Réalisation de l’application web et mobile

—

NEKO

ZIDI - Mohamed

SOMMAIRE

Introduction

Présentation personnelle

Je m'appelle Zidi Mohamed, j'ai 27 ans je suis en reconversion professionnelle suite à 5 ans passés dans les troupes de montagne. J'ai suivi le cursus de la Coding School de la plateforme en 2021 et j'ai obtenu mon titre de développeur web et web mobile. Aujourd'hui en dernière année de bachelor IT (spécialisation développeur web) afin de préparer mon titre de concepteur / développeur d'application web et en alternance au sein de l'entreprise Richardson depuis octobre.

Présentation du projet en anglais

During my last year at La Plateforme, I had to complete a project, and I decided to develop a mobile chat application. To do so, I used React Native framework for the mobile app development, VueJs for the admin panel, and NodeJS with ExpressJS for API development.

The chat app I created is called Neko, and the first objective was to allow employees of a company to have the possibilty of exchanging with each other internally.

Using the latest development technologies, I designed Neko to offer a smooth and enjoyable user experience.

Compétences couvertes par le projet

Ce projet couvre les compétences du titre suivantes :

- Maquetter une application
- Développer des composants d'accès aux données
- Développer la partie front-end d'une interface utilisateur web
- Développer la partie back-end d'une interface utilisateur web
- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données
- Concevoir une application
- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
- 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

Résumé du projet

J'ai développé une application mobile de chat pour une entreprise fictive. Cette application permet aux employés de communiquer entre eux de manière sécurisée et efficace. J'ai utilisé React Native pour la partie front-end et j'ai créé une API en Node.js avec Express pour gérer le back-end. L'API a été conçue pour être robuste et facile à utiliser, avec une sécurité renforcée pour protéger les données de l'entreprise.

En plus de l'application mobile, j'ai également créé un panel d'administration web en Vue.js. Le panel d'administration a été conçu pour permettre aux administrateurs de gérer les utilisateurs, les canaux de discussion et les paramètres de l'application. Tout cela a été soutenu par la même API en Node.js.

En résumé, j'ai créé une solution de communication interne complète pour l'entreprise fictive, avec une application mobile et un panel d'administration web, tous deux soutenus par une API robuste en Node.js. Cette solution a permis à l'entreprise de communiquer plus efficacement, tout en protégeant la sécurité de ses données sensibles.

Organisation et cahier des charges

Présentation de l’entreprise

Dans le cadre de la réalisation de ce projet Neko, une entreprise fictive a été créée afin d’obtenir une mise en situation la plus réaliste possible.
Neko Corp, développe une application mobile gratuite de discussion instantanée nommée Neko. Elle permet à ses membres de pouvoir échanger librement via des messages interposés.

Les besoins client

L’objectif

L'objectif est de proposer une application de chat à une entreprise ayant le but de permettre à ses employés d’échanger librement sur différents sujets concernant l’entreprise. Prévue pour être utilisée uniquement sur mobile, un travail en amont sera réalisé pour qu’elle soit compatible sur le

système d’exploitation mobile Android ainsi qu’iOs.

Les cibles

Pour que l’application soit accessible, il faudra absolument être inscrit.

Les inscrits : Il s’agit des utilisateurs qui pourront accéder à leur profil, le modifier mais aussi voir celui des autres membres. Ils auront aussi bien évidemment la possibilité d’échanger via le chat global de l’application avec tous les autres membres.

Les administrateurs : Il s’agit des utilisateurs qui ont un droit d’accès spécial à l’application. Ils pourront notamment supprimer et/ou modifier un user. De plus, pourra tout aussi supprimer un message sur le chat.

Le périmètre du projet

L’application sera disponible intégralement et uniquement en français.

Par ailleurs, l’application ne sera pas déclinée en version web, seulement le panel admin sera accessible en version web.

Les fonctionnalités attendues

Application mobile

Page d'accueil :

Lorsque l'utilisateur lance l'application mobile, il est redirigé vers une page d'accueil sur laquelle il doit se connecter ou s'inscrire.

Page Home :

Sur cette page, on y trouve les possibilités de navigation sur l'application, c'est-à-dire aller sur la page message, sur son profil …

Page messages :

Pages permettant d'envoyer des messages entrain les membres.

Page mon profil :

Page mise à jour de son profil. Modification de la photo de profil, pseudo.

Page profil de groupe :

Page permettant de voir les membres d'un groupe.

Panel admin

Le panel administrateur n'est accessible que via un site web qui permet une fois l'administrateur connecté de pouvoir supprimer un utilisateur, bannir un utilisateur et modifier le rôle d'un utilisateur.

Contexte technique

L'application mobile devra être accessible sur tous les systèmes d'exploitation Android et IOS. L'application web devra être accessible sur tous les navigateur

Conception du projet

Choix de développement

Choix des langages

Avec NodeJs comme environnement de développement.

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

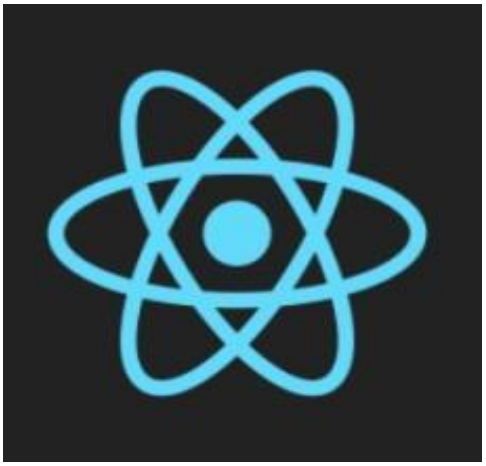
Choix des frameworks

Pour la création de mon API j’ai choisi d'utiliser Expressjs qui est un framework web léger de NodeJs souvent considéré par certains comme une librairie car il ne fournit pas de fonctionnalités prédéfinies pour les aspects plus avancés d'une application web, tels que l'authentification, la gestion des bases de données, la gestion de la sécurité, etc. Souvent, les développeurs ajoutent des modules tiers pour répondre à des besoins spécifiques.

J’ai choisi Express car il a un cadre d’exploitation libre et gratuit. Il comporte un ensemble de paquets, pour des fonctionnalités, des outils

qui aident à simplifier le développement.
Ayant un calendrier à respecter, Express Js permet de développer, de façon rapide et efficace une API. Ce qui correspond parfaitement à mon besoin.

Pour la création de mon application mobile, j'ai choisi react native car il est écrit en javascript et qu'il permet de developper des projets pour android et iOS.



Logiciels et autres outils

Dans le cadre de ce projet, j'ai dû utiliser d'autres outils :

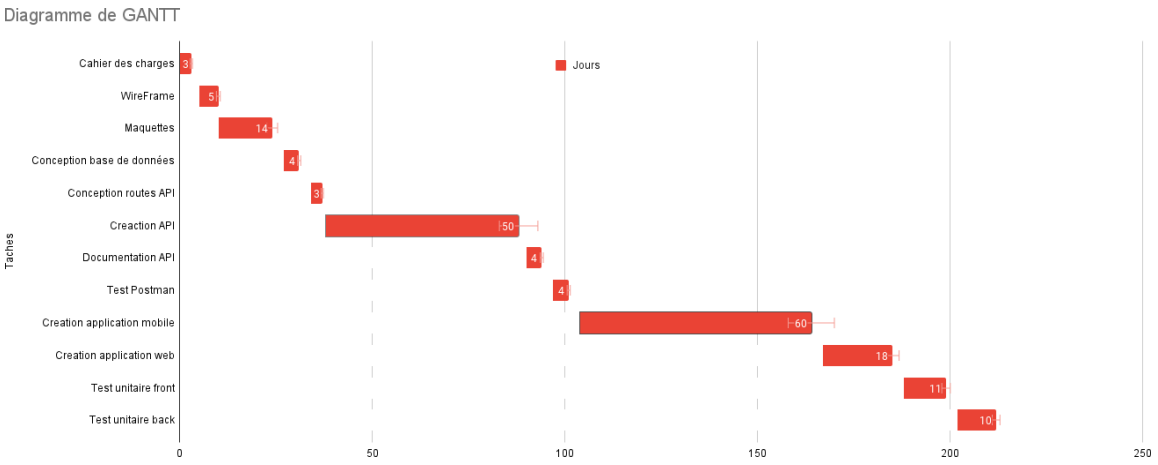
- Visual Studio Code pour écrire mon code ;
- Postman pour effectuer les requêtes API ;
- TortoiseGit et GitHub pour le versionning de mon code ;
- Trello pour organiser mon travail ;
- NPM pour installer les paquets ;
- Adobe XD pour la création de mes maquettes ;
- LucidChart pour le maquettage de ma base de données
- Expo pour émuler mon application mobile sur mon téléphone ;
- Canva pour la création de ma charte graphique et du logo.

Organisation du projet

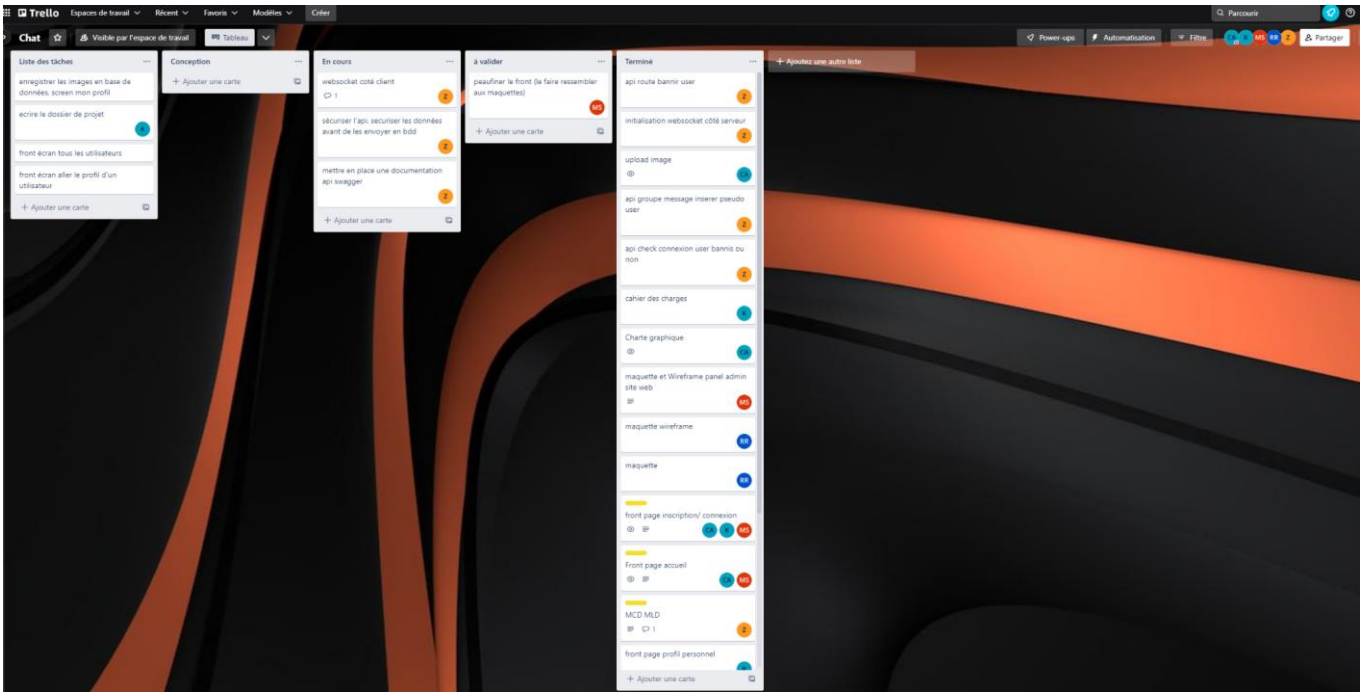
Ce projet a été réalisé durant mon année de formation en groupe, avec des temps d’entreprise et d'autres projets à rendre. Donc l'organisation du travail a été essentielle. On a commencé le projet par un listing des tâches.

Premièrement, j’ai tout d’abord créé un diagramme de gantt.

C’est un outil qui m’a permis de planifier mon projet, ainsi d'avoir une vue d’ensemble des tâches à effectuer. Il m’a permis de suivre l'avancement de mon projet, et faire des ajustements afin d’optimiser le temps que j’avais sur ce projet.

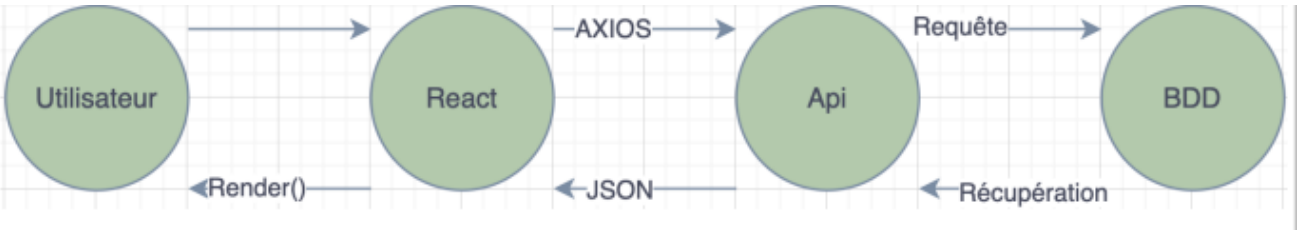


On a utilisé des outils comme trello afin de lister les tâches à effectuer et les attribuer à un membre.



Architecture logicielle

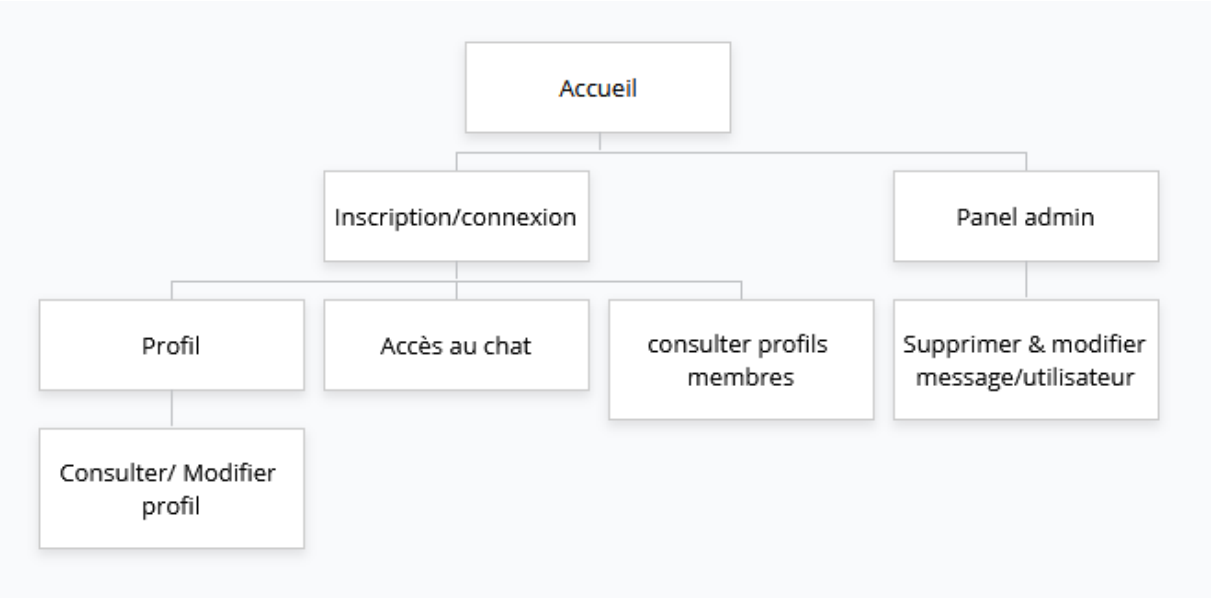
Les utilisateurs qui ont un statut salarié auront uniquement accès à l'application mobile. Les utilisateurs avec un statut DRH auront accès à la fois à l'application mobile et au site web. L'application mobile et le site web utilisent la même API.



Conception du front-end de l’application

Pour créer la maquette et la charte graphique, on a utilisé figma, simple d’utilisation et permet la réalisation de maquettes réalistes.

Arborescence du projet



Charte graphique

Pour la charte graphique, il fallait définir, une palette de couleurs, les logos et les polices utilisées pour la typographie afin d’obtenir une identité visuelle.

La palette de couleurs se compose donc de quatre couleurs dont le jaune, l’orange, le bleu marine et du beige. Avec ces couleurs l’objectif est d’avoir un visuel attrayant pour l'utilisateur, tout en restant sobre et épuré.

Palette de couleurs :

Concernant la typographie, le premier choix s’est porté sur la police Poppins de Google Fonts. Son esthétique donne une écriture élégante, simple et très agréable. Quant au deuxième choix de police, il s’agit de Roboto, cette police créée par Christian Robertson pour la plateforme Android. Son intérêt est d’apaiser la lecture en ligne des utilisateurs et utilisatrices en permettant à la police de s'adapter à tout type d'écran.

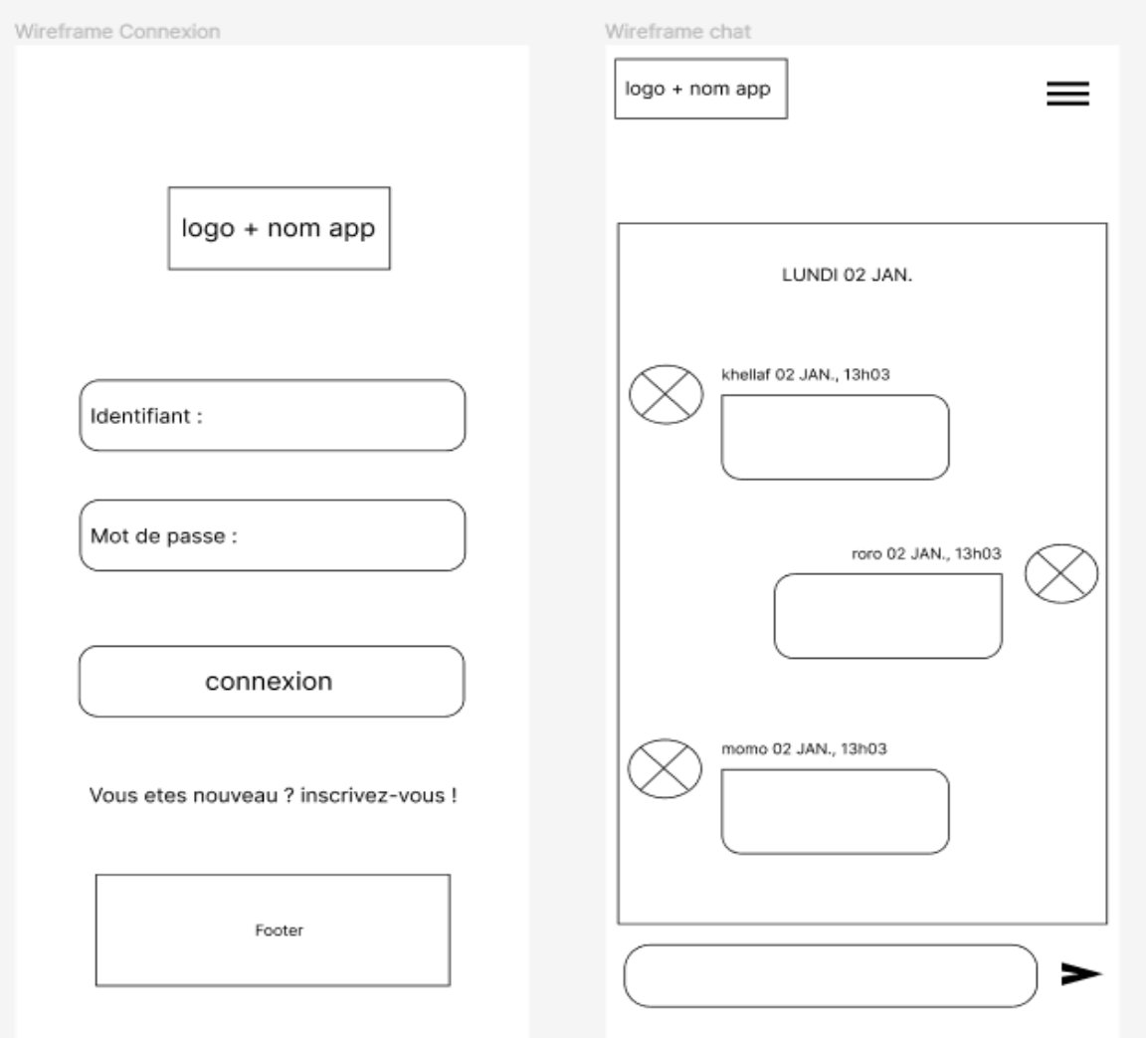
Les logos font partie de l’identité visuelle de l’application Neko et **constituent un point de repère et véhicule l’image renvoyée auprès des utilisateurs. Ils** sont déclinés au nombre de trois :

Les logos de l'application permettent d'habiller les pages de l'application et de faciliter la navigation en associant des repères visuels à certaines actions pour plus de convivialité lors du parcours des utilisateurs.

Maquettage

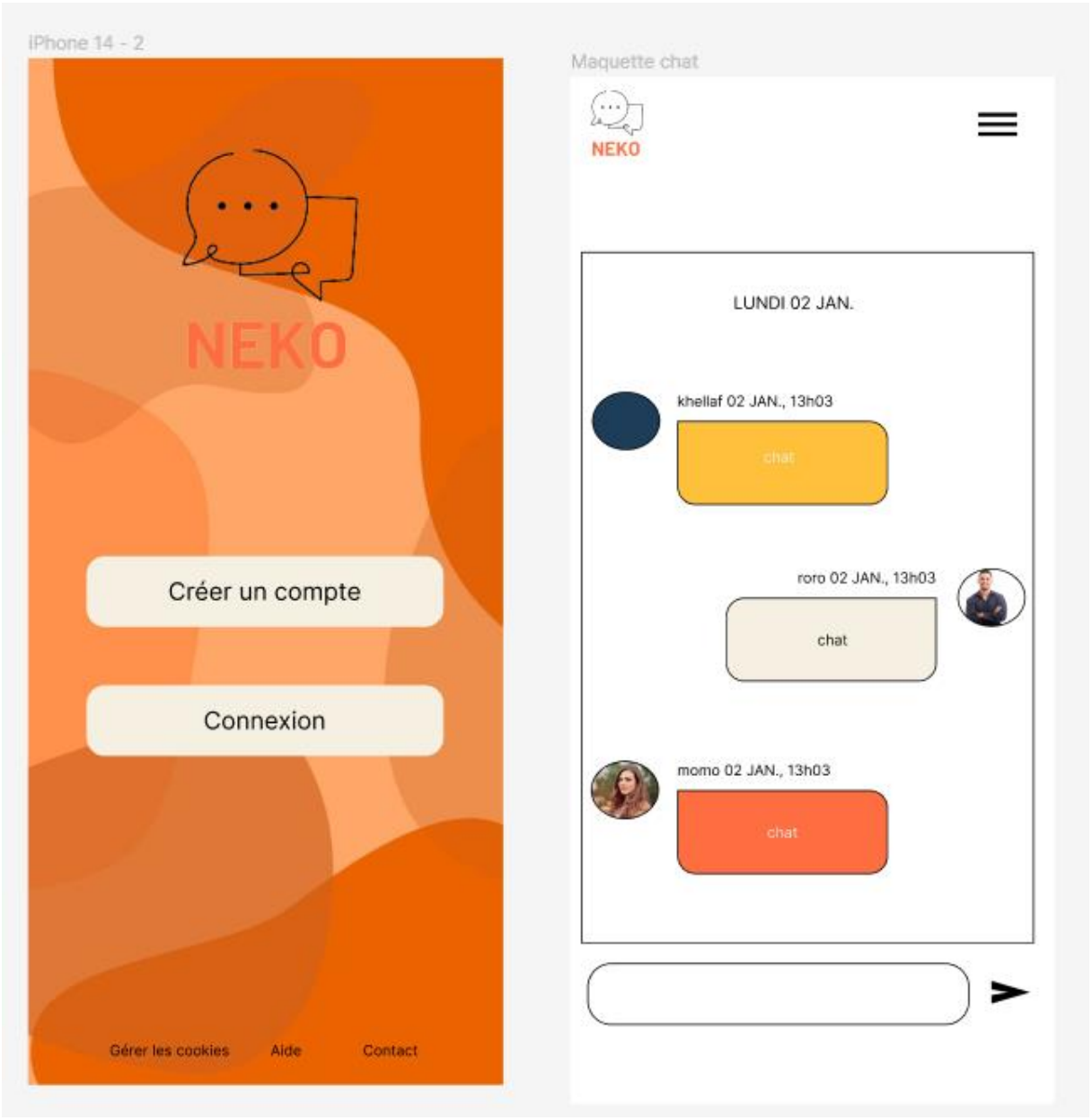
Les maquettes ont été réalisées sur le site gratuit **Figma.**
Les maquettes **Wireframe** permettent de visualiser comment agencer des éléments sur les différentes pages.
Le choix s'est porté sur un design minimaliste, pour rester dans le style Flat Design très en vogue dans le web.

Maquettes des écrans d'inscription et du chat entre membres (wireframe) :



Par la suite, des maquettes **Design** ont été réalisées afin de mettre en avant une identité visuelle de l'application. Le but étant que l'utilisateur puisse trouver l'application attrayante et agréable car la première impression qu'un utilisateur fait sur un site est la partie front-end.

Maquettes des écrans d'inscription et du chat entre membres (Design) :


iPhone 14 - 2
NEKO
Créer un compte
Connexion
Gérer les cookies
Aide
Contact
Maquette chat
NEKO
LUNDI 02 JAN.
khellaf 02 JAN., 13h03
chat
roro 02 JAN., 13h03
chat
momo 02 JAN., 13h03
chat

Conception backend de l’application

La base de données

Mise en place de la base de données

Pour ce projet, il est indispensable d’avoir une base de données, afin de stocker, centraliser de façon sécurisée les données des utilisateurs. De plus une base de données va permettre d’optimiser l’organisation du travail et de récupérer les informations rapidement.

Après plusieurs recherches sur le framework back-end Express JS, il s’est avéré que les bases de données les plus utilisées avec l’environnement d’exécution Node JS sont les bases de données NoSQL.

Durant ma formation, j’ai utilisé uniquement des bases de données relationnelles pour mes différents projets. Pour la réalisation de cette application, je souhaitais acquérir de nouvelles connaissances et compétences, par conséquent, j’ai choisi d’utiliser une base de données non relationnelle.

Ainsi pour créer une base de données NoSQL et la stocker, j’ai utilisé MongoDB.

Ce dernier est un SGBD orienté documents c’est à dire que les données sont stockées sous forme de documents plutôt que dans un format relationnel, il stocke les données dans des collections qui contiennent plusieurs documents sous forme de collections.

MongoDB est flexible et permet d’associer & stocker plusieurs types de données

Avoir une base de données NoSql permet de stocker et de gerer des **volumes de données** plus importants par rapport à une BDDR.

Conception de la base de données

La conception d'une base de données NoSQL avec MongoDB implique une approche flexible et adaptative pour modéliser les données.

La première étape a été d'identifier les besoins spécifiques de l'application et de décider quel type de base de données NoSQL serait le plus adapté pour répondre à ces besoins.

Une fois que le choix de l’utilisation de mongoDb a été établi, la conception de ma base de données pouvait commencer.

J’avais l’habitude d’utiliser la méthode merise pour conceptualiser ma base de données or pour les bases de données NoSQL, il n'existe pas de méthode de conception standardisée comparable à la méthode Merise. Cela dit, il est tout de même recommandé de suivre certaines bonnes pratiques pour garantir l'efficacité et la performance de la base de données. Par exemple, il est important de bien comprendre les besoins de l'application et de choisir le type de base de données NoSQL le plus approprié pour répondre à ces besoins.

Il est également important d’avoir des schémas implicites pour les documents enregistrés dans les différentes collections de la base de données. Dans le cadre de mon projet je souhaitais enregistrer les informations concernant les utilisateurs et les messages

que ces derniers pouvaient envoyer. Pour la collection “utilisateurs” je définis un document pour chaque utilisateur contenant des informations telles que l'identifiant de l'utilisateur, son nom, son adresse email, son mot de passe, etc. Pour la collection "messages", je définis un document pour chaque message contenant des informations telles que l'identifiant du message, l'identifiant de l'utilisateur qui a posté le message, le contenu du message, la date et l'heure du message, etc.

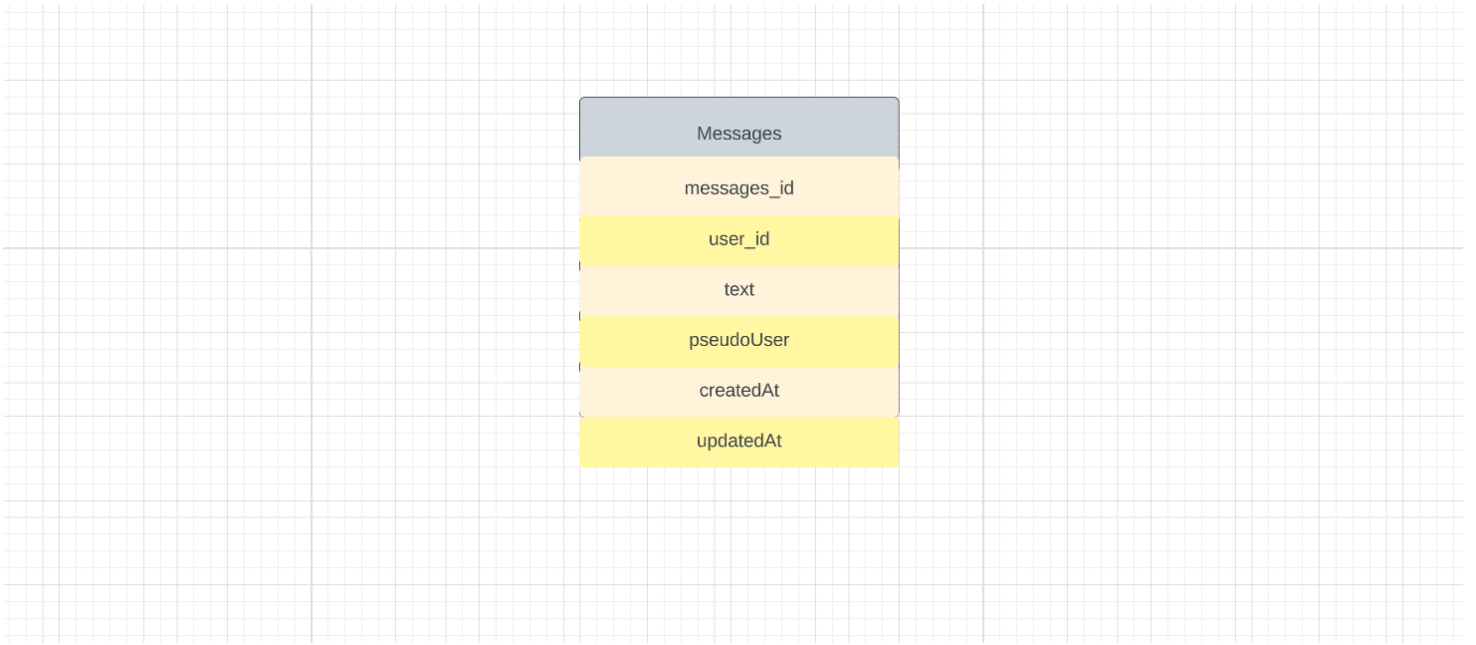
J’ai dû également adopter une approche de stockage imbriqué pour stocker directement certaines données d’un document étranger dans un document principal plutôt que de lier mes documents à l’aide de références telles que des identifiants. Cela permet d'éviter d’effectuer des requêtes supplémentaires donc d’avoir un gain de performance dans mon application.
Voici un aperçu des données pertinentes qui m’ont aidé à construire une représentation claire des besoins :

- Pour les utilisateurs : j’ai besoin de stocker le pseudo, le mail, le mot de passe, la bio, son droit utilisateur, son statut de connexion et si cet utilisateur a été banni.
- Pour les messages : j’ai besoin de stocker le corps du message, l’id de l’utilisateur, le pseudo de l’utilisateur, date de création et la date de mise à jour du message.

J’ai construit ma base de données noSql après avoir cerné les besoins, en procédant à la construction de modèles pour définir les documents enregistrés sous forme de collections :

- Modèle de données utilisateur :

- Modèle de données message :



Développement du backend de l’application

Organisation

Mon back end a pour but d'être utilisé à la fois par mon application mobile et mon espace administrateur. Le développement d’une API pour ne pas répéter ma logique métier était la meilleure solution.

J’ai développé mon back en utilisant le principe de separation of concern SoC (séparation des préoccupations en français), j'ai pris des mesures pour améliorer son efficacité en me concentrant sur la logique et l'optimisation de mon code. Cette approche a permis d'augmenter considérablement la lisibilité du code et de le rendre plus facile à maintenir pour les prochaines versions. J'ai également évité les répétitions en créant des fonctions et des services, et en divisant la logique de mon code en différents fichiers et services.

Arborescence

Toujours en essayant de respecter le principe de séparation des préoccupations, j’ai décidé de suivre une architecture multi-tiers (ou N-tiers). Ce type d’architecture permet de rendre une application plus modulaire, évolutive et facile à maintenir, en réduisant les interdépendances entre les différentes parties de l'application et en permettant une gestion plus fine des mises à jour et des changements.

Mon back end est donc composer des dossiers suivants :

- **Routes** : regroupant tous les fichiers de mes routes
- **Controllers** : Regroupant tous les controllers
- **Middleware** : Contient tous les middleware utilisés pour le fonctionnement de l'API
- **model** : contenant les modèles de toutes mes tables
- **Config** : contient ma connexion à la base de données
- **Utiles** : contient les librairies et fichiers utiles
- **test** : destiné aux test unitaire - Newman

Fonctionnement de l'API

Lorsqu'un client envoie une requête à mon API, le routeur analyse l'URL et détermine la route et la méthode appropriées pour la requête. En fonction de cela, un contrôleur est appelé pour gérer la demande. Le contrôleur, à son tour, fait appel à un service qui interagit avec le modèle pour récupérer les données nécessaires. Les données sont ensuite analysées par le service et une réponse est renvoyée au client au format JSON, accompagnée d'un code d'état correspondant. En résumé, le processus consiste en une séquence de traitement organisée et modulaire, où chaque couche est responsable d'une tâche spécifique, afin de fournir une réponse précise et complète au client.

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 créé
  - **204** : NO - CONTENT
  Indique que la requête a bien été effectué et qu'il n'y aucune réponse à envoyer
  - **400** : BAD REQUEST
  Indique que le serveur ne peux pas comprendre la requête a 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
  - **405** : METHOD NOT ALLOWED
  Indique que la requête est connue du serveur mais n'est pas prise en charge pour la ressource cible
  - **500** : INTERNAL SERVER ERROR
  Indique que le serveur a rencontré un problème.

Les différentes méthodes HTTP utilisées dans ce projet :

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

Middleware

Un middleware est une couche logicielle qui se situe entre deux autres couches de logiciels. Il s'agit d'une simple fonction qui remplit une tâche spécifique. Dans le cadre du framework Express, un middleware est une fonction intercalée entre la requête et la réponse.

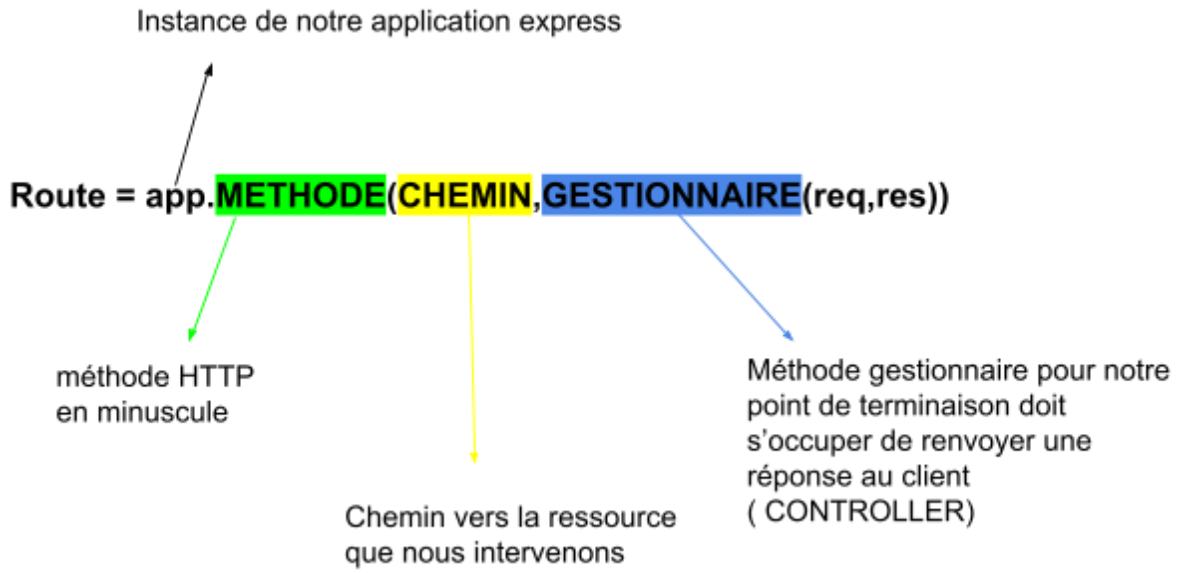
Cette fonction a accès aux paramètres de la requête et de la réponse, ce qui permet d'effectuer de nombreuses actions. L'un des objectifs du middleware est de vérifier les données envoyées, par exemple dans le corps des requêtes. Il est possible d'appliquer un middleware à une seule route ou à plusieurs.

Bien que le framework propose quelques middlewares par défaut, il est possible d'en créer de nouveaux selon les besoins du projet.

Routage

Afin de mettre en place le routage, j'ai utilisé le routeur de express js.
ExpressJs met à disposition un middleware qui gère facilement le routage du projet.

Pour déclarer une nouvelle route à express , on peut le résumer à cette exemple :



Comme mentionné précédemment, mon objectif a été de rendre mon code facilement modifiable et maintenable. Pour ce faire, j'ai divisé cette partie en plusieurs étapes. La première chose que j'ai faite a été de séparer les routes et le code d'implémentation associé.

```
// ROUTES

app.use('/api/users', require('./routes/usersRoute'));
app.use('/api/messages', require('./routes/messagesRoute'));
```

Ce code qui est un exemple tiré de mon projet utilise la méthode use du framework Express en JavaScript pour monter deux routes distinctes de l'API : /api/users et /api/messages.

La méthode use permet de monter un middleware ou une route à une URL spécifique. Dans ce cas, elle est utilisée pour monter les routes des utilisateurs et des messages.

La syntaxe require('./routes/usersRoute') et require('./routes/messagesRoute') est utilisée pour importer les fichiers JavaScript qui définissent les routes pour les utilisateurs et les messages respectivement.

En utilisant cette syntaxe, lorsqu'un client effectue une requête HTTP sur l'URL /api/users, Express traite cette demande en utilisant les routes définies dans le fichier usersRoute.js, tandis que pour l'URL /api/messages, les routes définies dans messagesRoute.js sont utilisées.

En utilisant cette structure, il est plus facile de diviser les fonctionnalités de l'API en différentes parties et de les organiser en conséquence.

Cela rend également le code plus facile à maintenir et à modifier car chaque route a son propre fichier et ses propres fonctions pour gérer les demandes

```
const router = express.Router();
const { protect } = require('../middleware/authMiddleware');

router.route('/:id').get(protect, getUser).delete(protect, deleteUser);
```

```
// @desc Recupérer un user
// @route GET /api/users/:id
// @access private
const getUser = asyncHandler(async (req, res) => {
    await User.findById(req.params.id)
        .then((user) => {
            res.status(200).json(user);
        })
        .catch((error) => res.status(400).json({ error: "l'Utilisateur
n'existe pas" }));
})
```

Ce code définit une route pour la gestion des utilisateurs de l'API. Il utilise la méthode Router d'Express pour créer un objet de routeur qui peut être utilisé pour définir des routes.

La première ligne crée un nouvel objet de routeur en appelant la méthode Router() d'Express et en l'assignant à la variable router.

La deuxième ligne importe un middleware nommé protect du fichier authMiddleware.js situé dans le dossier middleware. Ce middleware est utilisé pour protéger certaines routes de l'API en exigeant que l'utilisateur soit authentifié avant de pouvoir accéder à ces routes.

La troisième ligne utilise la méthode route() pour définir une route sur l'URL / pour cette instance de routeur. Cette route répond aux méthodes HTTP GET et DELETE. En utilisant la méthode get() et en lui passant la fonction getUser, cette route permet de récupérer les informations d’un utilisateur dans la réponse avec son id. En utilisant la méthode delete() et en passant le middleware protect suivi de la fonction deleteUser, cette route permet de supprimer un utilisateur en utilisant son identifiant.

Controller

Ce dossier contient tous les controllers avec toutes les methodes associées aux différentes routes.

Config

Mon dossier config comporte un fichier qui gère la connexion à ma base de données. J’utilise un ORM (Object-Relational Mapping) pour me connecter à ma base de données et pour gérer toutes les requêtes à celles-ci (CRUD).

Mongoose offre une interface simple et intuitive pour interagir avec une base de données MongoDB, en utilisant des schémas et des modèles pour structurer les données et faciliter leur manipulation.

En utilisant Mongoose, il est facile de définir un schéma pour chaque modèle de données que l'on souhaite stocker dans la base de données. Les schémas permettent de spécifier les champs, les types de données et les contraintes pour chaque document, ainsi que les méthodes et les fonctions pour la manipulation de ces documents.

Mongoose facilite également la connexion à la base de données, en offrant des options pour la configuration de la connexion et la gestion des erreurs de connexion. On peut ainsi se connecter à la base de données en utilisant une URL de connexion unique, ou en spécifiant les détails de connexion de manière explicite.

Model

Dans mon dossier models se trouve les schemas des documents que j’enregistre en base de données.

```
const mongoose = require('mongoose');

const userSchema = mongoose.Schema({
    pseudo: {
        type: String,
        required: [true, 'Pseudo requis']
    },
    mail: {
        type: String,
        required: [true, 'Mail requis'],
        unique: true
    },
    password: {
        type: String,
        required: [true, 'Password requis']
    },
    bio: {
        type: String,
        required: false
    },
    image: {
        type: String,
        required: false
    },
    isAdmin: {
        type: String,
        default: 'false',
    },
    isConnect: {
        type: String,
        default: 'false',
    },
```

Sécurité

Les API permettent d'accéder à des informations stockées dans une base de données, mais elles présentent également des risques de sécurité. Voici quelques exemples de risques courants :

- Les injections SQL : ces attaques consistent à insérer un code malveillant dans une requête pour accéder illégalement à une base de données et en prendre le contrôle. L'attaquant peut alors accéder à toutes les informations stockées dans la base de données.
- Credential stuffing : cette attaque consiste à voler les identifiants de connexion d'un utilisateur.
- Attaques DDoS : ces attaques visent à surcharger une API en envoyant une grande quantité de trafic.
- Man-in-the-middle : cette attaque consiste à rediriger un utilisateur vers un service compromis, permettant à l'attaquant de récupérer les identifiants ou les clés de l'utilisateur.

Pour sécuriser mon API j’ai mis en place plusieurs actions que je vous décrirai dans ce chapitre.

Chiffrement des données sensibles

Les données sensibles des utilisateurs tels que les mots de passe ne sont pas stockés en dur dans la base de données. J’ai décidés des les stocker hachés pour cela des, j'utilise bcrypt.

Bcrypt est une bibliothèque de hachage de mots de passe qui est couramment utilisée pour stocker des mots de passe de manière sécurisée dans les applications web. Cette bibliothèque utilise une fonction de hachage unidirectionnelle, qui prend un mot de passe en entrée et génère une chaîne de caractères aléatoire appelée "hash" en sortie.

JWT

Dans le but d’identifier un utilisateur authentifié, j’utilise un token JWT (JSON web token).

Le JWT est un jeton auto-suffisant qui contient toutes les informations nécessaires pour vérifier l'identité de l'utilisateur et les autorisations accordées. Il est créé en signant numériquement un ensemble de données (payload) avec une clé secrète ou une paire de clés publique-privée. Le jeton peut ensuite être envoyé à un serveur pour prouver l'identité de l'utilisateur et accéder aux ressources protégées.

Le principal avantage des JWT est qu'ils sont portables et peuvent être utilisés dans des environnements distribués, tels que les microservices et les applications mobiles. De plus, les JWT sont généralement auto-expirants, ce qui signifie qu'ils ont une durée de vie limitée et ne peuvent être utilisés que pendant une période spécifiée.

Je sécurise les routes privées de mon api avec le JWT, une fois identifié l’utilisateur a accès à certaines routes protégées.

Ce jeton est composé de trois parties :

- Un header : détermine l’algorithme utilisé pour générer la signature
- un payload : Il s'agit de la section contenant les informations de l'utilisateur, sous forme de chaîne de caractères encodée en base 64 et hachée.
- La signature : Celle-ci est créée à partir du header et du payload générés et d’un secret. Une signature invalide implique systématiquement le rejet du token. La signature du jeton a une importance fondamentale , il sert à vérifier que les informations connues sont inchangées.

Lorsqu'un utilisateur tente de se connecter à son compte, une requête est envoyée au serveur. Si les informations fournies sont correctes, le serveur répondra en renvoyant un jeton sous forme de JSON. Ce jeton contient des informations sur l'utilisateur connecté, telles que son identifiant, son adresse e-mail et son rôle. Par la suite, le client inclura ce jeton dans toutes les demandes ultérieures

Gestion des Droits

Pour sécuriser les routes, j’ai développé un système de droits. Pour ce faire, j’ai créé un middleware.

```
const protect = asyncHandler(async (req,res,next) => {
    let token;

    if(req.headers.authorization &&
req.headers.authorization.startsWith('Bearer')){
        try {
            // Récupération du token depuis le header
            token = req.headers.authorization.split(' ')[1];
            // Vérification du token
            const decoded = jwt.verify(token, process.env.JWT_SECRET);
            // Récupération de l'utilisateur du token sans le password
            req.user = await User.findById(decoded.id).select('-password');
            next();
        } catch (error) {
            console.error(error);
            res.status(401);
            throw new Error('Acces interdit, token invalide');
        }
    }
    if(!token){
        res.status(401);
        throw new Error('Acces interdit, manque de token');
```

Ma fonction protect est un middleware que j’utilise pour protéger les routes d’utilisateur non authentifié. Elle vérifie la présence d'un token JWT valide dans l'en-tête "Authorization" de la demande HTTP.

Si un token est présent et valide, la fonction décrypte le token, extrait l'identifiant de l'utilisateur et récupère les informations de l'utilisateur à partir de la base de données. Les informations de l'utilisateur sont stockées dans l'objet "req.user" pour permettre aux routes suivantes d'y accéder.

Si le token est manquant ou invalide, la fonction renvoie une erreur HTTP 401 "Accès interdit" avec un message approprié.

Cette fonction est basée sur la bibliothèque "jsonwebtoken" qui est utilisée pour générer et vérifier les tokens JWT.

Par exemple, pour avoir accès à la route permettant d’ecrire un message il faut préalablement que l’utilisateur se soit inscrit puis authentifié pour qu’un token lui soit fourni, lui permettant d’avoir accès à cette route sinon il lui sera renvoyé une erreur

```
const { protect } = require('../middleware/authMiddleware');

router.route('/').get(protect, getAllMessages).post(protect, setMessage);
```

Helmet

ExpressJs est un framework robuste mais il n'est pas parfait en matière de sécurité, rien ne protège le serveur nodeJs des vulnérabilités.
Par défaut, expressJs laisse les entêtes HTTP pour faciliter le développement des projets. Dans l'entête de la requête, on peut découvrir que l'application a été créée avec express, l'utilisateur n'est pas obligé de le savoir. Au contraire, un utilisateur malveillant peut s'en servir pour repérer les failles de ce framework.

C'est pour cela que j'ai choisi d'utiliser Helmet.js .
Il sécurise l'application Node.js contre certaines menaces comme les XSS, Content Security Policy et autres.
Helmet est livré avec une collection de modules Node. Ils permettent de configurer les en-têtes et d'empêcher les vulnérabilités.

Voici les en-têtes utilisés par Helmet pour sécuriser le serveur :

- Content-Security-Policy : pour la protection contre les attaques de type cross-site scripting et autres injections intersites.
- X-Powered-By : supprime le header X-Powered-By. Ce dernier leak la version du serveur et son vendor.
- Strict-Transport-Security : impose des connexions (HTTP sur SSL/TLS) sécurisées au serveur.
- Cache control : définit des headers Cache-Control et Pragma pour désactiver la mise en cache côté client.
- X-Content-Type-Options : pour protéger les navigateurs du reniflage du code MIME d'une réponse à partir du type de contenu déclaré.
- X-Frame-Options : définit l'en-tête X-Frame-Options pour fournir une protection clickjacking.
- X-XSS-Protection : active le filtre de script inter sites (XSS) dans les navigateurs Web les plus récents.

Pour le mettre en place, il faut l'utiliser comme un middleware, dans la fonction use de l'objet express.

Express validator

Express Validator est une bibliothèque JavaScript open source qui offre une solution de validation et de sanitization de données pour les applications Node.js utilisant le framework Express. Elle permet de valider les données de la requête HTTP entrante avant qu'elles ne soient traitées par les routeurs ou les contrôleurs.

Avec Express Validator, vous pouvez facilement définir des schémas de validation pour les paramètres de requête, les corps de requête et les paramètres d'URL, et définir des règles de validation pour chaque champ de données. Si les données de la requête ne satisfont pas aux règles de validation, Express Validator renvoie une erreur avec un message d'erreur approprié.

L’utilisation de cette bibliothèque me permet d’améliorer la sécurité de mon application en empêchant les attaques d'injection de code et les erreurs de traitement de données, tout en offrant une meilleure expérience utilisateur en affichant des messages d'erreur clairs et précis.

Exemple de Problématique rencontrée

Mon application étant une application de chat en temps réelle, l’utilisation des websockets était logique pour la réussite de celui-ci.

Les websockets sont une technologie de communication en temps réel qui permettent une communication bidirectionnelle entre un navigateur et un serveur Web. Contrairement aux requêtes HTTP traditionnelles, qui sont des connexions ponctuelles, les websockets permettent une communication bidirectionnelle en temps réel entre le navigateur et le serveur. Cela signifie qu'une fois qu'une connexion websocket est établie, les données peuvent être envoyées et reçues simultanément, sans la nécessité de fermer et de rouvrir une connexion pour chaque échange de données.

Pour mettre en place une connexion websocket, vous devez d'abord établir une connexion entre le client (le navigateur Web) et le serveur. Le client envoie une demande d'ouverture de connexion websocket au serveur, qui répond avec une réponse d'acceptation si la demande est autorisée. Une fois la connexion établie, les données peuvent être envoyées et reçues par le biais de la connexion.

Cependant, il y a des difficultés potentielles que vous pourriez rencontrer en travaillant avec les websockets. L'une des principales difficultés est la gestion de la connexion websocket elle-même. Comme les websockets sont des connexions persistantes, il est important de mettre en place des mécanismes pour surveiller et gérer la santé de la connexion. Cela peut inclure la gestion de la déconnexion

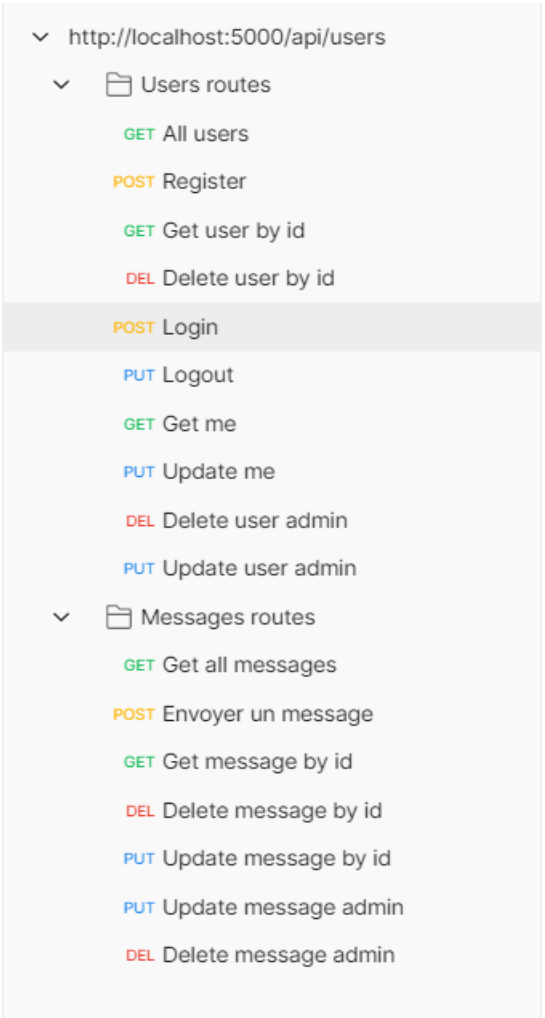
inattendue, la reconnexion automatique et la gestion des erreurs.

Recherches anglophones

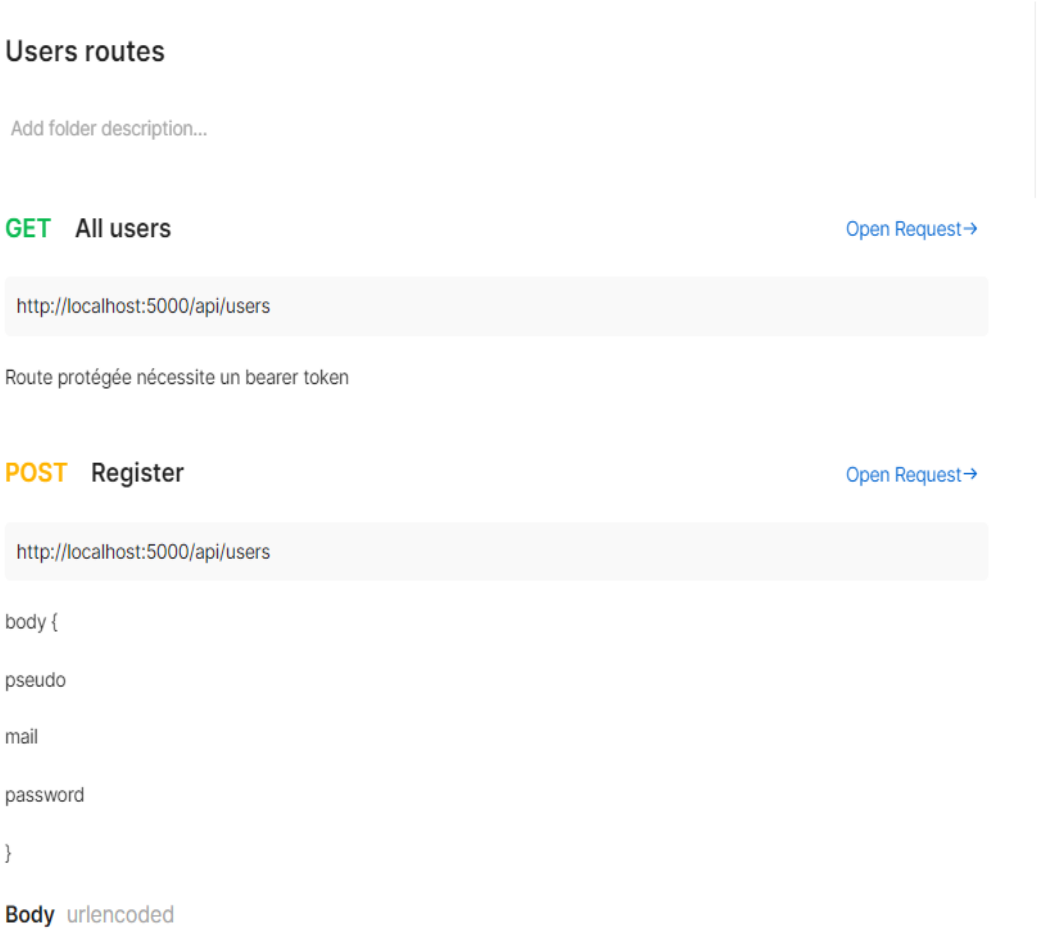
Pour résoudre le problème cité juste au-dessus, j'ai parcouru stack overflow mais aussi la documention de socketio qui est en anglais. J'ai également parcouru plusieurs documentations en anglais pour intégrer certaines bibliothèques à mon projet tel que helm, express validator.

Documentation

L'objectif d'une API est de permettre à d'autres développeurs de l'utiliser, d'où l'importance d'une documentation claire et précise. La documentation inclut des informations techniques et des instructions détaillées sur le fonctionnement de l'API, et doit être mise à jour régulièrement pour refléter les modifications apportées au code ou l'ajout de nouvelles fonctionnalités.



La documentation de l’API est écrite directement sur postman. Postman est un outil de test d'API populaire qui peut également être utilisé pour générer de la documentation.



Tests

Postman

Mon API expose des données, il est donc important de tester son API afin de s’assurer de la qualité des données exposées.
A chaque modification ou création de route, j’ai effectué des tests avec Postman afin de s'assurer du fonctionnement ou non de mon API.

A chaque test, je vérifie que les données envoyées sont bien celles attendues, qu'elles envoient les données, que le statut de la requête HTTP est correct. Je vérifie aussi que les erreurs sont bien gérées.

Développement Frontend De L’Application

Principales fonctionnalités

Contexte d’authentification

Mon application utilise l'authentification pour permettre aux utilisateurs de se connecter et d'accéder aux fonctionnalités de l'application. L'authentification se fait à travers une API backend, qui reçoit des demandes d'authentification des clients et vérifie les informations d'identification fournies par l'utilisateur pour autoriser l'accès à l'application.

Lorsqu'un utilisateur se connecte, les informations d'identification sont envoyées à l'API backend qui renvoie une réponse contenant un token d'authentification si les informations sont correctes. Ce token d'authentification est stocké dans le contexte d'authentification de l'application pour permettre à l'utilisateur de naviguer dans l'application sans avoir à se reconnecter à chaque fois.

Si l'utilisateur se déconnecte de l'application, le token d'authentification est supprimé du contexte d'authentification. Lorsqu'un utilisateur tente d'accéder à une fonctionnalité protégée, l'application vérifie d'abord si le token d'authentification est présent dans le contexte d'authentification pour autoriser l'accès. Si le token n'est pas présent ou est invalide, l'utilisateur est redirigé vers l'écran de connexion.

```
import { createContext, useContext } from "react";
import { useState } from "react";

const authContext = createContext(null);

export const useAuth = () => {
    return useContext(authContext)
};

const AuthProvider = ({children}) => {
    const [user,setUser] = useState(null);
    return(
        <authContext.Provider value={[user,setUser]}>
            {children}
        </authContext.Provider>
    )
}
```

Ci-dessus je définis un contexte d'authentification avec la fonction createContext() de React. Le contexte est initialisé à null. Ensuite, la fonction useAuth() permet d'accéder à ce contexte dans d'autres composants de l'application.

Le composant AuthProvider utilise le hook useState() de React pour stocker les informations d'authentification de l'utilisateur, qui sont passées au contexte avec la fonction Provider. Cela permet de propager ces informations dans toute l'application et de les rendre accessibles depuis n'importe quel composant qui utilise useAuth().

Ainsi, lorsqu'un utilisateur se connecte ou se déconnecte, la modification de l'état du composant AuthProvider entraîne automatiquement la mise à jour de l'état du contexte d'authentification. Les composants qui utilisent useAuth() peuvent alors accéder aux informations d'authentification à jour sans avoir à les stocker localement ou à passer les informations de composant en composant.

```
axios.post(`http://${IP_ADRESS}:5000/api/users/login`, {
              mail: username,
              password: password,
            })
            .then( (response) => {
              setUser({
                ...response.data,
                idToken: response.data.token
              });
            console.log(user);
            })
            .catch( (error) => {
              console.error(error);
              console.log(error);
              setErrorMessage(error.response.data.message)
            });


export default HeaderInformations;
```

Voici l’appel à l’api pour la connexion utilisateur, j’utilise la librairie Axios pour envoyer une requête POST vers une API.
La requête envoie un objet contenant l'adresse email et le mot de passe saisis par l'utilisateur, afin d'authentifier l'utilisateur.
Si la requête est réussie, les données renvoyées par l'API incluant le token d’identification sont enregistrées dans le context utilisateur pour lui permettre de naviguer sur l’application.Si la requête échoue, le code affiche un message d'erreur à l'utilisateur.

Websockets

Pour utiliser les websockets j'utilise Socket.IO qui est une bibliothèque JavaScript qui permet de créer des applications en temps réel sur le Web en utilisant des WebSockets, une technologie de communication bi-directionnelle entre un serveur et un client.

Les WebSockets permettent une communication en temps réel entre un navigateur et un serveur, en évitant les limitations liées aux requêtes HTTP traditionnelles.

```
socketService.initializeSocket();
    socketService.on("socket_message", (msg) => {
      fetchMessages();
      setMessages(messages => [...messages, msg]);
    });
```

Ces deux lignes de code utilisent une instance de socketService pour initialiser une connexion socket et écouter des événements.

La première ligne initialise une connexion avec le serveur via une connexion socket.

La seconde ligne écoute l'événement "socket_message" émis par le serveur via la connexion socket. À chaque fois que cet événement est reçu, cela déclenche une fonction qui appelle fetchMessages() pour récupérer les derniers messages du serveur, puis utilise la fonction setMessages pour ajouter le nouveau message à l'état local de l'application.

Une fois cet événement émit, il sera ensuite diffusé à tous les clients connectés.

Sécurité

La sécurité de mon application a été renforcée grâce à l'utilisation de regex (expressions régulières). Les regex permettent de valider les données entrées par l'utilisateur, comme les champs de formulaire, en s'assurant qu'elles répondent à des critères spécifiques. Par exemple, j’ai utilisé des regex pour valider les adresses email, les mots de passe, les numéros de téléphone, les URL, etc.

Cela peut aider à prévenir les attaques malveillantes telles que l'injection de code et les attaques par déni de service. En utilisant des regex pour valider les données entrées par l'utilisateur, j’ai pu m’assurer que les données sont correctes et éviter ainsi les failles de sécurité potentielles.

Conception de l’espace administrateur

Conception de la partie administration

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

J’ai fait le choix d’utiliser VueJs pour me permettre une montée en compétences sur ce framework. Ce choix peut paraitre incohérent avec le fait d’utiliser react native sur le front de l’application mobile mais étant en formation j’ai préféré faire ce choix pour en apprendre le plus possible sur les frameworks/ bibliothèques les plus utilisés dans le monde du développement.

J’ai commencé par créer une user story afin d’organiser mon travail.

User Story

En tant qu'administrateur du site web, il pourra accéder à un panel administrateur sécurisé en Vue.js pour gérer les utilisateurs et les contenus du site.

Pour cela, l'accès au panel administrateur sera protégé par une page de connexion. Seules les personnes disposant des droits d'administrateur pourront se connecter avec un nom d'utilisateur et un mot de passe valides.

Une fois connecté, le panel administrateur affiche une vue d'ensemble des statistiques du site, comme le nombre d'utilisateurs actifs, le nombre d’utilisateurs inscrits dont les cinq derniers inscrits et les 5 derniers

messages envoyés. Depuis le panel admin l’administrateur pourra également gérer les utilisateurs, en modifiant le statut de nouveaux comptes ou en supprimant des comptes existants si nécessaire.

En ce qui concerne la gestion de contenu, il pourra supprimer des messages s’il considère que ceux-ci sont inappropriés.

Le panel administrateur doit être facile à utiliser, avec une interface utilisateur claire et intuitive. L’administrateur est sensé pouvoir effectuer toutes les actions de gestion en quelques clics, sans avoir besoin d'une connaissance technique avancée.

Enfin, le panel administrateur doit être sécurisé et protégé contre les attaques malveillantes.

Choix du langage et framework

Pour réaliser ce projet j’ai utilisé VueJs pour le front, pour le back end, j’utilise mon api développée pour l’application mobile.

Vue.js spécifiquement, sa popularité a considérablement augmenté ces dernières années et il est devenu l'une des bibliothèques les plus populaires pour la création d'interfaces utilisateur. Il est fort probable que Vue.js continue de gagner en popularité et d'être utilisé dans l'industrie du développement Web à l'avenir, en particulier avec l'ajout de nouvelles fonctionnalités et l'amélioration de sa compatibilité avec d'autres technologies.

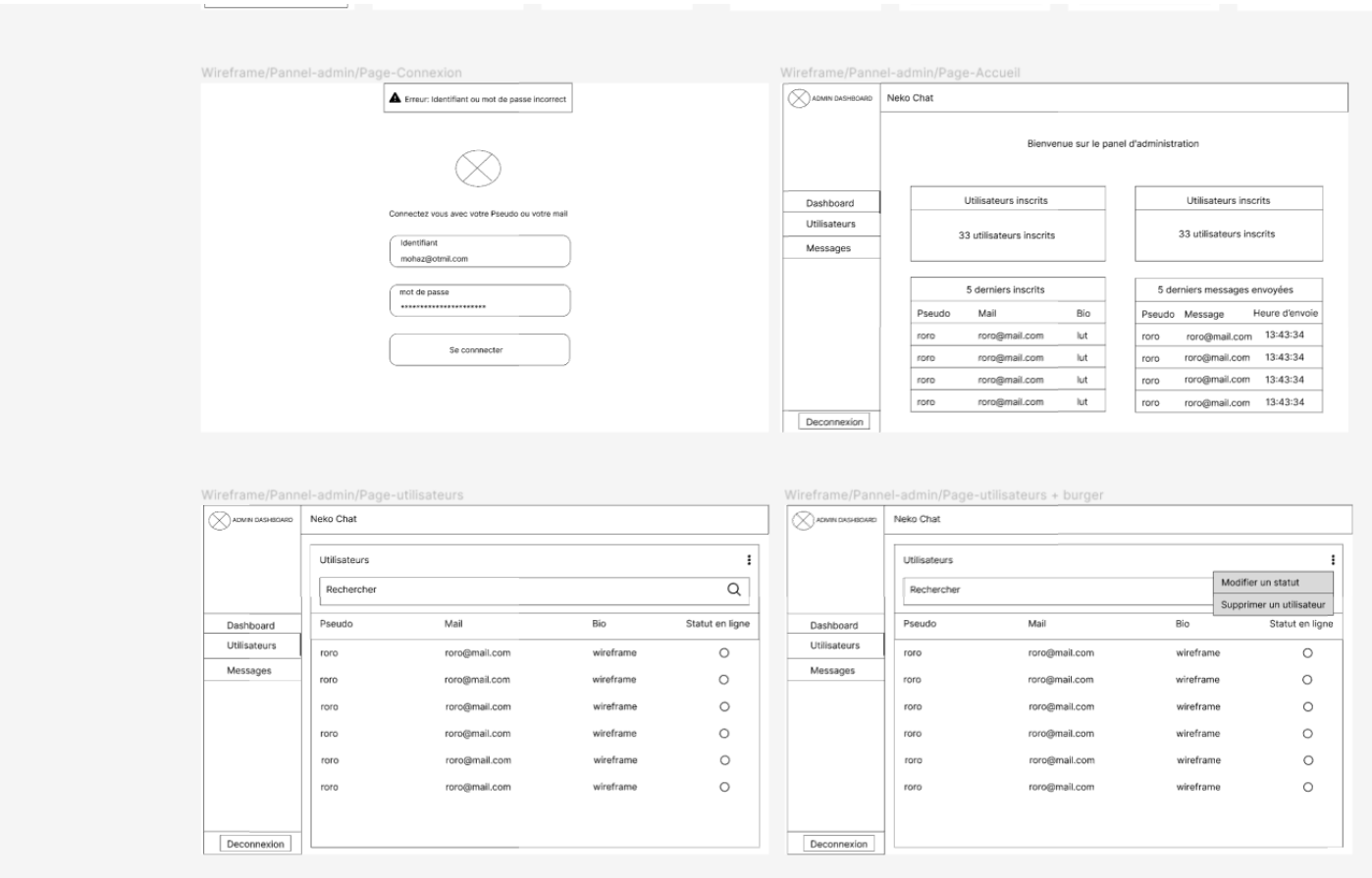
Conception du frontend du site web

Charte graphique

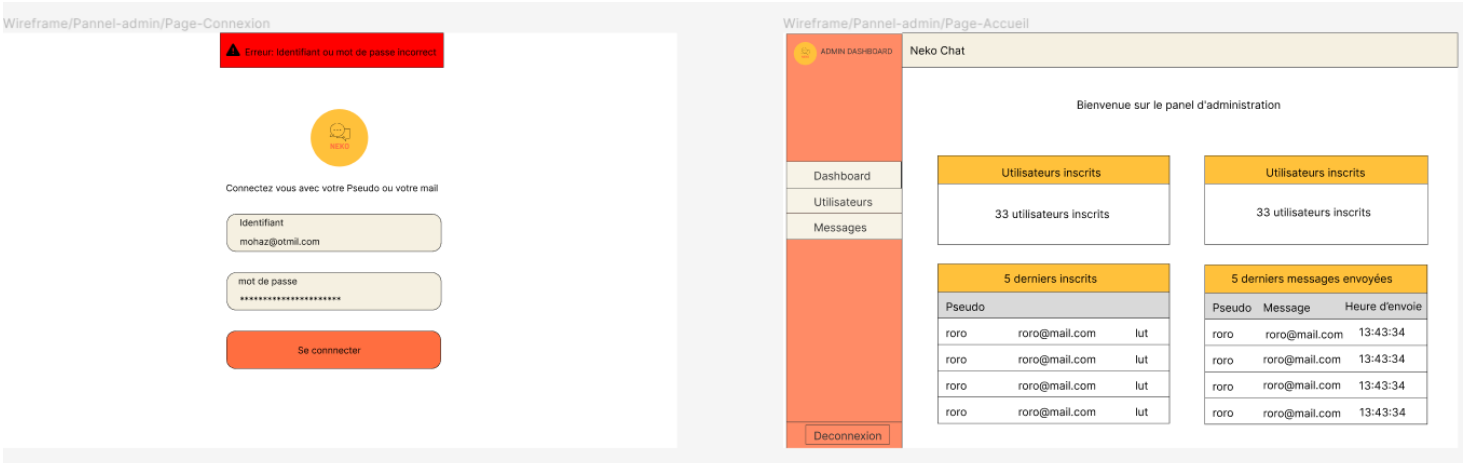
Dans l’optique d'être un cohérent, mon site web utilise la charte graphique de l’application mobile présentée plus haut.

Maquettage

J’ai procédé au maquettage de mon application web de la même manière que pour mon application mobile.



Maquette wireframe



Maquette design

Conception du back end du site web

Pour la partie back de ce site, j'utilise la même API que l'application mobile.

Conclusion

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

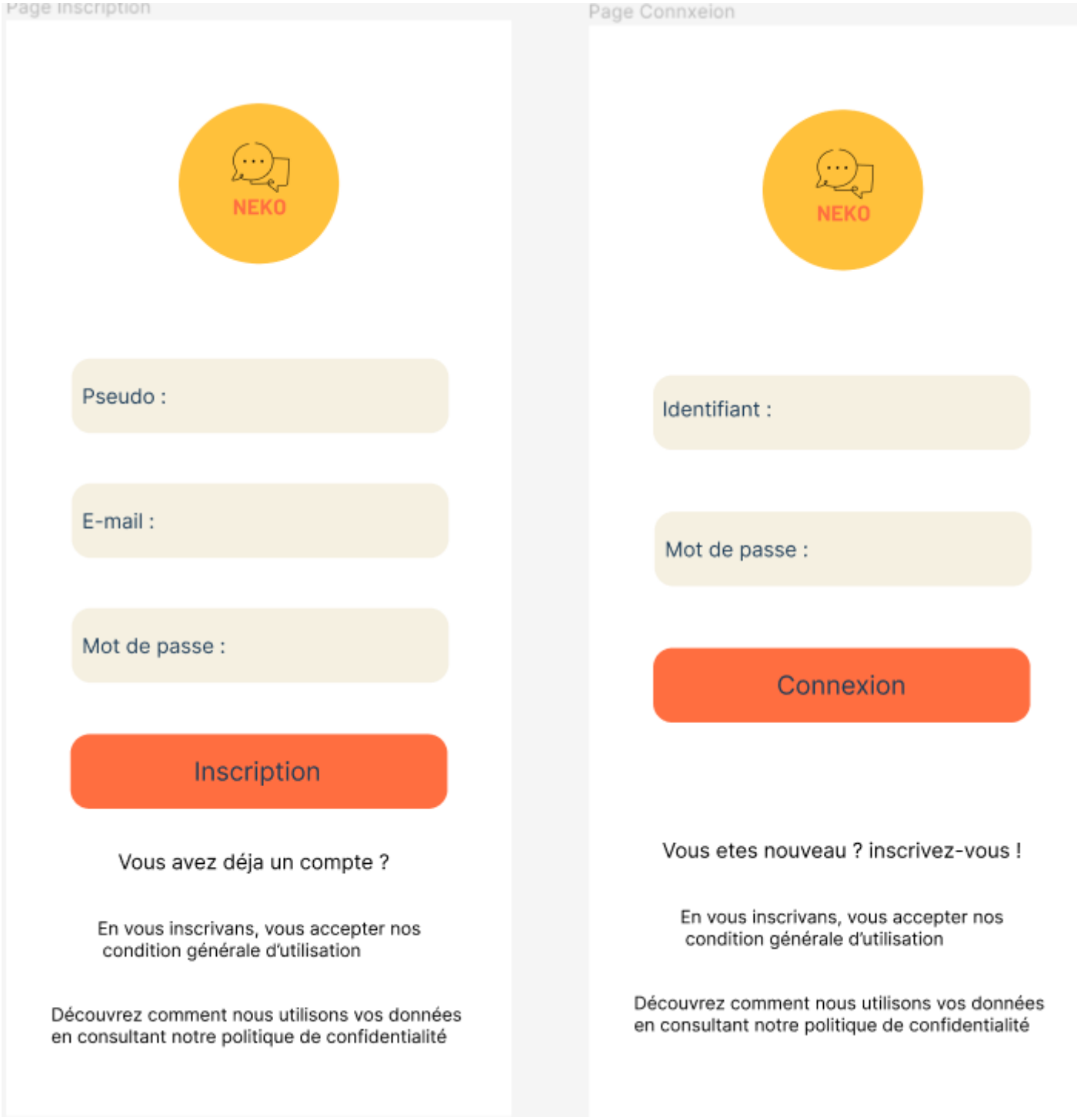
Ce projet qui s’est déroulé sur l'année de formation m’a appris à organiser mon travail, savoir collaborer avec une équipe de développeur, savoir résoudre des problèmes techniques complexes et faire face à des contraintes de temps.

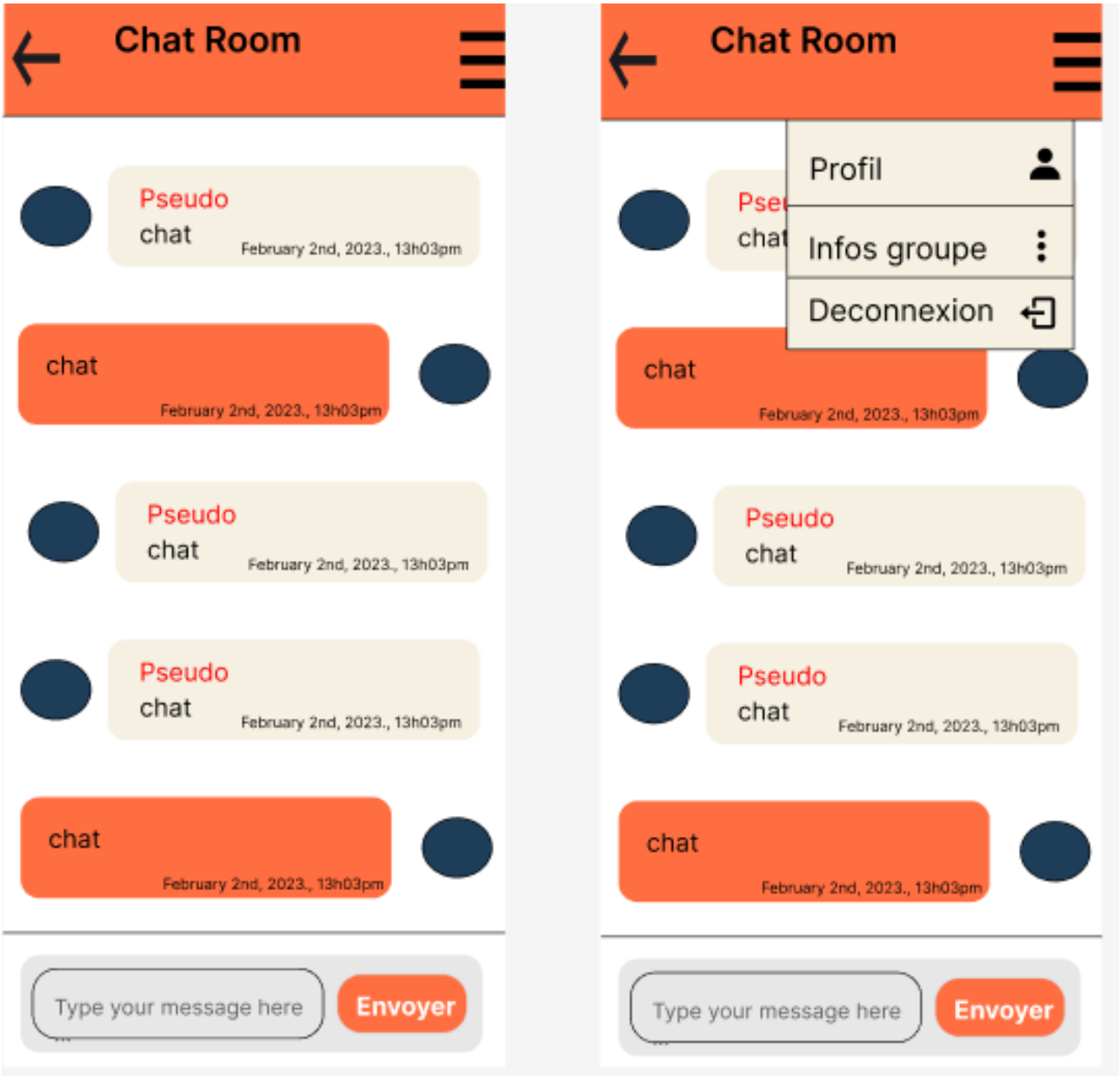
Au cours de ce projet, j’ai travaillé sur de nombreux aspects, tels que la conception, le développement, l'intégration de l'application mobile en React Native et de l'API en Node.js. J’ai également développé un back-office en Vue.js pour la gestion des données de l'application.

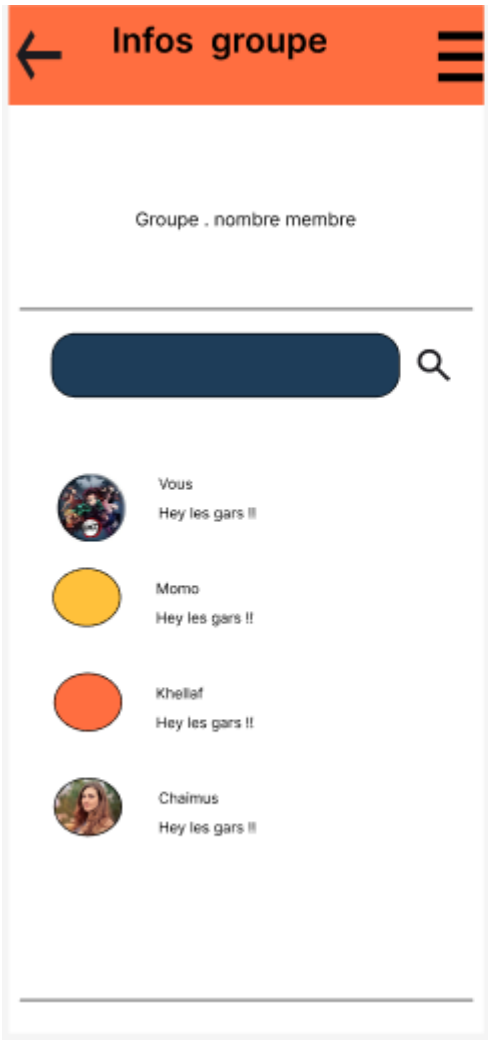
En somme, ce projet m’a permis d'acquérir des compétences précieuses en développement d'applications web et mobiles, en gestion de projet et en travail d'équipe. Vous êtes maintenant mieux équipé pour relever les défis de projets similaires à l'avenir et pour continuer à développer vos compétences dans ce domaine en constante évolution.

ANNEXE

Annexe 1 : Maquette de l'application mobile




Chat Room
Pseudo
chat
February 2nd, 2023., 13h03pm
chat
February 2nd, 2023., 13h03pm
Pseudo
chat
February 2nd, 2023., 13h03pm
Pseudo
chat
February 2nd, 2023., 13h03pm
chat
February 2nd, 2023., 13h03pm
Type your message here
Envoyer
Chat Room
Profil
Infos groupe
Deconnexion
chat
February 2nd, 2023., 13h03pm
Pseudo
chat
February 2nd, 2023., 13h03pm
Pseudo
chat
February 2nd, 2023., 13h03pm
chat
February 2nd, 2023., 13h03pm
Type your message here
Envoyer


Infos groupe
Groupe . nombre membre
Vous
Hey les gars !!
Momo
Hey les gars !!
Khellaf
Hey les gars !!
Chaimus
Hey les gars !!


Infos utilisateur
Information de rorocastagne
E-mail
E-mail@gmail.com
Pseudo
roro-castagne
Actu
Je suis trop beau