

DOSSIER DE PROJET PROFESSIONNEL



» **La Plateforme**

Réalisation d'une application mobile de chat

-
Lucas Verdier

Introduction

Je m'appelle Lucas Verdier, j'ai obtenu mon diplôme de "Développeur web et web mobile" en 2022. Cette année, je poursuis ma formation au sein de La Plateforme pour obtenir le diplôme de "Concepteur Développeur d'Application". J'effectue mon année en alternance dans la société ViaXoft à Marseille. Une entreprise qui produit des applications web qui accompagnent les entreprises du tourisme dans leur métier.

Présentation du projet en Anglais

Compétences couvertes par le projet

Le projet présenté couvre les compétences du REAC suivantes :

- 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.
- Concevoir une base de données.
- Mettre en place une base de données.
- Développer des composants dans le langage d'une base de données.
- 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.

ORGANISATION & CAHIER DES CHARGES

Les utilisateurs du projet

Ce projet se décompose en deux parties, destinées chacune à un type d'utilisateurs différents : une partie application mobile et une partie interface web admin.

L'application mobile est destinée à toute personne souhaitant utiliser une application de chat et inscrite sur l'application. Les utilisateurs devront impérativement s'inscrire, et auront ensuite la possibilité d'échanger avec les autres utilisateurs de l'application.

L'interface web Admin est destinée aux administrateurs de l'application ARCO. Les utilisateurs ayant le rôle "admin" auront accès à un dashboard permettant de gérer les utilisateurs et les chatroom (CRUD).

Les fonctionnalités attendues

Application mobile

Page d'accueil :

Lorsque l'utilisateur lance l'application, s'il n'est pas authentifié, il arrive sur une page avec deux boutons, un pour se connecter et un pour s'inscrire. Si l'utilisateur est déjà authentifié (token d'authentification toujours actif) il arrivera directement sur une page listant les différentes chatroom.

Page “Channels”:

Une page qui répertorie toutes les chatrooms disponibles. Il est possible de cliquer sur chacune d'entre elles afin d'avoir accès aux messages de la chatroom.

Page “InChannel”:

Une page qui représente l'intérieur d'un channel, avec tous les messages, l'utilisateur qui a envoyé le message et la date où le message a été envoyé. D'ici, il est possible d'interagir avec les utilisateurs présents dans la room. Au moyen d'une text area l'utilisateur peut rédiger son message et l'envoyer.

Page “Profile”:

Une page où sont affichées les informations de l'utilisateur, sur cette même page il est également possible de modifier ses informations.

La navigation sur l'application se fait grâce à un footer (bottom-navigation-bar), où les pages sont représentées par une icône. La bottom-navigation est toujours affichée à partir du moment où l'utilisateur est authentifié.

Application Web

Contexte technique

L'application mobile est développée dans le but d'être disponible sur IOS et Android.

L'application web sera accessible sur tous les navigateurs.

CONCEPTION DU PROJET

Choix de développement

Choix des langages :



Pour la réalisation du projet, j'ai utilisé JavaScript pour le back-end et le front-end.



Et NodeJs comme environnement de développement pour exécuter du code JavaScript côté serveur.

J'ai fait ces choix car JavaScript est un langage dynamique et polyvalent qui peut être utilisé à la fois côté serveur et à la fois côté client et permet ainsi de créer une application avec un seul langage.

De plus JavaScript possède une très large communauté et ainsi beaucoup de librairies sont disponibles pour répondre aux besoins de l'application.

Dernièrement, JS est un langage de programmation directement interprété par le navigateur et ne nécessite pas d'installation sur la machine de l'utilisateur.

Framework Back-end:

express

ExpressJS est un framework populaire pour NodeJs, il apporte peu de surcouche et laisse ainsi une liberté d'action et de très bonne performances. Il n'impose pas d'architecture et s'adapte donc parfaitement aux besoins de l'application.

De plus, ExpressJS utilise un système de middleware qui permet de gérer facilement l'authentification par le biais de tokens par exemple. Dernièrement, étant très populaire, ExpressJS bénéficie d'une large communauté, il est donc très facile de se documenter ou bien trouver des solutions aux problèmes rencontrés.

Librairie front :



React JS est une librairie moderne (2016) développée par Facebook qui a beaucoup de succès. Son approche sous forme de composants réutilisables permet une large personnalisation sans surcharger le code. Il est plus facile à maintenir car chaque composant a ses propres propriétés. Aujourd'hui, REACT JS s'impose comme une des librairie JS

les plus utilisées aujourd'hui, il est donc très facile également de se documenter.

Nous utilisons ReactJS pour l'application web et React Native pour l'application mobile.

React Native est très similaire à React JS dans son fonctionnement (composants) et écriture et permet de développer des applications multi plateformes (IOS et Android).

Autres outils utilisés :

- IDE: IntelliJ.
- Postman pour tester les routes API.
- GitHub pour collaborer en équipe et le versioning de notre code.
- Trello pour l'organisation du projet et la gestion des tickets.
- Figma pour maquetter notre application.
- Expo pour émuler notre application sur téléphone.

ORGANISATION DU PROJET

Ce projet a été réalisé dans le cadre de notre formation sur notre temps de présence en centre (du 02/01/23 au 27/01/23).

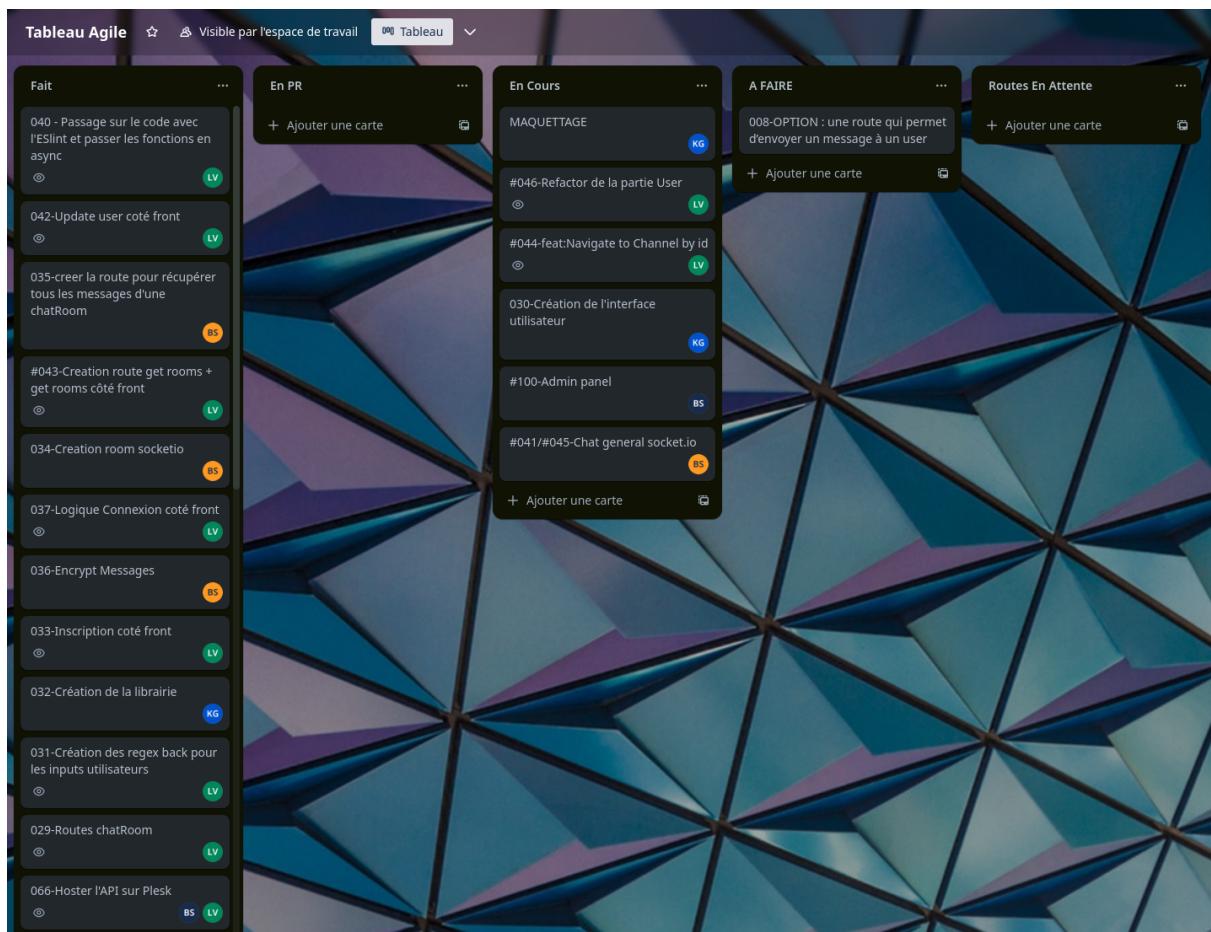
Ayant un timing serré il a fallu nous organiser et bien répartir les tâches. Pour ce faire nous avons premièrement listé toutes les features par ordre de priorités sur un fichier excel en classant chaque feature selon un MUST, SHOULD, COULD, WISH. Cela nous a permis de déterminer les versions de notre application.

Notre MUST, SHOULD, COULD, WISH pour ARCO :

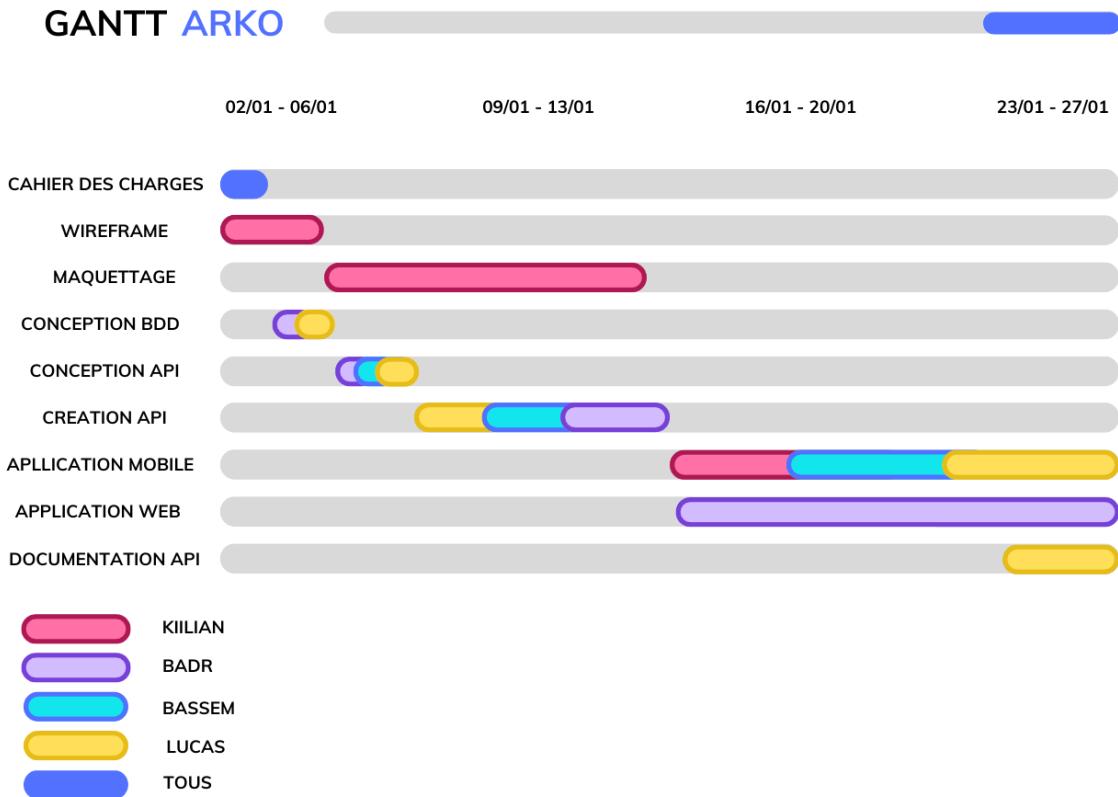
MUST (v1)				
1. Une inscription				
2. Une connexion				
3. Un salon de chat général				
4. Une page profile				
5. Un annuaire de tous les utilisateurs				
6. Une page qui détaille les informations d'un user				
7. Panel Admin (CRUD)				
8. Un utilisateur peut supprimer son message				
SHOULD (v2)				
1. Chat privé avec un utilisateur				
2. Barre de recherche pour utilisateurs				
3. Envoyer des photos/vidéos				
4. Authentification double facteur				
5. Un utilisateur peut modifier son message				
6. Si première ouverture de l'application, tutoriel de comment s'en servir sous forme de slide en se servant des frames Figma				
7. Dans l'update modifier son avatar				
COULD (v3)				
1. Créer des chatrooms avec plusieurs utilisateurs				
2. Ajouter un "Répondre" quand on reste appuyer sur un message				
WISH (v4)				
1. Connexion via Google, facebook...				
2. Envoyer un message vocal				

Ensuite nous avons établi une timeline à travers un “Diagramme de Gantt”. Cet outil nous a permis de planifier notre projet afin de respecter la première deadline du 27/01/23.

Nous avons ensuite rédigé et assigné les tickets grâce à Trello.



Nous avons également établi un diagramme de Gantt afin d'avoir une timeline à suivre :



Concernant GitHub, afin de collaborer au mieux nous avons définis les règles suivantes :

- Il est impossible de merger directement sa branche avec la branche "Master". Il faut obligatoirement créer une Pull Request, cette dernière doit être review et approuvée par minimum deux membres du groupe. La Pull Request met la demande de merge en attente, les utilisateurs peuvent parcourir le code ajouté par le développeur et soit approuver soit ajouter un commentaire sur un bout de code afin de donner des conseils ou demander des explications.
- Nous avons définis une norme de nommage pour nos branches et commits:
 - > Branches : elles commencent par # suivi du numéro de la tâche (exemple : #042).

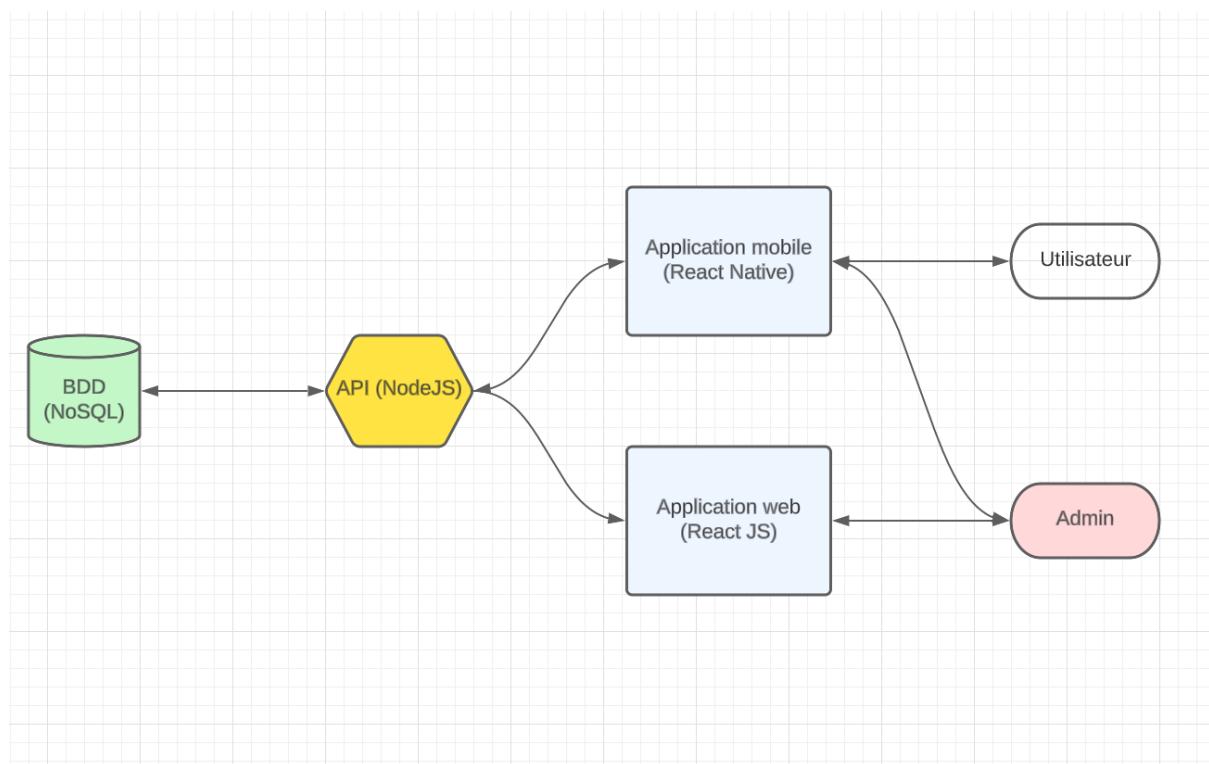
-> Commits : <numéro de tâche>-<nature de la tâche>:<descriptif du commit> (exemple: 042-feat:add-get-user-route)

Celà nous a permis d'avoir une unité dans notre façon de travailler.

Architecture des applications :

Les deux applications (web et mobile) s'appuient toutes deux sur la même API reliée à la base de données.

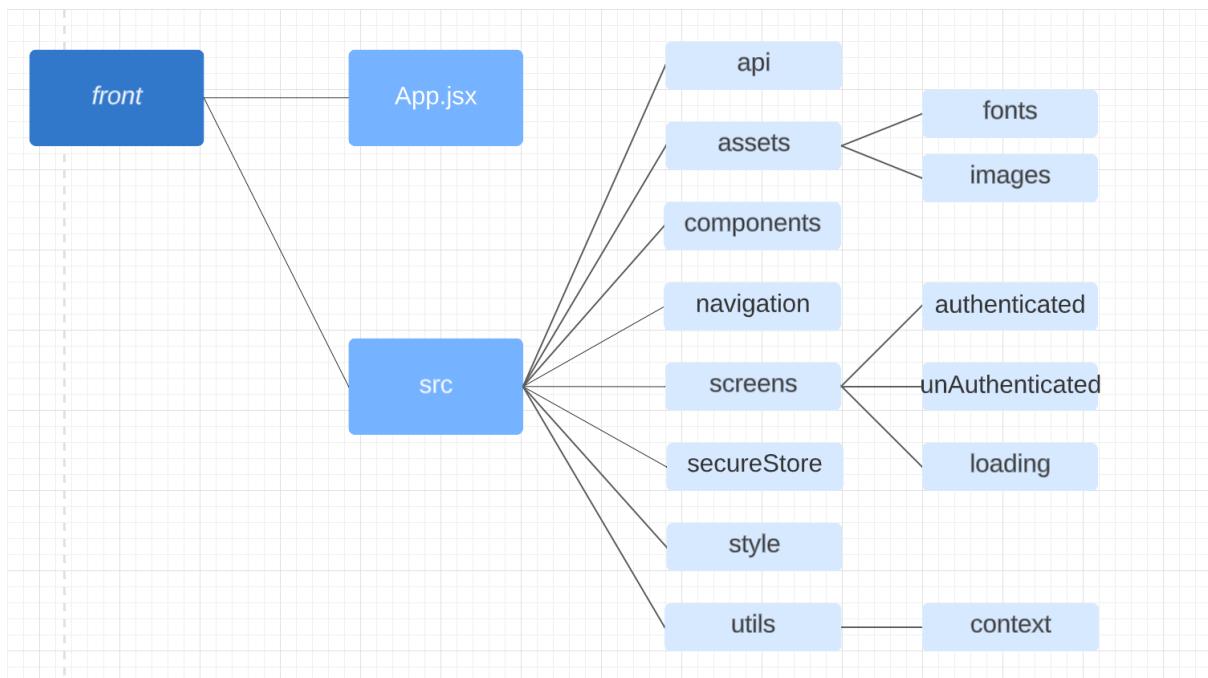
Tous les utilisateurs auront accès à l'application mobile, mais seuls les utilisateurs avec le rôle admin auront accès à l'application web.



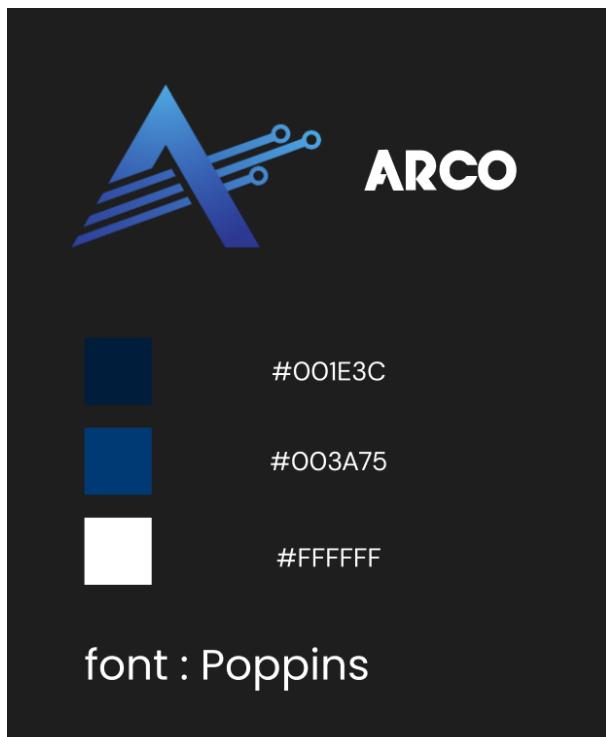
CONCEPTION DU FRONT-END DE L'APPLICATION

Pour créer les maquettes nous avons utilisé Figma, de par sa gratuité, sa simplicité d'utilisation et son aspect collaboratif. Il y est effectivement facile d'y travailler en équipe, car il est possible de travailler à plusieurs en même temps sur le même projet.

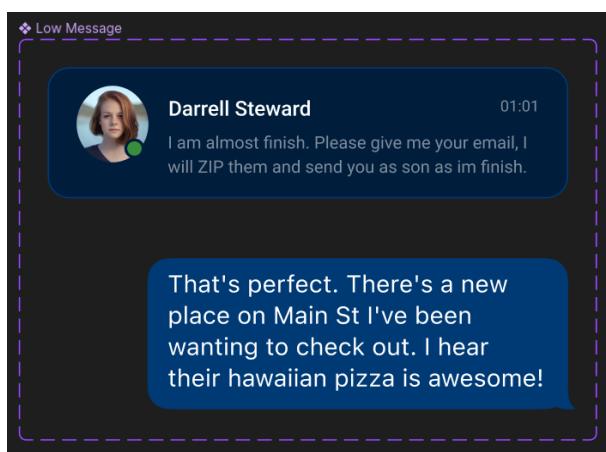
Arborescence du projet :



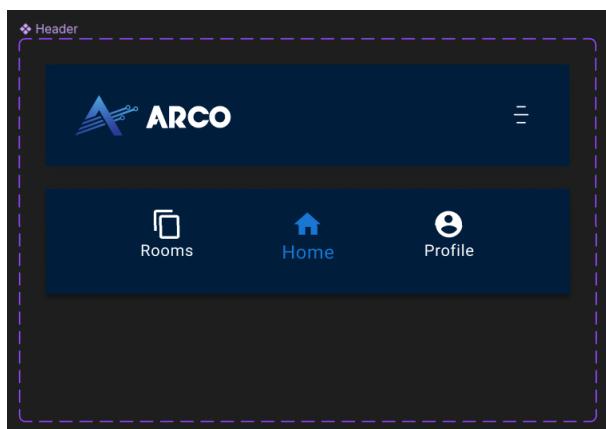
Charte graphique :



Nous avons commencé par établir une charte graphique avec logo, un code couleur ainsi qu'une police d'écriture pour notre application.



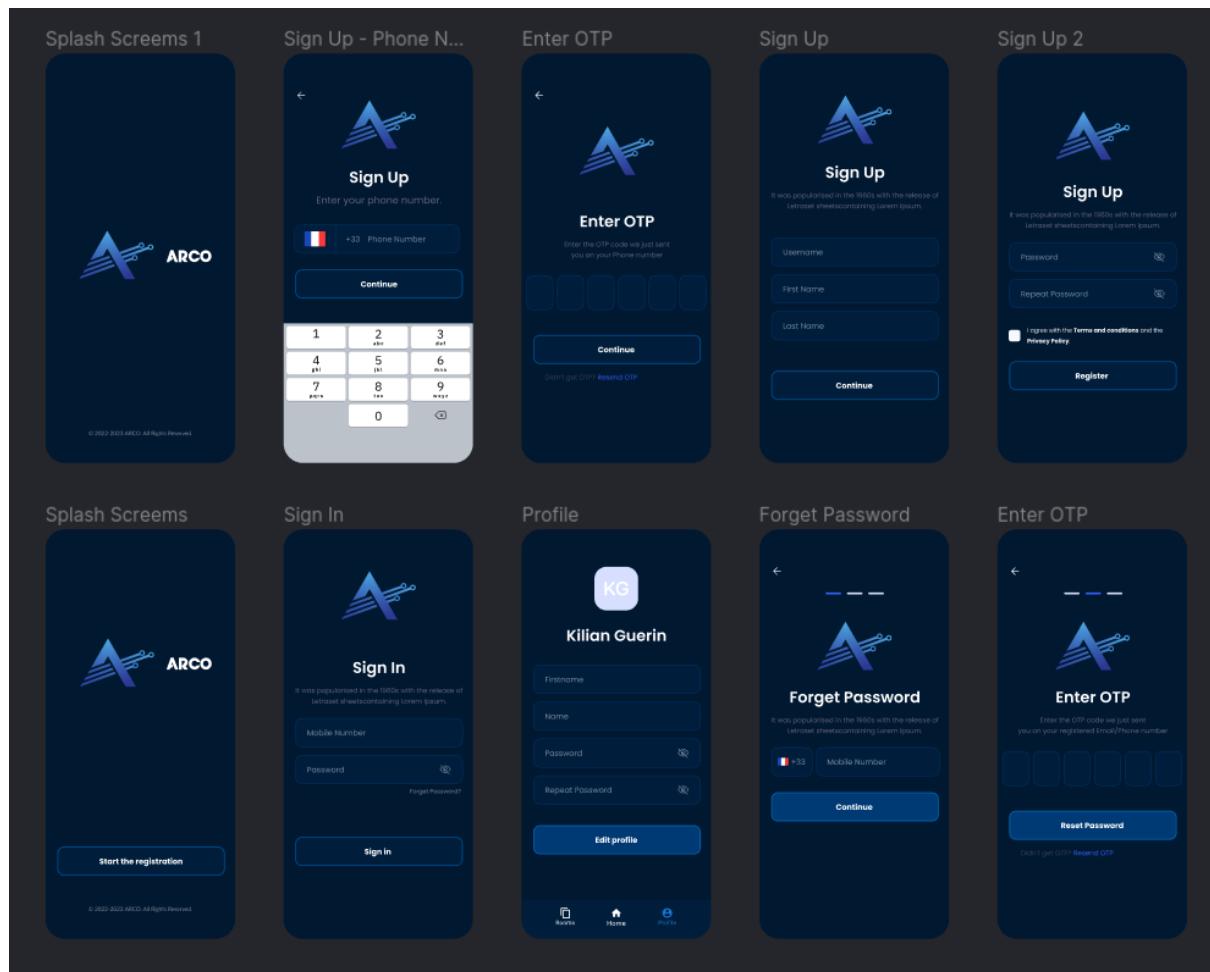
Une fois la charte graphique établie, nous avons commencé à créer des composants Figma réutilisables afin de faciliter la création des futures maquettes.



Ici par exemple la bottom-navigation que l'on retrouve dans quasi l'intégralité des écrans.

Maquettage de l'application mobile :

Pour maquette l'application nous avons utilisé Figma, et nous sommes servis des composants précédemment créés.



CONCEPTION DU BACK-END DE L'APPLICATION

Modélisation de la base de données

Nous avons fait le choix de travailler en s'appuyant sur une base de données NoSQL (Not only SQL).

Avantages d'une base de données NoSQL pour une application mobile de chat:

- Modèle de données flexibles : les données peuvent être facilement ajoutées et/ou supprimées sans impacter la structure de la base de données.
- Facilement scalable: permet de gérer un très grand flux de données.
- Disponibilité de la données. Le NoSQL permet d'avoir de très bonnes performances sur la vitesse d'accès aux données.
- Il est aussi possible de travailler avec des données structurées (Not ONLY sql).

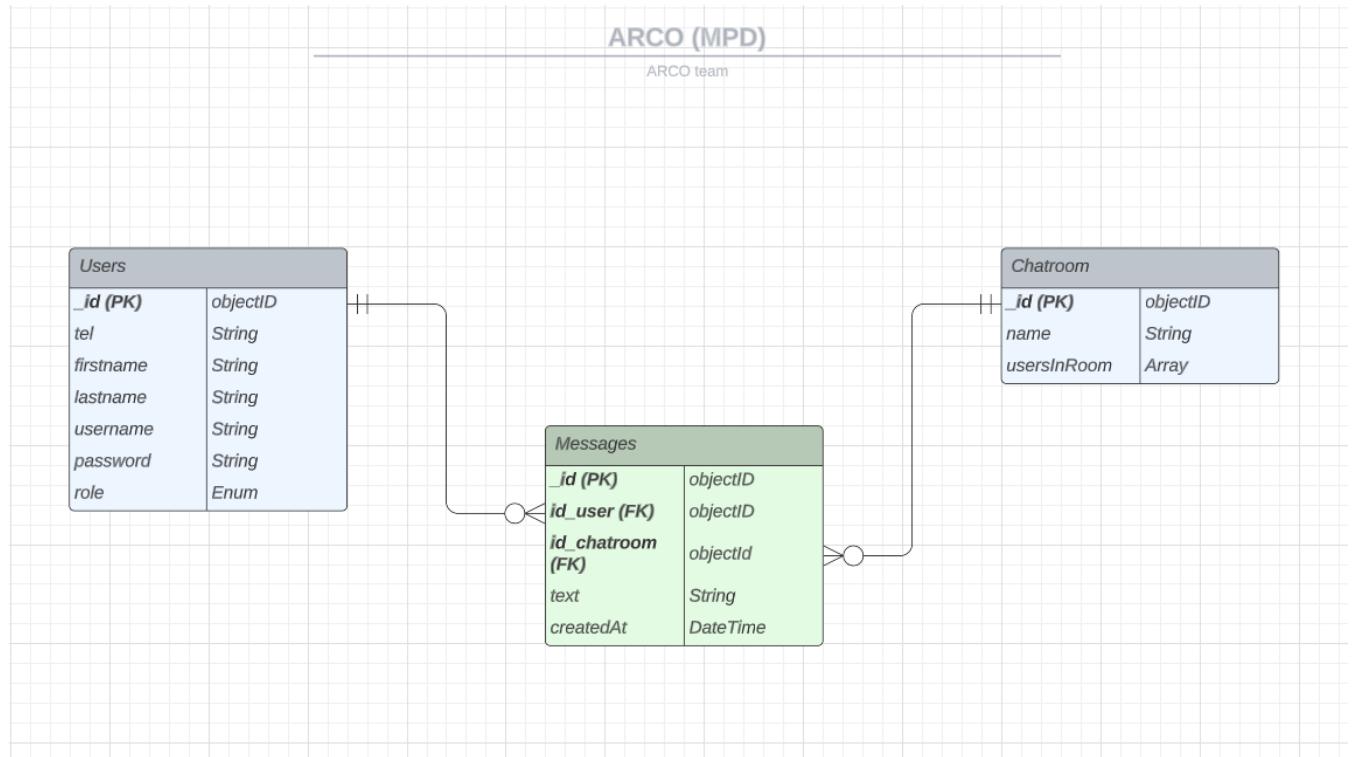
Pour ce faire nous avons choisi MONGO DB comme **système de gestion de base de données**.

Un autre avantage de MongoDB est le fait que la base de données est directement hébergée sur le Cloud de MongoDB. Ce qui facilite le travail d'équipe pendant le développement du projet.

Lors de la conception de la base de données, nous avons créé 3 collections :

- User
- Message
- Chatroom

Représentation des collections et relations entre elles :



La collection “Messages” joue le rôle de table intermédiaire. A savoir qu'elle contient obligatoirement un ID USER et un ID CHATROOM (clefs étrangères). Ce qui permet sur mes routes API de pouvoir chercher un message par son auteur, mais aussi savoir dans quel chatroom il se trouve.

Les clés primaires sont annotées PK (primary key) et les clés étrangères FK (foreign key).

Représentation d'un objet de la table “message” :

Création d'une API

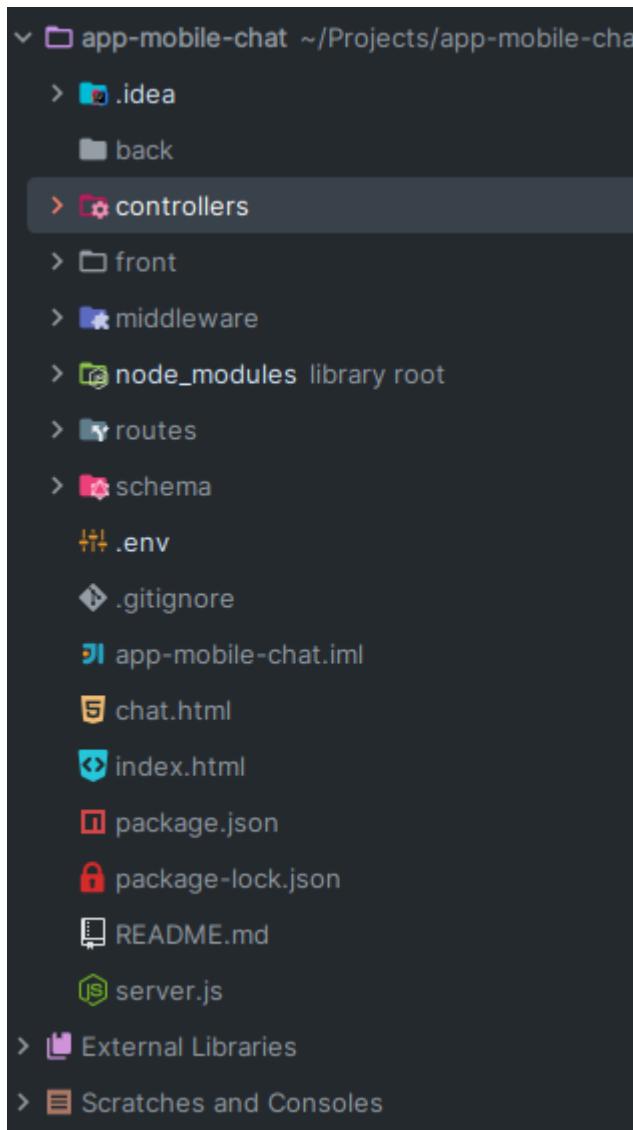
Conception et développement back-end de l'application (API)

Pour faire le lien entre notre front et notre base de données nous avons décidé de concevoir puis développer une API (Application Programming Interface).

Développer une API à part permet d'extraire et séparer la logique du front. De ce fait le front, au moyen de requêtes API, peut envoyer et recevoir des données déjà formatées.

Le back-end étant séparé du front cela améliore la clarté et l'accessibilité aux données.

Arborescence :



Le back-end est composé des dossier suivants :

- Les controllers : ils gèrent la logique et ainsi ce que doivent retourner les routes.
- Les routes : toutes les routes qui pourront être interrogées par le

front (CRUD).

- Middleware : c'est un composant intermédiaire dans notre requête, il permet d'effectuer des vérifications liés à une requête (vérification de l'intégrité des Tokens par exemple).
- Schemas : contient les fichiers qui permettent de générer et configurer les collections en base de données.

Fonctionnement de l'API :

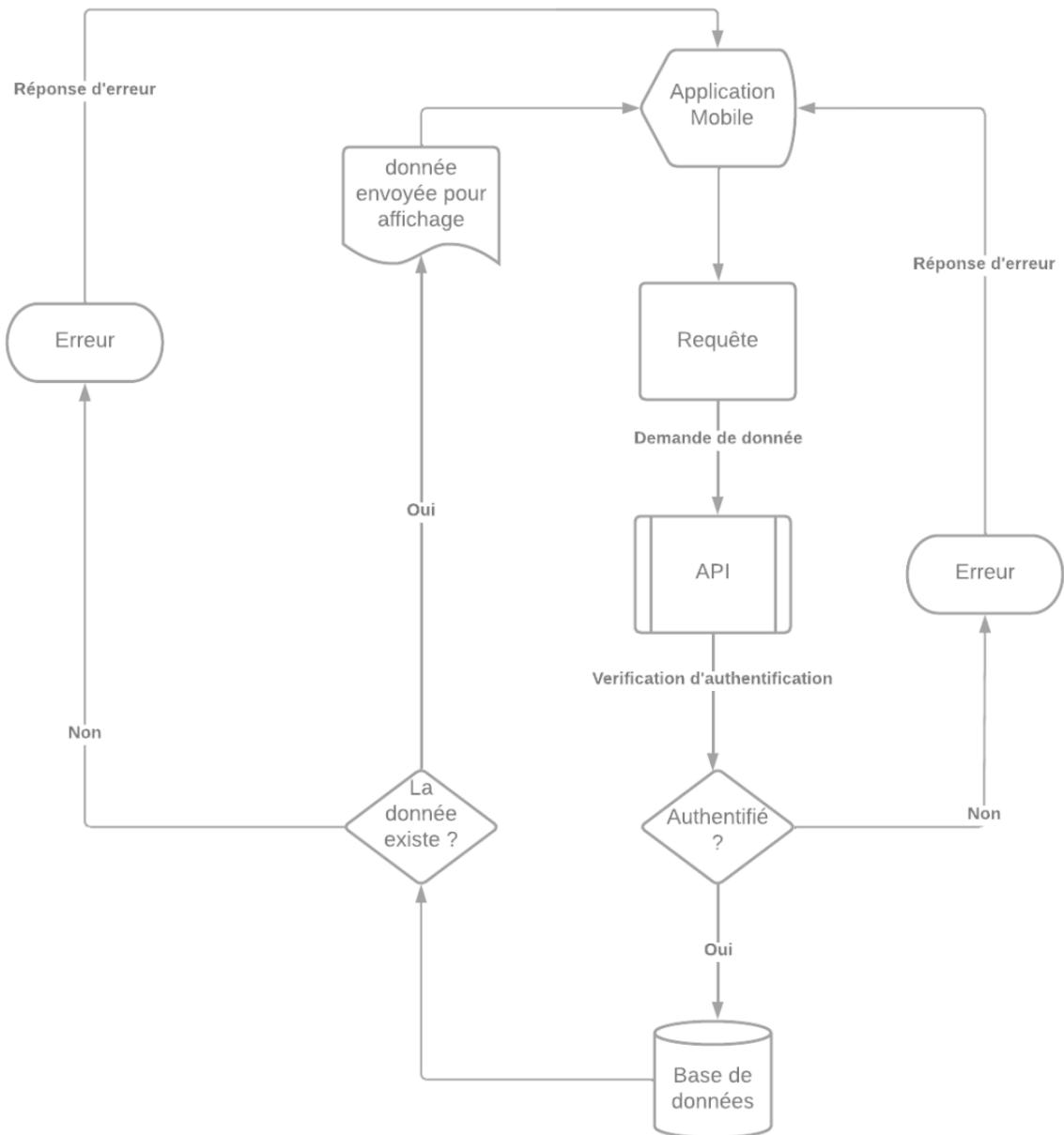
Le côté client interroge une route API au moyen d'une requête API qui peut avoir différents types selon ses besoins :

- GET : permet de recevoir de la donnée.
- POST: permet d'envoyer des données.
- DELETE : permet de supprimer des données.
- PUT : update total d'une entité.
- PATCH : update partiel d'une entité.

Chaque route est définie par une URI (extension de l'URL) qui permet d'interroger le controller qui y est associé. Cette fonction callback controller interagit avec la base de données et retourne un objet JSON et le code statut adéquat.

Avant d'interroger une fonction controller un middleware peut être interrogé comme composant intermédiaire, c'est souvent le cas pour vérifier si un utilisateur est bien authentifié.

Schéma de l'API :



Notre API est une **API REST**, c'est-à-dire qu'elle se plie à des standards. Le format en entrée et sortie de endpoint est uniformisé ainsi qu'un format d'erreur standardisé avec un code de retour http correspondant.

La documentation de l'API est donc facilement accessible et lisible pour les utilisateurs car ce sont toujours les mêmes standards.

Code	Définition
200 : OK	<i>Indique que la requête a réussi. Cela signifie que la</i>

Code	Définition
	<i>la demande du client a été traitée avec succès par le serveur, et la réponse renvoyée contient les informations demandées.</i>
201 : CREATED	<i>Indique que la requête a réussi et qu'une ressource a été créée. Ce code est généralement renvoyé après une opération de création réussie, indiquant que la ressource demandée a été créée avec succès sur le serveur.</i>
204 : NO CONTENT	<i>Indique que la requête a réussi, mais qu'il n'y a pas de contenu à renvoyer dans la réponse. Contrairement aux autres codes de statut, le code 204 est généralement utilisé lorsque le serveur a traité avec succès la demande du client, mais ne renvoie aucune donnée supplémentaire dans la réponse (ex : pour le DELETE).</i>
400 : BAD REQUEST	<i>Indique que le serveur ne peut pas comprendre la requête en raison d'une mauvaise syntaxe ou d'une structure incorrecte. Cela peut se produire si la requête est mal formée, si des paramètres sont manquants ou si les données envoyées ne sont pas valides selon les attentes du serveur.</i>
401 : UNAUTHORIZED	<i>Indique que la requête n'a pas été effectuée car des informations d'authentification sont manquantes ou invalides. Ce code est généralement renvoyé lorsque l'accès à la ressource demandée nécessite une authentification, mais les informations fournies ne sont pas valides ou sont absentes.</i>
403 : FORBIDDEN	<i>Indique que le serveur a compris la requête, mais ne l'autorise pas. Cela peut être dû à des raisons telles que des permissions insuffisantes pour accéder à la ressource demandée, une tentative d'accès à une ressource protégée sans les autorisations</i>

Code	Définition
	<i>appropriées, ou une politique de sécurité en place qui interdit l'accès.</i>
404 : NOT FOUND	<i>Indique que le serveur n'a pas trouvé la ressource demandée. Cela peut se produire si l'URL ou le chemin d'accès spécifié dans la requête ne correspond à aucune ressource existante sur le serveur.</i>
409 : CONFLICT	<i>Indique un conflit dans la requête. Ce code de statut est généralement utilisé lorsqu'il y a un conflit avec l'état actuel du serveur, c'est-à-dire lorsque la requête ne peut pas être traitée en raison d'un conflit avec une ressource existante. Dans le cas spécifique de l'adresse e-mail déjà utilisée, le code 409 peut être approprié pour signaler ce conflit.</i>
500 : INTERNAL SERVER ERROR	<i>Indique que le serveur a rencontré un problème interne lors du traitement de la requête. Ce code est utilisé lorsque le serveur rencontre une situation inattendue qui l'empêche de répondre correctement à la demande du client. Il peut s'agir d'erreurs de configuration, de bogues logiciels ou d'autres problèmes internes du serveur.</i>

Fonctionnement des Controllers :

Nous avons donc un fichier JS controller par collection en base de données (et un controller qui gère les erreurs) :

- chatRoomController.js
- messageController.js
- userController.js

Dans ces fichiers composés de fonctions, nous appelons et envoyons nos données à la base de données grâce aux méthodes fournies par Mongo.

Exemple de fonction au sein du controller “chatRoomController” :

```
1 usage  ↵ lucas-verdier
const createChatRoom = async (req, res) :Promise<void>  => {
  const { name, usersInRoom } = req.body;
  try {
    if (checkRole(req.auth.role)) {
      const chatRoom :HydratedDocument = new ChatRoom( doc: { name, usersInRoom });
      await chatRoom.save();
      res.status(201).json({ chatRoom });
    }
  } catch (err) {
    res.status(401).json({ err });
  }
};
```

```
1 usage  ↵ lucas-verdier
const checkRole = (role) :boolean  => {
  if (role !== 'admin') {
    throw new Error('Vous ne possédez pas les droits pour effectuer cette action.');
  }
  return true;
}
```

On voit donc ici une fonction asynchrone qui permet à un admin de créer une nouvelle chatRoom.

La donnée est envoyée dans le body (corps de la requête) sous format JSON.

Quand la route est interrogée, la fonction “checkRole” si l’utilisateur à le rôle “admin”. Si c’est le cas les informations du body sont “save” (méthode de mongo) sont sauvegardées en base de données. Si

l'utilisateur n'est pas admin une erreur est renvoyée au front afin qu'il puisse gérer l'erreur.

L'objet req est généré par la requête HTTP et représente la requête alors que l'objet res, également généré par la requête, représente la réponse à retourner.

Fonctionnement des routes :

Dans ce dossier sont présents les fichiers qui gèrent les routes API (URI). Pour chaque fichier controller, il y a un fichier route. Dans ce fichier on déclare donc le type de requête HTTP (PUT, GET, POST, DELETE), l'URI qu'il faut interroger depuis la front, la fonction que l'uri prend en callback et le middleware qui doit être interrogé.

Exemple du fichier chatRoom.js issu du dossier route :

```
router.post( path: '/create', auth, createChatRoom);
router.put( path: '/insert-user', auth, addUserToChatRoom);
router.put( path: '/remove-user', auth, removeUserFromChatRoom);
router.get( path: '/rooms', getAllRooms);

module.exports = router;
```

Pour reprendre l'exemple de la création d'une chatRoom (première route), on voit ici que c'est une route de type POST, qui peut être interrogée au moyen de l'URI “/create”, qui passe par le middleware “auth” qui check la validité des tokens de l'utilisateur, et a pour retour la fonction createChatRoom du controller chatRoom.

Fonctionnement des middleware :

Le middleware sous ExpressJS est une fonction créée qui agit comme composant intermédiaire entre la requête et le callback. Le middleware permet d'effectuer des tâches avant que la requête passe à la fonction callback.

Par exemple pour vérifier si un utilisateur est authentifié et si ses tokens sont valides.

Ci-dessous, toujours pour l'exemple de la création d'une chatRoom, nous avons besoin de savoir si l'utilisateur est connecté (en mode admin qui plus est). Quand l'utilisateur interroge la route, la requête passe d'abord par le middleware 'auth', ce dernier vérifie l'intégrité du token d'authentification et du refresh token.

Vérification du token d'authentification :

```
jwt.verify(authHeader, ACCESS_TOKEN_SECRET, options: (err, tokenData) : any | undefined => {
  if (tokenData === undefined) {
    console.log("Token access :");
    return res.sendStatus(403);
}
```

Vérification du refresh token :

```
jwt.verify(
  refreshtoken,
  REFRESH_TOKEN_SECRET,
  options: (err, refreshTokenData) : any | undefined => {
    if (refreshTokenData === undefined) {
      user.findById(userId).then(onfulfilled: (user : module:mongoose.Schema<any, Model<any>>) => {
        const refreshtoken = jwt.sign(
          payload: {userId: user._id, role: user.role},
          REFRESH_TOKEN_SECRET,
          options: {expiresIn: "30d"})
        );
        return res.status(417).json({refreshtoken});
      });
    } else if (userId === refreshTokenData.userId) {
      console.log("Token refresh :");
      return res.sendStatus(403);
    } else {
      req.auth = refreshTokenData;
      next();
    }
  }
)
```

Le token JWT (JSON WEB TOKEN) d'authentification contient des informations permettant d'identifier un utilisateur (son id, son rôle), c'est en quelque sorte la carte de visite de l'utilisateur sur l'application.

Le refresh token, lui, est régénéré tous les N temps. Il permet tout ces temps de vérifier l'intégrité de l'utilisateur et ses informations. Celà permet d'ajouter une couche de sécurité, car si jamais un utilisateur est corrompu et modifie son rôle par exemple, à l'expiration du refresh token, les informations seront vérifiées. Si elles ne correspondent pas, l'utilisateur ne sera plus authentifié.

Connexion à la base de données :

Nous nous connectons à une base de données NoSQL MongoDB, l'avantage est qu'elle est directement hébergée sur le Cloud ce qui fluidifie et facilite le travail en équipe.

Connexion :

```
function connection(url) : void
{
    mongoose.set("strictQuery", false);
    mongoose.connect(
        url,
        { useNewUrlParser: true, useUnifiedTopology: true }
    )
        .then(() => console.log("Connexion à MongoDB réussie !"))
        .catch(() => console.log("Connexion à MongoDB échouée !"));
}
```

Nous nous servons de MongoDB Compass comme SGBD, il permet de gérer et visualiser nos données au seins de collections.

Collection "Messages" :

_id	text	user_id	chatroom_id	iv	createdAt
ObjectID('645f56049bf0bcf41e8233ae')	"18dd3528b268ea2c990cad4e02f5ab"	"63b5940d8496097da3460a1e"	"63b6df42243047db8d7aeef2a"	"x5oA06EBWCipq/H+Oulz5g=="	2023-05-13T09:19:00.362+00:00
ObjectID('645f56279bf0bcf41e8233b1')	"7014ddb0cf3c63e6adef0f07034ad0b15"	"63b5940d8496097da3460a1e"	"63b6df42243047db8d7aeef2a"	"trjfS3E2ZLkM0tcoKRQ=="	2023-05-13T09:19:35.246+00:00
ObjectID('645f56319bf0bcf41e8233b4')	"48657555ec95ccb1b3870f944d7518c"	"63b5940d8496097da3460a1e"	"63b6df42243047db8d7aeef2a"	"h/vCP556QvN1Pa4EUniqIA=="	2023-05-13T09:19:45.516+00:00

On voit ici que l'on a accès sur le côté à toutes nos collections, ces dernières contiennent des documents qu'il est possible de consulter. Il est également possible d'ajouter des documents manuellement ou bien encore de les supprimer.

Pour interagir avec cette base de données MongoDB nous utilisons les méthodes de requête de Mongo et mongoose, telles que `find()`, `findOne()`, `findById()`, `deleteOne()` etc...

Il est souvent possible d'ajouter des critères à ces méthodes afin d'affiner la requête.

Les requêtes en base de données sont initialisées et appelées lorsque la route API correspondante est interrogée.

Cycle d'une requête utilisateur :

1. Requête utilisateur sur une route API
2. La route API exécute la fonction callback qui lui est associée.
3. La fonction callback qui contient la requête reçoit en params ou en body les informations nécessaires à la requête.

4. La requête en base de données est exécutée. Si elle réussit, la donnée est ajoutée, supprimée, modifiée selon la nature de la requête. Si elle échoue, une erreur est renvoyée en retour de la fonction callback.
5. La fonction callback associée à la route API interrogée renvoie soit un message de succès soit un message d'erreur.

Exemple de requête vers la base de données :

```
router.get( path: '/details/:id', auth, getUserInfos);
```

```
const getUserInfos = (req, res) : void => {
  User.findById(req.params.id) Query...
    .then( onfulfilled: user : ... => {
      res.status(200).json({ status: "ok", message: "Infos de l'utilisateur", data: {user} });
    } ) Promise<T>
    .catch(error => res.status(500).json({ error }));
};
```

Ici on peut observer que pour requêter et obtenir les informations d'un utilisateur en particulier, le côté client doit interroger l'URI "/details/:id". Ou ":id" est un params qui représente l'ID de l'utilisateur.

Une fois l'URI interrogée, le middleware effectue ses tâches d'authentification, ensuite est interrogée la fonction callback associée, ici "getUserInfos".

"getUserInfos" reçoit alors req (qui peut contenir des informations utiles en params ou bien encore en body) et retournera res.

Dans la méthode, User représente la collection interrogée. Ensuite c'est la méthode de mongoose findById qui est appelée, cette dernière prend en paramètre l'id de l'utilisateur. Dans notre cas, il est récupéré par l'information params (:id).

Ensuite, si la requête réussit (si un utilisateur est trouvé) on renvoie un statut 200 avec les informations de l'utilisateur qui seront exploitables par le front. Si la requête échoue, on renvoie une erreur serveur 500 avec l'objet error détaillant les raisons de cette erreur, qui servira à renvoyer le message adéquat à l'utilisateur.

Test des routes API avec Postman :

Lors du développement de notre API nous avons utilisé POSTMAN pour tester nos routes.

Nous avons travaillé sur un postman en ligne commun afin d'avoir tous la même version (WORKSPACE).

Nous avons donc créé dans notre Workspace toutes les routes de notre API dans le but de toutes les tester. Chaque route est identifiée par une méthode de requête HTTP et un nom de route. Pour chaque folder du workspace il est possible de préciser le type d'Authorization pour le dossier (Token d'authentification etc...) ainsi que des variables qui pourront être utilisées par toutes les routes. Ces routes peuvent hériter des paramètres de leur parent afin de ne pas avoir à configurer les authorization à chaque fois par exemple.

Notre workspace :

The screenshot shows a Postman workspace titled "ARCO". The left sidebar displays a hierarchical structure of API routes:

- Users Routes**:
 - GET ALL USERS
 - POST INSERT USER
 - POST CONNECT USER
 - GET GET USER INFOS
 - PUT UPDATE USER FROM USER
- Message Routes**:
 - POST SEND MESSAGE
 - GET DECRYPT ONE MESSAGE
 - GET Get ALL MESSAGES DECRY...
 - GET GET MESSAGES BY ROOM ID
- Admin**:
 - PUT UPDATE USER FROM ADMIN
 - DEL Delete Message
 - DEL DELETE USER
 - POST CREATE CHATROOM
 - PUT INSERT ONE USER IN CHAT
 - PUT REMOVE USER FROM CHAT...
- Rooms**:
 - GET GET ALL ROOMS

The right panel shows the "Authorization" tab selected. It includes fields for "Type" (set to "Bearer Token") and a "Token" placeholder {{Token}}. A note at the top states: "This authorization method will be used for every request in this collection. You can override this by specifying ...". A tooltip message provides a warning about sensitive data and recommends using variables.

Chaque requête est donc caractérisée par une méthode de requête HTTP (POST dans l'exemple ci-dessous), ainsi qu'une URI à interroger. Ci-dessous on peut voir la route hérite de la variable {{uri}} qui contient la racine de l'URI afin de ne pas avoir à la réécrire à chaque fois.

Dans l'exemple ci-dessous, on cherche à connecter un utilisateur. Pour ce faire on envoie dans le body l'username et le password. Plus bas on pourra observer la réponse reçue par le serveur (notre API).

Requête :

The screenshot shows the Postman interface with a POST request configuration. The method is set to POST and the URL is {{uri}}/user/login. The 'Body' tab is selected, showing a JSON payload:

```
1 {  
2   "username": "admin",  
3   "password": "admin"  
4 }
```

On observe donc ci-dessous la réponse à notre requête, ici c'est un statut 200 (donc succès), et sont alors renvoyées les informations attendues, en l'occurrence un ID user, un access Token ainsi qu'un refresh token.

Il est possible de formater notre réponse comme on le souhaite. Dans l'exemple ci-dessous c'est le format JSON qui est choisi.

Réponse:

Status: 200 OK Time: 387 ms Size: 1.43 KB | Save Response ▾

Body Cookies Headers (23) Test Results

Pretty Raw Preview Visualize JSON ↻

```
1 {  
2     "userId": "6460d4317437c362a7652bc0",  
3     "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2NDYwZDQzMTC1  
4     "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2NDYwZDQzMTC1  
5         KGTG0gjSLXwkrDN5DJV-1-lWEkh-fGn5LWMtDbTMmT0"  
}
```

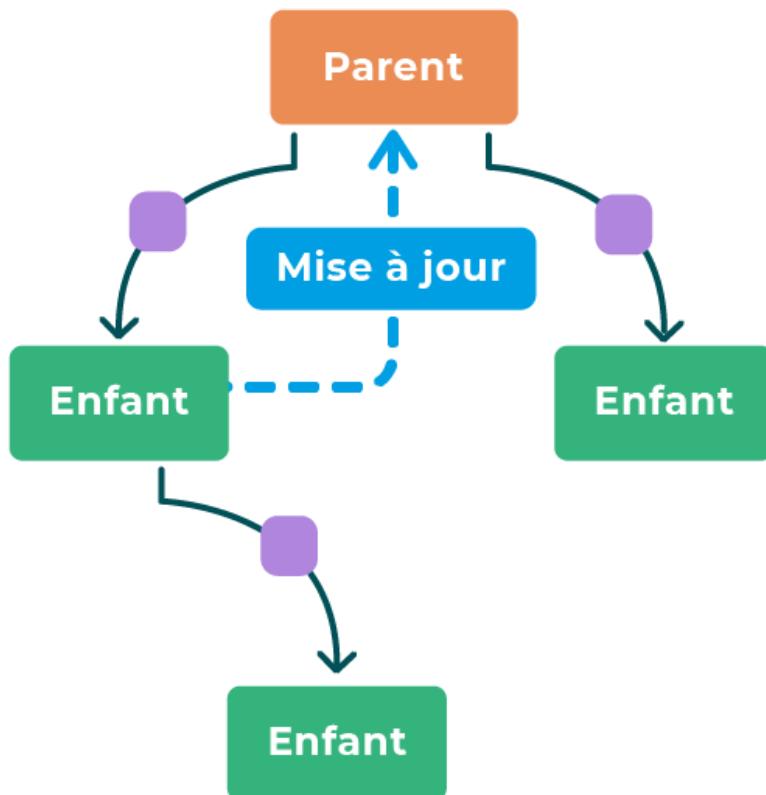
CONCEPTION FRONT-END DE L'APPLICATION

Mise en place de composants React Native

React Native est une librairie Java qui permet la création de composants pour les applications mobiles.

Un composant à vocation à être réutilisable afin d'alléger un maximum la codebase.

Les composants React sont basés sur des relations parents/enfants, en effet un composant parent appelle un(des) composant(s) enfant(s) en lui passant de la données par le biais de "props". Les props permettent de personnaliser un composant enfant selon l'utilisation voulue en utilisant le même composant.



Dans le cadre de notre application la réutilisation des composants à été primordiale (vignettes de channels, bulles de texte dans le chat, boutons, inputs...). Sans ces composants le code aurait été beaucoup plus lourd.

L'avantage des composants React est qu'il est possible d'associer de la logique à un petit éléments, et donc avoir notre logique et notre rendu (JSX) dans la même page.

Exemple de composant qui affiche un channel :

```
export default function ChannelScreen({ navigation }) {
  const [user, setUser] = useContext(UserContext);
  const [state :{screenHeight:number} , setState] = useState( initialState: {
    screenHeight: 0
  });
  const [rooms :{} , setRooms] = useState( initialState: {
    roomId: null,
    name: '',
    usersInRoom: 0,
  });

  1 usage  ↵ Kilian
  const onContentSizeChange = (contentWidth, contentHeight) :void  => {
    setState( value: { screenHeight: contentHeight });
  };

  1 usage  new *
  const getRooms = async () :Promise<void>  => {
    const response :AxiosResponse<any>  = await axios.get(
      url: 'https://lucas-verdier.students-laplateforme.io/api/chat-room/rooms',
    )
    console.log(response);
    setRooms( value: {
      roomId: response.data.data.room[0]._id,
      name: response.data.data.room[0].name,
      usersInRoom: response.data.data.room[0].usersInRoom.length,
    });
  }

  useEffect( effect: () :void  => {
    getRooms();
  }, deps: []);
}
```

Ici notre composant “ChannelScreen” a pour vocation d'afficher les channels disponibles. Dans notre v1, un seul channel général est prévu.

Le composant reçoit en props la fonction navigation, ce qui permet lors de certain event d'appeler cette fonction et ainsi naviguer entre les différents screens de l'application.

Des states sont initialisés, relatifs aux informations d'un channel. Les states sont initialisées et modifiés grâce au Hook useState() importé plus haut dans le code.

Définition d'un state : un state (état) correspond à une valeur qui est initialisée au moment où le state est déclaré, un peu à la manière d'une variable. Ce dernier peut être mis à jour au moyen de la méthode setState(). Un state peut être de différent type (string, int, array, object etc....).

Ce qui différencie un state d'une variable c'est sa persistance.

En effet, un composant a un cycle de vie, à chaque fois qu'un state est mis à jour le composant va re-render (se rafraîchir) et se réinitialiser avec les valeurs initiales, SAUF les states eux, dont les valeurs persistent.

Si une variable elle est déclarée à null, puis mise à jour à "Hello", lors du re-render du composant, elle sera de nouveau égale à null.

Définition d'un Hook : les hooks sont des fonctions composants qui permettent d'effectuer différentes tâches. Certains sont apportés directement par la librairie React, mais il est possible d'en créer soi-même. Ils sont reconnaissables par leur nom car ils commencent par `useLeNomDeMonHook.jsx`.

Les plus connus sont le useState() qui permet d'initialiser un état et de mettre à jour un état. Un autre très connu est le useEffect() qui permet de programmer des actions à effectuer à un moment donné du cycle du composant.

Dans l'exemple ci-dessous nous utilisons useState() pour gérer les valeurs de nos channels, ainsi que le useEffect() pour ordonner au composant d'effectuer la requête API get channels au l'instanciation du composant.

Pour indiquer à un useEffect() de faire une action une seule fois à l'instanciation d'un composant il faut lui passer un tableau de dépendances vide.

Ici il appelle donc la fonction asynchrone getRooms(), cette dernière, fait un call API et grâce à sa réponse on peut mettre à jour nos state grâce à la méthode setState();

Utilisation du hook useState() et useEffect() :

```
usage - new
const getRooms = async () : Promise<void> => {
  const response : AxiosResponse<any> = await axios.get(
    url: 'https://lucas-verdier.students-laplateforme.io/api/chat-room/rooms',
  )
  console.log(response);
  setRooms( value: {
    roomId: response.data.data.room[0]._id,
    name: response.data.data.room[0].name,
    usersInRoom: response.data.data.room[0].usersInRoom.length,
  });
}

useEffect( effect: () : void => {
  getRooms();
}, [ deps: [] ]);
```

Les states n'ont pas uniquement vocation à être initialisés et mis à jour, leur finalité est de pouvoir faire du rendu dynamique selon leur valeur. Dans l'exemple ci-dessous, je me sert de la valeur de mes states pour configurer mon rendu. Le composant enfant <Room /> en plus d'un type et navigation prend une props “info”. Le contenu de cette props va servir à personnaliser l'affichage du composant Room (son nom, ses utilisateurs).

Return de mon composant ChannelScreen :

```
return (
  <View style={styles.container}>
    <ScrollView
      scrollEnabled={scrollEnabled}
      onContentSizeChange={onContentSizeChange}
      style={styles.container_input}>
      <Room type='group' navigation={{navigation, roomId}} infos={{{
        'name': rooms.name,
        'members': rooms.usersInRoom,
        'lastMessage': 'Hello World',
        'lastMessageTime': '12:00',
      }}>
    </ScrollView>
  </View>
);
```

Ci dessous des extraits du composant <Room />, on peu voir qu'il reçoit les propriétés du parent (navigation, type, roomId, et l'objet infos). Ces valeurs servent ensuite à afficher des informations personnalisées dans le rendu JSX de la Room.

Composant enfant <Room /> appelé par ChannelScreen :

```
export default function CardRooms({ navigation, type, roomId, infos :[] = [] }) {
  <Text h4 style={{ color: "#fff", fontFamily: "PoppinsBold" }}>{infos.name.match(/(\^\$|\$?|\b\$)?/g).join("")}.match</Text>
  <View style={{ marginLeft: 20 }}>
    <Text style={{ color: "#fff", fontFamily: "PoppinsBold", fontSize: 18 }}>{trimString(infos.name, length: 23)}</Text>
  </View>
}
```

Développement du dashboard Admin

Pour le dashboard admin nous avons développé une application web en React JS, qui s'appuie sur la même API et la même base de données que notre application mobile.

React JS est une librairie JavaScript qui permet la création de composants pour les applications web.

Elle est dans son fonctionnement très similaire à React Native (utilisation de hooks, de props, de composants réutilisables etc....).

Pour développer ce dashboard admin nous avons utilisé la librairie de composants MUI.

MUI a un important panel de composants facilement personnalisables ce qui facilite le développement et permet d'éviter de perdre trop de temps à écrire du CSS.

Ci-dessous le composant DataGrid de MUI-X qui permet d'afficher des tableaux très personnalisables. Ici, on s'en sert pour afficher la liste des utilisateurs.

La DataGrid prend un propriété des rows et des columns, valeurs initialisées dans le corps du composant.

Initialisation des “rows” :

```
useEffect(() => {
  fetch("http://localhost:8800/api/user/users")
    .then((res) => res.json())
    .then((res) => {
      setUsers(res.data.users);
    });
}, []);
```

On utilise le hook useEffect() pour fetch la liste des utilisateurs et mettre à jour le state users.

Initialisation des colonnes :

```
const columns = [
  {field: "_id", headerName: "ID", width: 70},
  { field: "lastname", headerName: "lastname" },
  {
    field: "firstname",
    headerName: "firstname",
    flex: 1,
    cellClassName: "name-column--cell",
  },
  {
    field: "actions",
    type: "actions",
    getActions: (getRowId) => [
      <GridActionsCellItem icon={<DeleteIcon />} onClick={() => Delete(getRowId.row._id)} label="Delete" />,
      <GridActionsCellItem icon={<EditIcon />} onClick={() => handleOpen(getRowId.row._id)} label="Edit" />,
    ],
  },
];
```

Ici on renseigne les noms que doivent avoir les colonnes.Ici sont aussi initialisés et configurés les actions possibles,en l'occurrence ici, supprimer ou modifier un utilisateur.

Initialisation de la DataGrid MUI :

```
<DataGrid
  checkboxSelection
  rows={users}
  columns={columns}
  getRowId={(users) => users._id + users.firstname + users.lastname}
/>
```

Visuel de la DataGrid affichant les utilisateurs de l'application :

The screenshot shows a dark-themed application interface. On the left, there is a sidebar with the title "ADMINIS" at the top. Below it is a circular profile picture of a man, followed by the text "Zadmin" and "Arko's Admin". The sidebar has three main sections: "Data" (with "Manage Users"), "Pages" (with "Profile Form"), and "Pages" again (with "Manage ChatRooms" and "Manage Messages"). The main content area is titled "Users" and subtitle "Managing the Users". It features a DataGrid with the following columns: a checkbox column, "ID", "lastname", and "firstname". The data grid contains six rows of user information:

ID	lastname	firstname
63b453...	Verder	Lucas
63b453...	Sataf	Jojo
63b458...	Zerin	Zilian
63b594...	Zaldersoon	Bradadmin
645ba8...	Lucas	Verdier
6460d4...	admin	admin

At the bottom right of the main content area, there are buttons for "Rows per page: 100" and "1–6 of 6".

This screenshot provides a closer look at the DataGrid from the previous image. The columns are labeled "checkbox", "ID", "lastname", and "firstname". The same six rows of data are visible:

ID	lastname	firstname
63b453...	Verder	Lucas
63b453...	Sataf	Jojo
63b458...	Zerin	Zilian
63b594...	Zaldersoon	Bradadmin
645ba8...	Lucas	Verdier
6460d4...	admin	admin

At the bottom right of the grid, there are buttons for "Rows per page: 100" and "1–6 of 6".

Search 🔍

Users Manage the Users

ID	lastname	firstname
63b453...	Verdier	Lucas
63b453...	Sataf	Jojo
63b458...	Zerin	Zilian
63b594...	Zaldersoon	Bradra
645ba8...	Lucas	Verdier
6460d4...	admin	admin

User Update

First Name

Last Name

telephone

Username

password

UPDATE USER

Rows per page: 100 ▼ 1–6 of 6 < >

La sécurité dans l'application

La sécurité et la conformité à la RGPD est primordiale, nous avons donc mis l'accent sur ces deux points.

En effet, ARCO est une messagerie où tous les messages sont cryptés et tous les mots de passes sont hashés.

Sécurité côté API (outils nodeJS)

Nous avons utilisés :

- **cors** : c'est un middleware nodeJS qui permet de contrôler l'accès aux données de l'API depuis différents domaines.
Il permet d'établir des règles et configurer les headers des requêtes http : avec qui il est possible d'envoyer/recevoir de la donnée.
Quel format on peut recevoir également.
- **helmet** : C'est un middleware NodeJs qui permet de protéger des failles XSS et CSRF en configurant les entêtes HTTP de manière appropriée.
- **bodyParser** : il protège des injections et attaques en analysant le corps des requêtes HTTP.
- Utilisation de **mongoose** et **mongoDB** pour la connexion à la base de données et leurs méthodes de requête. Ces outils incluent de base des fonctionnalités sécurisées.

Utiliser des outils de sécurité est important mais nous ne sommes pas reposés en intégralité dessus, car ces derniers peuvent présenter des failles, surtout si elles ne sont pas mises à jour régulièrement.

Nous avons donc tout de même vérifié les données envoyées par l'utilisateur par le biais de regex :

```
const checkUsernameFormat = (data) :boolean => {
  if (data.match('![@#$%^&*(),.?":{}|>\s]+')) {
    throw new Error('Username invalide, certains caractères spéciaux et/ou espaces ne sont pas supportés !');
  }
  return true;
}
```

Cela permet de vérifier qu'un utilisateur n'essaie pas d'échapper des caractères dans l'input.

Afin de respecter au maximum la RGPD et ainsi protéger les données de nos utilisateurs nous avons crypté les messages de nos utilisateurs grâce à une librairie nodeJS nommée "crypto".

Il faut au préalable définir dans le .env un algorithme de transformation de la donnée ainsi qu'une clef secrète.

En plus de l'algorithme de cryptage, on génère grâce à crypto une chaîne de caractère nommée IV. Cette dernière est une couche supplémentaire de cryptage. Le cryptage de la donnée est donc réalisé depuis l'algorithme ainsi que l'IV, ce qui sécurise encore plus notre donnée. Car si l'algorithme de cryptage est découvert, il y a encore l'IV et la clef secrète qui assurent l'intégrité de la donnée.

Grâce à ces deux éléments et la key secret il est alors possible d'encrypter le texte en hexadécimal.

Extrait de la méthode “sendMessage”, où le cryptage de la donnée est fait :

```
try {
  const { chatroom_id, text } = req.body;
  const user_id = req.auth.userId;

  // Chiffrement du texte
  const algorithm = process.env.ALGO_MSG;
  const key = process.env.KEY_MSG;
  const initIv :Buffer = crypto.randomBytes( size: 16 );
  const cipher :CipherGCM = crypto.createCipheriv(algorithm, key, initIv);
  let encryptedText :string = cipher.update(text, inputEncoding: "utf-8", outputEncoding: "hex");
  encryptedText += cipher.final( outputEncoding: "hex" );

  // Conversion de l'IV en base64
  const iv :string = initIv.toString( encoding: 'base64' );

  // Création de l'objet Message
  const message :HydratedDocument = new Message( doc: { text: encryptedText, user_id, chatroom_id, iv } );
}
```

Extrait de la méthode “decryptMessage” qui permet de décrypter le message selon l'id passé en paramètre :

```
try {
  const key = process.env.KEY_MSG;
  const algorithm = process.env.ALGO_MSG;

  const message :Query<...> & = await Message.findById(messageId);
  if (!message) {
    return null;
  }

  const iv :Buffer = Buffer.from(message.iv, encoding: "base64");
  const decipher :DecipherGCM = crypto.createDecipheriv(algorithm, key, iv);
  let decryptedText :string = decipher.update(message.text, inputEncoding: "hex", outputEncoding: "utf-8");
  decryptedText += decipher.final( outputEncoding: "utf8" );

  return decryptedText;
}
```

Implémentation de la méthode “decryptMessage” dans la méthode qui fetch un/des message(s) :

```
try {
  if (!req.params.hasOwnProperty(v: 'id')) {
    return res.status(400).json({ message: 'Missing room ID' });
  }

  const messages : Query<...>... = await Message.find( filter: {chatroom_id: req.params.id});
  let decryptedObj : any[] = [];
  for (let i : number = 0; i < messages.length; i++) {
    const decryptedText : ... = await decryptMessage(messages[i]._id);
    decryptedObj.push([
      decryptedText,
      messages[i].user_id,
      messages[i].createdAt,
    ]);
  }
}

res
  .status(200)
  .json({
    status: "ok",
    message: `Messages de la room ${req.params.id}`,
    data: decryptedObj,
  });
  
```

Le texte est donc stocké crypté en base de données, quand on veut le récupérer pour l'afficher dans la chat, ce dernier est requêté par son id, puis décrypté par la méthode “decryptMessage” puis renvoyé sous forme la forme d'un objet qui contient :

- le texte en clair
- l'user id de l'auteur du message
- la date de création du message

Conclusion

En conclusion, ce projet était l'occasion de collaborer avec des étudiants sur un projet de groupe.

C'était également l'occasion de mettre en commun toutes les connaissances et bonnes pratiques apprises dans nos alternances respectives.

Nous avons choisi de développer une application mobile de chat. Après une réflexion commune, nous avons choisis de baser notre application mobile sur une stack MERN (MongoDB, ExpressJS, React Native, NodeJS), à notre sens particulièrement adapté à ce type de projet.

C'est une stack composée majoritairement de langages JavaScript qui permettent de monter très rapidement des projets dynamiques tels qu'un chat instantané.

Ces choix de technologies, une méthode de travail AGILE, ainsi qu'une solide conception de projet au préalable nous ont permis de nous faciliter la tâche lors du développement des applications.

Afin de suivre le concept de la méthode agile, nous avons beaucoup communiqué tout au long du projet, ce qui a permis de nous remettre en question tout au long du projet, d'ajuster nos méthodes de travail, mais aussi de nous conforter dans nos choix quand cela a bien fonctionné.

J'ai beaucoup apprécié travailler avec d'autres étudiants, mais aussi le développement de l'API basée sur une base de données NoSQL. C'était une première pour moi avec ce type de base de données, j'ai trouvé Mongo très complet, agréable à manipuler et adapté à nos besoins.

Ce projet a été une expérience enrichissante tant sur le plan technique que relationnel, j'ai pris de conscience de l'importance d'avoir une conception solide pour mener à bien un projet de A à Z.

