
    }

    return (
        //ajouter dans le press qu'on peut aller dans les messages de la room également si c'est notre room
        <TouchableOpacity
            style={styles.container}
            onPress={() => {deletePress ? deleteRoom() : blocRoomPress()}}
            // disabled={}
        >
            {/* Image */}
            <View style={styles.tinyIcon}>
                <Image
                    style={styles.img}
                    source={{uri: 'https://64.media.tumblr.com/6b9e50e7237cf04fd44b6f56ce75e848/04f19f3b5d21afac-b
                    />
                </View>
                <Text style={styles.name}>{name}</Text>
            </TouchableOpacity>
    )
}

```

Par exemple, ici nous avons un composant réutilisable. Il s'occupait de former les petites bulles de présentation des rooms. Ce composant était réutilisé dans la même page selon des states différents (il a été cliqué ou non).

III. Développement

Un fois la conception terminée et les bases posées, chacun pouvait alors commencer à développer selon les tickets attribués.

Dans cette partie je vais me concentrer principalement sur le code et son explication.

1. Extrait de code

1. Navigation

Pour la navigation, nous avons utilisé React Navigation, notamment Stack qui ajoute un effet de “stacking” entre les pages (installation via npm).

```

return () =>
  <Stack.Navigator
    screenOptions={{
      headerStyle: {
        backgroundColor: '#C5AAFF',
      },
    }}
    initialRouteName={loggedIn ? ROUTES.HOME : ROUTES.LOGIN}>
    <Stack.Screen
      name={ROUTES.HOME}
      component={TabBar}
      options={{ headerShown: false }}>
    />
    <Stack.Screen name={ROUTES.LOGIN} component={Login} options={{ headerShown: false }}/>
    <Stack.Screen name={ROUTES.PROFILE} component={Profil} />
    <Stack.Screen name={ROUTES.REGISTER} component={Register} options={{
      headerShadowVisible: false, // applied here
      headerBackTitleVisible: false,
      headerTintColor: 'black'
    }}/>
  </Stack.Navigator>
];

```

La manière dont le navigateur marche est principalement grâce à des noms et des composants (screens) associés.

Le stack.navigator est le composant de base où les screens sont répertoriés. Le notre prend en props initialRouteName qui permet de choisir un nom à la route rendered en premier. On a également un screenOptions qui permet de personnaliser le navigateur.

Ensuite, s'empilent les stack.screens : chacun a un nom et un composant associé - on peut également y ajouter des options comme ici en disant qu'on ne veut pas que le header soit visible. En effet, pour certaines pages, il ne nous était pas utile.

C'est le navigateur à son état le plus basique qui permet une passation entre les pages selon les "press" de l'utilisateur : c'est celui qui permet par exemple dans la page "messages" de cliquer sur une room utilisée par l'utilisateur et d'atterrir sur la page des messages de cette room.

Cependant, notre navigateur a également une tab-bar située sur le bas pour une accessibilité directe avec le pouce de l'utilisateur. Aussi, nous l'avons codé de la sorte :

```

<Tab.Navigator {...{screenOptions}} >
  <Tab.Screen name={ROUTES.CHATROOMS} component={ChatRoomStack} options={{
    tabBarIcon: ({focused}) => (
      <View style={{alignItems: 'center', justifyContent: 'center',}}>
        <Image
          source={require('../assets/icons/chuu-purpl.png')}
          resizeMode='contain'
          style={{
            width: 35,
            height: 35,
            tintColor: focused ? '#B2FFDF' : '#ADADAD'
          }}
        />
        <Text style={{color: focused ? '#B2FFDF' : '#ADADAD', fontSize: 10}}>
          rooms
        </Text>
      </View>
    ),
    headerShown:false
  }}>

```

De la tab-bar, l'utilisateur pouvait avoir accès à ses rooms, ses messages ainsi qu'à son profil. Le concept est le même : un navigator contient des screens avec un nom qui renvoient à des composants (screens) particuliers.

2. 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.

```

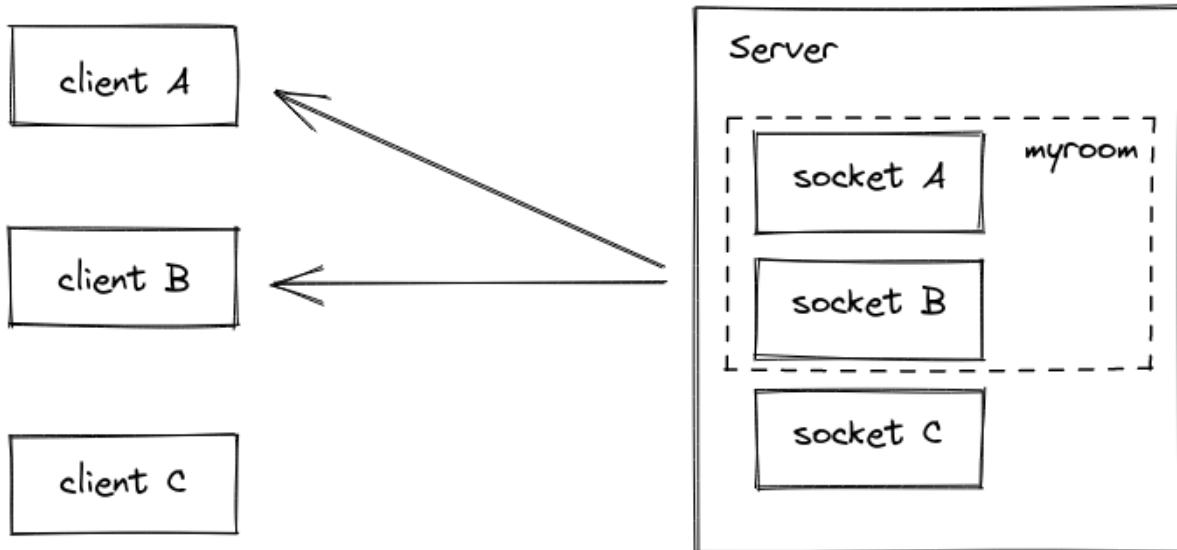
const io = require('socket.io')(server)

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".



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.

```
import { io } from 'socket.io-client';
// ...
const [socket, setSocket] = useState(io("http://localhost:3000"));

useEffect(() => {
  socket.emit('joinIn', route.params.id_room)
<Messages style={{ height: '40%' }} idRoom={route.params.id_room} socket={socket} />

```

Puis, dans le composant Message, on écoutait l'évènement “newMessage”.

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

Ce dernier était émis depuis notre contrôleur 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'évènement “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.

3. Contact page : get answers from the API

Semblable à un jeu d'essai, je développe dans cette sous-partie une fonctionnalité de notre application. Lorsqu'un utilisateur se connecte à l'application, s'il participe à des rooms, il est redirigé vers une page qui répertorie toutes ses chatrooms et le dernier message envoyé dans chacune de ces rooms.



Pour ce faire, dans le screen contact.js, nous appelions l'API et faisions une requête http (ci-dessous "url") au serveur depuis le client.

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

L'uri était le chemin que devait prendre le routeur pour retrouver le controller correspondant.

```
uri = "/rooms/contact"
```

Tandis que l'API était une constante qu'on importait.

Le routeur allait donc à l'index.js qui répertorier l'ensemble des "grandes routes".

```
//Rooms route
app.use('/rooms', rooms);
```

Ensuite, il retrouvait la route correspondant aux rooms.

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

Et dans cette dernière, il retrouvait enfin le controller demandé.

```
// route get dernier message, d'un chat name etc...dans lequel le participant participe
router.get('/contact', signIn, displayRoomsAndChat);
```

Dans roomsController.js, on trouvait donc :

```
// get dernier message, d'un chat name etc...dans lequel le participant est
const displayRoomsAndChat = (req, res) => {
  const sql = `SELECT rooms.id, rooms.name, messages.content FROM messages CROSS JOIN rooms ON messages.id_room = rooms.id WHERE rooms.id = ${req.params.id} ORDER BY messages.id DESC LIMIT 1`;
  db.query(sql, function(error, data){
    if (error) throw error;
    else res.status(200).send(data);
  })
}
```

S'il y avait une erreur, elle était traitée. Dans le cas contraire, le serveur renvoyait une réponse positive et les informations demandées par le côté client.

Du côté client, nos contacts étaient un useState alors on “set”-tait (setContacts) la réponse fournie par l’API dans notre state.

```
contacts.length > 0 ?
  contacts.map((contact) =>
    <ContactMenu
      key={contact.id}
      contact={contact}
      onPress={() => navigation.navigate(ROUTES.MESSAGES, {id_room: contact.id, room_na
    />
  )
  :
<View style={styles.container2}>
  <Text style={styles.text}>
    You are not subscribed to any rooms.
  </Text>
  <TouchableOpacity style={styles.cta}>
    <Text style={styles.textCta}> Get a room </Text>
  </TouchableOpacity>
</View>
```

Si la réponse était “vide”, donc que l’array contacts se trouvait vide (l’utilisateur ne participait à aucune room), on envoyait une view simple avec un texte lui indiquant qu’il n’avait aucune chatroom.

```
<ScrollView style={styles.bg}>
  <View style={styles.tabs}>

    <TouchableOpacity onPress={() => underlined(2)}>
      <Text style={underline === 2 ? styles.selected : styles.notSelected}>More bands</Text>
    </TouchableOpacity>
    <TouchableOpacity onPress={() => underlined(1)}>
      <Text style={underline === 1 ? styles.selected : styles.notSelected}>My chuu bands</Text>
    </TouchableOpacity>
  </View>
  <View style={styles.container}>
    {
      rooms.length >= 1 ?
        rooms.map((room) =>
          <BlocRoom
            key={room.id}
            name={room.name}
            room={room}
            tab={underline}
            pressRoom={setNewRooms}
            // specialClass={moreRooms.find(moreRoom => moreRoom.id === room.id) ? true : false
            // touchable={touchable}
            tb = {false}
            disabled = {disabled}
            moreRooms = {moreRooms}
          />
        )
        :
        <Text>No rooms.</Text>
      }
    </View>
```

Ici par exemple, nous avons un “composant” qui est réellement une page entière. On peut d’ailleurs remarquer que BlocRoom y est utilisé, il est instancié plus haut par import.

```
import BlocRoom from '../components/BlocRoom';
```

Il y a, néanmoins, une différence notable entre ReactJS et React Native. En effet, le React que je mentionne tout au long du dossier est React Native. React Native est utilisé pour la création d'applications mobiles qui s'adaptent aussi bien en IOS qu'en android tandis que ReactJS est plutôt utilisé pour les applications web qui nécessitent donc un browser. La raison étant que ReactJS se sert de Javascript, CSS et HTML pour construire les interfaces lorsque React Native use en plus des composants natifs comme on peut le voir sur les captures d'écran du dessus : l'un des plus basiques est le composant <View> qui est un conteneur capable de supporter les flexbox, style, contrôle d'accessibilité etc.

Aussi, un détail important à noter est que le style de nos composants était rédigé sur le même fichier et non sur un fichier css dédié. Pour ce faire, on avait créé une constante styles qui était au final une fonction avec un retour d'objet que l'on parcourait en donnant la clef souhaitée (classname) à notre composant natif.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    marginTop: 52,
    alignItems: 'center'
  },
  boxTitle: {
    marginTop: 90,
  },
  boxForm: {
    marginTop: 60,
  }
});
```

```
<Text style={styles.title}>
```

2. Tests

Les tests en développement permettent de vérifier que les fonctionnalités développées marchent correctement et que s'il y a des erreurs, ces dernières sont traitées via le code. Il existe plusieurs tests qui peuvent être aussi bien manuels qu'automatisés :

- 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.

Pour un projet aussi minime, nos tests n'étaient pas forcément les plus développés. Néanmoins, nous avons utilisé Postman pour tester notre API et ses requêtes manuellement.

Dans un premier temps, nous avons créé la collection Postman recensant l'ensemble des appels possibles vers notre API.

Collection Postman de l'API :

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

- Header Bar:** Home, Workspaces, API Network, Explore, Search Postman, Invite, Settings, Upgrade.
- Left Sidebar:** Collections (Chuu Chat), Environments (chuu localhost), History.
- Request Panel:**
 - Method:** POST
 - URL:** {{baseURL}}/users/auth
 - Body:** Contains a script to set tokens from the response.
 - Environment:** chuu localhost variables (login: Naomi, password: Naomi999).
- Response Panel:**
 - Variables:** authToken and refreshToken with their respective values.
 - Global Variables:** None.
 - Status:** 200 OK, Time: 138 ms, Size: 896 B.
- Bottom Navigation:** Body, Cookies, Headers (9), Test Results, Pretty (selected), Raw, Preview, Visualize, JSON.
- Collection Tree:**
 - Chuu Chat
 - USERS
 - POST SIGN IN
 - GET VERIF SIGNIN
 - POST INSSCRIPTION
 - GET GET ALL USERNAMES
 - POST UPDATE USER INFO
 - PARTICIPANTS
 - ROOMS
 - GET GET UNSUBSCRIBED ROOM...
 - GET GET ALL ROOMS
 - GET GET LAST MESSAGE FROM ...
 - GET GET LAST MESSAGE FROM ...
 - MESSAGES
 - POST POST MESSAGE IN GLOBAL ...
 - POST POST MESSAGE IN CHATRO...
 - DEL DELETE MESSAGE
 - POST EDIT MESSAGE
 - GET GET MESSAGES BY ROOM
 - GET GET MESSAGES FROM GLO...
 - ADMIN
 - DEL (ADMIN) DELETE MESSAGE ...
 - DEL (ADMIN) DELETE ROOM
 - POST (ADMIN) CREATE ROOM
 - PATCH (ADMIN) UPDATE ROOM
 - PATCH (ADMIN) UPDATE USER NAME
 - PATCH (ADMIN) UPDATE USER ROLE

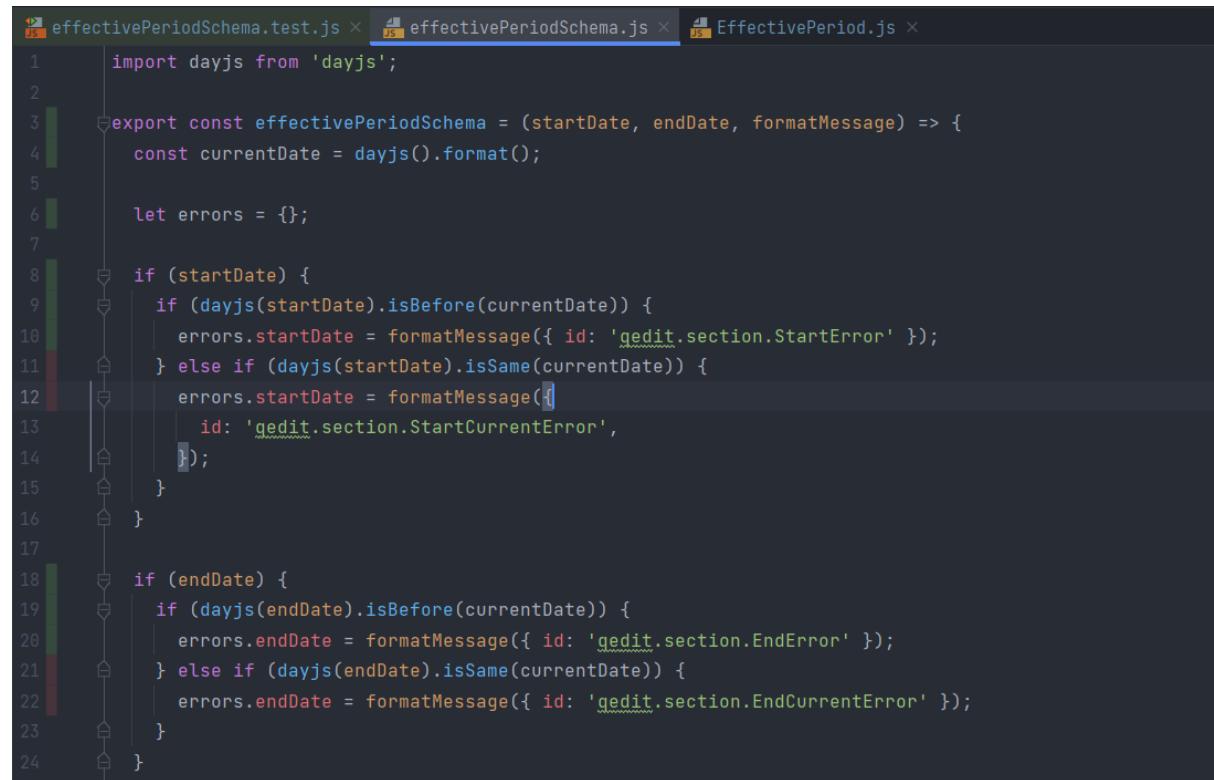
Afin de fluidifier le processus de test, nous avons créé 2 environnement Postman contenant le nom de domaine de l'API ainsi que les informations de l'utilisateur connecté. Cela permet alors de passer d'un utilisateur à un autre sans avoir à modifier la collection.

L'outil "Tests" 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. 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.

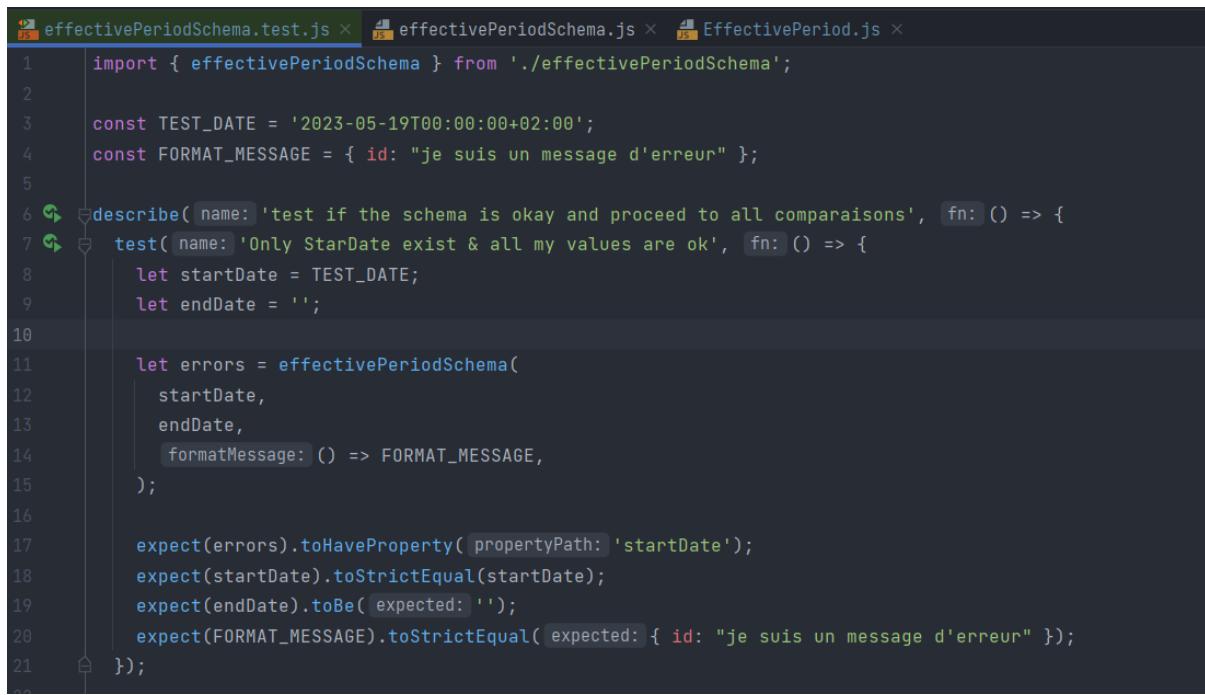
3. Test en alternance

En alternance au sein du projet Qbuilder j'ai eu l'occasions de faire des tests unitaires, plus précisement des test autours de fonctions qui peuvent être réutilisées au sein du code. Nous allons voir en détails le processus :



```
1 import dayjs from 'dayjs';
2
3 export const effectivePeriodSchema = (startDate, endDate, formatMessage) => {
4   const currentDate = dayjs().format();
5
6   let errors = {};
7
8   if (startDate) {
9     if (dayjs(startDate).isBefore(currentDate)) {
10       errors.startDate = formatMessage({ id: 'gedit.section.StartError' });
11     } else if (dayjs(startDate).isSame(currentDate)) {
12       errors.startDate = formatMessage({
13         id: 'gedit.section.StartCurrentError',
14       });
15     }
16   }
17
18   if (endDate) {
19     if (dayjs(endDate).isBefore(currentDate)) {
20       errors.endDate = formatMessage({ id: 'gedit.section.EndError' });
21     } else if (dayjs(endDate).isSame(currentDate)) {
22       errors.endDate = formatMessage({ id: 'gedit.section.EndCurrentError' });
23     }
24 }
```

Je vais tester au moyen de jest cette fonction si dessus elle utile une librairie dayJs et doit vérifier une période de temps valide.



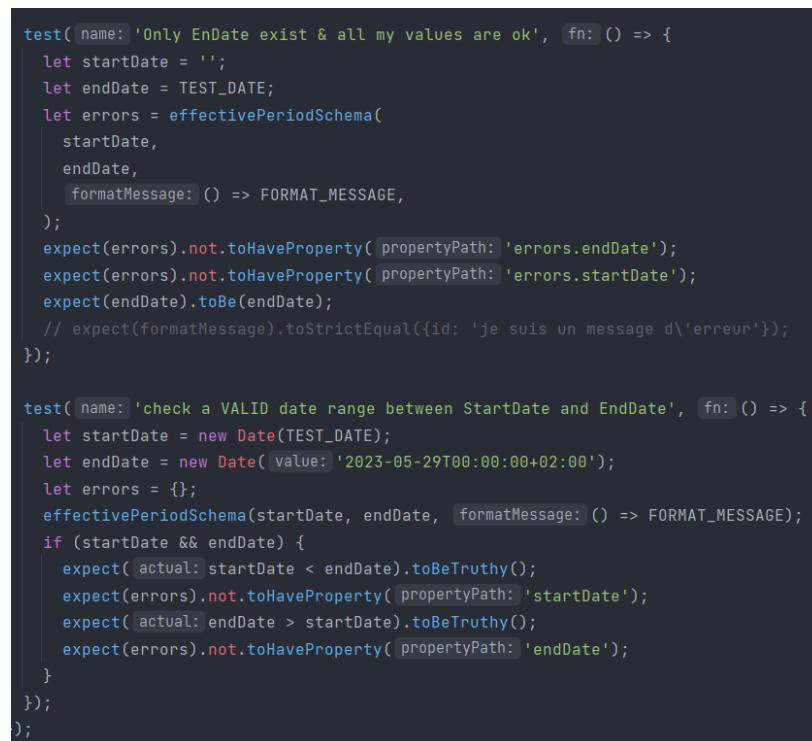
```
effectivePeriodSchema.test.js
1 import { effectivePeriodSchema } from './effectivePeriodSchema';
2
3 const TEST_DATE = '2023-05-19T00:00:00+02:00';
4 const FORMAT_MESSAGE = { id: "je suis un message d'erreur" };
5
6 describe( name: 'test if the schema is okay and proceed to all comparaisons', fn: () => {
7   test( name: 'Only StarDate exist & all my values are ok', fn: () => {
8     let startDate = TEST_DATE;
9     let endDate = '';
10
11     let errors = effectivePeriodSchema(
12       startDate,
13       endDate,
14       formatMessage: () => FORMAT_MESSAGE,
15     );
16
17     expect(errors).toHaveProperty( propertyPath: 'startDate' );
18     expect(startDate).toStrictEqual(startDate);
19     expect(endDate).toBe( expected: '' );
20     expect(FORMAT_MESSAGE).toStrictEqual( expected: { id: "je suis un message d'erreur" } );
21   });
22 });

effectivePeriodSchema.js
1 export default function effectivePeriodSchema( startDate, endDate, formatMessage ) {
2   const errors = {};
3
4   if ( !startDate || !endDate ) {
5     errors.endDate = 'Only EndDate exist & all my values are ok';
6     errors.startDate = 'Only StarDate exist & all my values are ok';
7   }
8
9   if ( !formatMessage ) {
10    errors.formatMessage = 'check a VALID date range between StartDate and EndDate';
11  }
12
13   if ( !formatMessage ) {
14     errors.endDate = 'check a VALID date range between StartDate and EndDate';
15   }
16
17   if ( !formatMessage ) {
18     errors.startDate = 'check a VALID date range between StartDate and EndDate';
19   }
20
21   if ( !formatMessage ) {
22     errors.formatMessage = 'check a VALID date range between StartDate and EndDate';
23   }
24
25   return errors;
26 }

EffectivePeriod.js
1 export { default as EffectivePeriod } from './effectivePeriodSchema';
```

Voici un cas de test je met un describe qui dois décrire au mieux mon but, il englobe plusieurs cas de test qui attendent ou simulent des retours de cette fonction testée.

Voici d'autre cas de figure au sein du même test:



```
test( name: 'Only EndDate exist & all my values are ok', fn: () => {
  let startDate = '';
  let endDate = TEST_DATE;
  let errors = effectivePeriodSchema(
    startDate,
    endDate,
    formatMessage: () => FORMAT_MESSAGE,
  );
  expect(errors).not.toHaveProperty( propertyPath: 'errors.endDate' );
  expect(errors).not.toHaveProperty( propertyPath: 'errors.startDate' );
  expect(endDate).toBe(endDate);
  // expect(formatMessage).toStrictEqual({id: 'je suis un message d\'erreur'});
});

test( name: 'check a VALID date range between StartDate and EndDate', fn: () => {
  let startDate = new Date(TEST_DATE);
  let endDate = new Date( value: '2023-05-29T00:00:00+02:00' );
  let errors = {};
  effectivePeriodSchema(startDate, endDate, formatMessage: () => FORMAT_MESSAGE);
  if (startDate && endDate) {
    expect( actual: startDate < endDate ).toBeTruthy();
    expect(errors).not.toHaveProperty( propertyPath: 'startDate' );
    expect( actual: endDate > startDate ).toBeTruthy();
    expect(errors).not.toHaveProperty( propertyPath: 'endDate' );
  }
});
```

Je vais dorénavant lancer mon test:

The screenshot shows a coverage report for a file named 'effectivePeriodSchema.test.js'. The top bar indicates 'Cover: 100% test if the schema is okay and proceed to all co...'. The main area displays a tree structure of test results. At the top level, there is a single green checkmark next to the file name. Below it, under 'Test Results', there is another green checkmark next to 'effectivePeriodSchema.test.js'. Underneath that, there is a green checkmark next to 'test if the schema is okay and proceed to all comparaisons'. Finally, under that, there is a green checkmark next to 'Only EndDate exist & all my values are ok'. To the right of each item, there is a timestamp '8 ms'.

Le test est bon car il passe tout les cas de figure, ici j'ai testé qu'un seul test je vais donc lancer tous les test avec un coverage pour avoir un aperçu de la couverture de test. Ainsi je peux voir si j'ai bien couvert l'ensemble des retours et/ou états que ma fonction peut générer, et ce grâce au différent cas que j'ai simulé grâce à Jest et ses méthodes.

Je relance avec tous les cas de test:

The screenshot shows a code editor with two files open: 'effectivePeriodSchema.js' and 'effectivePeriodSchema.test.js'. The code in 'effectivePeriodSchema.js' contains logic for validating start and end dates. The code in 'effectivePeriodSchema.test.js' contains three test cases: one for start date validation, one for end date validation, and one for both dates together. The bottom part of the screenshot shows a coverage report. The top bar says 'Cover: 100% test if the schema is okay and proceed to all co...'. The coverage summary at the bottom left says 'Tests passed: 3 of 3 tests - 41ms'. The coverage matrix on the right shows coverage percentages for various files: 'cleanNodes.js' (0%), 'cleanQuestionnaire.js' (0%), 'effectivePeriodSchema.js' (63.63%), and 'getContextVariables.js' (59.09%).

Tous mes tests sont passés et je peux voir quels conditions ont été vérifiées par mes expect mais aussi l'impact Ausein des autres fichiers du projet.

V. Conception de la partie administration

Lorsque l'ensemble du projet est réalisé pour être utilisé en tant qu'application mobile, la partie administration, elle, est plutôt une application web.

C'est à partir de cette dernière que l'administrateur aura une influence sur l'application mobile.

Pour avoir un accès à la partie administrateur, ce dernier doit déjà s'identifier en entrant le login et le password. Lorsque ces derniers sont vérifiés, on s'occupait alors de vérifier le rôle de l'utilisateur. En effet, un rôle "2" était synonyme d'être administrateur et donc pouvoir entrer dans la partie web.

```
if(roleAdmin[0].id_role === 2) {
    // store the token in local storage
    sessionStorage.setItem('token', json.token);
    sessionStorage.setItem('refresh_token', json.refresh);
    sessionStorage.setItem('id_admin', roleAdmin[0].id_role);
    window.location.href = front + "/app-mobile-chat-admin/admin/adminIndex.php";
}
```

L'administrateur était alors redirigé vers son index. Dans le cas contraire, il n'y avait pas accès.

On stockait également les tokens et le rôle de l'administrateur dans une sorte de session afin de pouvoir les utiliser plus facilement lors des différentes requêtes.

```
const bodyData = JSON.stringify(data)
const token = Token.get();
const refresh = refreshToken.get()
```

```
fetch(API + `/admin/users/${idUser}/update`, {
    method: 'PATCH',
    mode: 'cors',
    body: bodyData,
    headers: {
        'token1': token,
        'refreshtoken': refresh,
        'Content-Type': 'application/json',
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Credentials': 'true'
    },
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
}
```

Puisqu'en effet, l'identité de l'administrateur était vérifiée à chaque requête du côté client par un middleware au niveau serveur.

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

La connexion entre la web app et notre serveur était validée via les CORS (cross-origin resource sharing). En effet, notre client web app avait un port différent que celui du serveur de l'application.

Le CORS est un mécanisme HTTP-header qui permet à un serveur d'indiquer quelle origine (domaines ou ports) autre que la sienne peut accéder à ses ressources.

```
const cors = require('cors');
const corsOptions ={
    origin:['http://localhost:3000','http://localhost:8888','http://localhost:8881', 'http://localhost'],
    credentials:true, //access-control-allow-credentials:true
    optionSuccessStatus:200
}
app.use(cors(corsOptions));
```

VI. Problématiques rencontrées

Plusieurs spécificités techniques ont posé problème au cours du développement mais également la conception de l'application. Il y a eu bien sûr l'implémentation des sockets.io (et notamment comment dispatcher correctement les éléments : où écouter l'évènement etc... avec la bonne room), la navigation côté front aussi et les spécificités de React Native.

Il y a eu également, lors du développement du côté administrateur, les CORS qui pouvaient nous bloquer avec ce genre d'erreur :

```
✖ Access to XMLHttpRequest at 'http://localhost:5000/global config' step1:1
from origin 'http://localhost:8080' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
```

Il suffisait d'ajouter son port dans les autorisations.

Un autre souci était plutôt d'ordre social - ou comment coordonner une équipe de quatre personnes avec différentes alternances au cours d'une année segmentée. Bien que nous ayons mis en place la technique de l'agilité, il y a eu des petits ics et inconvénients qui ont joué un rôle défavorable quant à notre avancée. Que ce soit le désistement d'un membre, la baisse de motivation, ou encore des différends au niveau de l'organisation, il a fallu adapter l'avancement du projet et les différentes techniques employées selon le groupe.

VII. Recherches anglophones et documentation

La connaissance et la maîtrise de l'anglais est un outil fondamental pour un développeur web. Pour plusieurs raisons mais la principale étant que la plupart des outils qu'il

utilisera seront toujours d'abord en anglais lors de leur sortie, ce qui signifie que les mises à jour le seront aussi ainsi que les instructions.

Une autre raison est pour les recherches. Beaucoup seront à faire en anglais pour trouver les dernières façons de faire et celles plus détaillées.

Que ce soit des vidéos anglaises ou simples recherches sur mon navigateur, j'ai pour ma part principalement tout réalisé en langue anglaise.

Aussi sur ce projet, de nouvelles compétences ont dû être mises en jeu - il a fallu également se renseigner sur de nouveaux outils.

Par exemple, les sockets dont la documentation socket.io est en anglais :

How it works

The bidirectional channel between the Socket.IO server (Node.js) and the Socket.IO client (browser, Node.js, or another programming language) is established with a [WebSocket connection](#) whenever possible, and will use HTTP long-polling as fallback.

The Socket.IO codebase is split into two distinct layers:

- the low-level plumbing: what we call Engine.IO, the engine inside Socket.IO
- the high-level API: Socket.IO itself

D'autre part, Stackoverflow est une source majeure pour un développeur. Sur cette plateforme, il peut non seulement poser des questions, présenter ses problèmes mais aussi répondre et recevoir lui-même des réponses. C'est donc un outil important et qui est majoritairement en anglais.

Conclusion

Ce projet est un résumé global de mon année et de toutes les choses que j'ai pu apprendre. Même les difficultés rencontrées m'ont permis de m'améliorer sur des compétences primordiales en tant que développeur mais également de travailler l'amont : la conception.

J'ai pu réaliser, avec mon équipe, une veille technologique sur différents principes comme React Native, NodeJS etc... J'ai pu également pousser une vision plus professionnelle, c'est-à-dire qu'on a tenté avec notre groupe de nous rapprocher le plus possible de ce qui

pourrait être potentiellement fait dans nos entreprises d’alternance respectives et de trouver un pont entre les deux mondes.

Bien que pouvant être grandement amélioré, jongler entre ce projet et les différentes demandes du monde professionnel dans notre alternance a été un challenge de taille que nous avons surmontés grâce aux techniques de conception employées mais également grâce à notre charte de travail établie au préalable.