



Open Speech

Dossier de présentation pour le titre :
Concepteur développeur d'applications

Présenté par **Machin Max**

Sommaire

1 - Liste des compétences du référentiel couvertes par le projet	4
2 - Résumé du projet	5
3 - Analyse et cahier des charges	6
3.1 - Analyse de l'existant	6
3.2 - Besoins et contraintes	6
3.2.1 - contraintes fonctionnelles : App mobile	7
3.2.2 - contraintes fonctionnelles : Web app (admin)	7
3.3 - Les utilisateurs de l'application	7
3.4 - Contexte du projet	8
4 - Organisation et choix technologiques	8
4.1 - Langages, frameworks et outils utilisés	8
4.1.1 - Langages de programmation	8
4.1.2 - Frameworks	9
4.1.3 - Autres logiciels et outils	10
4.2 - Organisation du projet	11
4.3 - Architecture logicielle	13
5 - Conception : Front-end	14
5.1 - Cas d'utilisation et arborescence	14
5.2 - Identité visuelle	16
6 - Conception : Back-end	19
6.1 - Règles de données	19
6.2 - Modèles de données	20
6.2.1 - Modèle conceptuel de données	20
6.2.2 - Modèle logique de données	20
6.2.3 - Modèle physique de données	21
7 - Développement : Back-end	22
7.1 - Organisation et fonctionnement	22
7.2 - Routes	24
7.3 - Controllers	25
7.4 - Middlewares	26
7.5 - Sécurité	26
7.5.1 - Exemple de failles	26
7.5.2 - JSON Web Token	28
7.5.3 - Gestion des rôles	29
7.5.4 - Quelques outils	29
7.5 - Situation : upload de données avec fichier	30
7.6 - Swagger UI : Documentation de l'API	32
7.7 - Phase de tests	33
8 - Développement : Front-end	34
8.1 - Organisation et arborescence	34
8.2 - Exemple de composants génériques	36

8.3 - Mode sombre	39
8.4 - Navigation	39
8.5 - Persistance de données	40
8.6 - Traduction	42
9 - Espace administration	45
9.2 - Organisation et arborescence	45
10 - Problématiques rencontrées	47
11 - Recherche anglophone	48
12 - Compétences et axe d'amélioration	49

1 - Liste des compétences du référentiel couvertes par le projet

N° Fiche AT	Activités types	N° Fiche CP	Compétences professionnelles
1	Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité	1	Maquetter une application
		2	Développer une interface utilisateur de type desktop
		3	Développer des composants d'accès aux données
		4	Développer la partie front-end d'une interface utilisateur web
		5	Développer la partie back-end d'une interface utilisateur web
2	Concevoir et développer la persistance des données en intégrant les recommandations de sécurité	6	Concevoir une base de données
		7	Mettre en place une base de données
		8	Développer des composants dans le langage d'une base de données
3	Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité	9	Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
		10	Concevoir une application
		11	Développer des composants métier
		12	Construire une application organisée en couches
		13	Développer une application mobile
		14	Préparer et exécuter les plans de tests d'une application
		15	Préparer et exécuter le déploiement d'une application

2 - Résumé du projet

Il s'agit d'un projet d'application mobile et web réalisé dans le cadre de ma deuxième année de formation à La Plateforme_.

Selon la demande du client, le projet devait comprendre au minimum :

- Un système d'authentification
- Un système de chat général
- Site administratif
- Back-end : Node Js ou API Platform
- Front-end: React Native

Ce projet a été réalisé en groupe avec deux collaborateurs de ma promotion : **André Grassi & Mourad Boussiouf**

Il a pris en charge une grande partie des étapes de conceptualisation d'un véritable projet professionnel telles que :

- une arborescence
- conception graphique
- modèles de données
- une organisation / répartition du travail.

Pour développer la partie **front**, nous avons créé une application **Expo React Native**, utilisant principalement **CSS** pour le style.

Le **back-end** se compose d'une **API REST Node.js**.

Pour la partie **administration**, nous avons implémenté une application **web React.js**, utilisant **Sass** pour le style.

Nous avons créé une **base de données MySQL**.

Ciblant les utilisateurs d'Android et tous n'ont pas de téléphones Android, nous avons également utilisé **Android Studio** qui nous a permis de développer l'application sur un appareil similaire émulé pour tous les membres du groupe.

Partant d'un projet en milieu scolaire et sans véritable client, il a donc fallu penser à créer une entreprise fictive : Identité visuelle et commerciale.

Cela passe par la création d'une **charte graphique** et son positionnement sur le marché. Des fonctionnalités supplémentaires ont été ajoutées afin d'avoir un rendu final plus pertinent, d'acquérir plus de capacités et d'avoir un site plus fonctionnel.

Parmi les **fonctionnalités supplémentaires** que nous avons ajoutées :

- La possibilité de définir un code PIN pour sécuriser l'application.
- Téléchargement de l'avatar à la création du compte.
- Choix de la langue de traduction de l'application..
- Conversations privées.
- Un carnet de contact avec des invitations d'amis.

3 - Analyse et cahier des charges

3.1 - Analyse de l'existant

S'agissant d'une application mobile de messagerie, nous avons commencé par analyser le marché des applications telles que *Whatsapp*, ou encore *Messenger*.

C'est à partir de cette analyse que nous avons décidé de chercher une idée qui nous permettrait de nous différencier de la "concurrence".

Après plusieurs pistes de réflexion, une est ressortie plus particulièrement : La traduction de message instantanée.

En effet, aucune application de messagerie ne propose de telle fonctionnalité, qui pourrait d'ailleurs apporter un plus contre la barrière de la langue.

3.2 - Besoins et contraintes

Plusieurs contraintes étaient définies dans le cahier des charges notamment concernant le délai de livraison mais aussi au niveau technique et fonctionnel.

Contraintes de délai : ~ 30 jours.

Pour la partie technique nous avions le choix entre deux différentes technologies pour le développement du back-end, React Native étant imposé pour la création de l'application mobile. Nous avions le libre choix pour la base de données.

Contraintes techniques :

- front : React Native
- Back : Node Js / Express ou API Platform / Symfony
- Base de données : choix libre

Un certain nombre de routes API étaient demandées dans le cahier des charges, séparé entre l'application mobile et le panel administrateur ou Web app.

3.2.1 - contraintes fonctionnelles : App mobile

Fonctionnalités attendues :

- Des routes permettant l'authentification de l'utilisateur (connexion, inscription ...)
- Une route permettant de voir et mettre à jour son profil
- Une route permettant d'envoyer un message dans le chat général
- Une route permettant d'afficher la totalité des messages du chat général
- Une route listant tous les utilisateurs de l'application

3.2.2 - contraintes fonctionnelles : Web app (admin)

Fonctionnalités attendues :

- Des routes permettant la modération des utilisateurs (suppression, édition)
- Différentes routes de modération du site (suppression de messages, édition ..)

L'entreprise étant créée dans le cadre du projet, nous ne disposons pas vraiment de contraintes visuels (logo, charte graphique).

L'identité visuelle de l'entreprise a été développée durant la phase de conception 'front'.

Contraintes visuelles :

- Rendu final équivalent aux maquettes graphiques établies

3.3 - Les utilisateurs de l'application

Le projet se décompose en deux parties en fonction du rôle :

- L'application mobile React native permettant aux utilisateurs dit 'lambda' de s'inscrire et pouvoir profiter des différentes fonctionnalités proposées.
Ils auront la possibilité d'ajouter des contacts, choisir une langue de traduction et échanger avec des personnes à travers le monde.
- La Web App React Js permettant aux administrateurs et administrateurs seulement, d'avoir une vision globale des données du site et pouvoir gérer les différentes parties de l'app (users, messages, contacts)

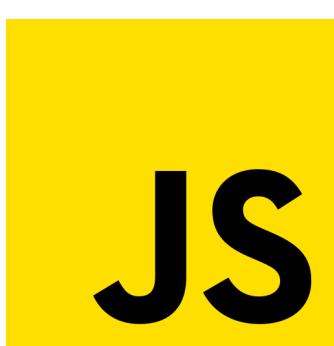
3.4 - Contexte du projet

Le projet est réalisé en groupe avec deux autres collaborateurs de ma promotion, mettant en avant le travail et l'organisation de groupe.
De plus, il est réalisé en "autonomie" nous obligeant à faire de nombreuses recherches et à nous documenter autant sur la partie technique que conceptuelle du projet. (avec le soutien des équipes pédagogiques)

4 - Organisation et choix technologiques

4.1 - Langages, frameworks et outils utilisés

4.1.1 - Langages de programmation



Pour la réalisation de ce projet, nous avons décidé d'utiliser uniquement Javascript avec Node Js comme environnement de développement.

Javascript est utilisable aussi bien côté serveur que côté client ce qui permet une certaine unicité de langage entre la partie front-end et back-end de l'application.

De plus, ce choix se justifie de part le fait que notre équipe est essentiellement plus à l'aise sur cette technologie.

Il est également très utilisé sur le Web et dispose donc d'un grand nombre de forums et documentations portant sur d'éventuelles problématiques que nous pourrions rencontrer.

Le front-end de l'application pourra être "dynamique".



Dans l'intention d'optimiser notre temps de production, nous avons décidé d'utiliser **SASS** pour le développement du front-end de la partie **administration** (Web app).

SASS est l'acronyme de **Syntactically Awesome Style Sheet**. C'est un **préprocesseur**¹ qui ajoute des fonctionnalités à CSS. Il permet, entre autres, de mieux structurer et simplifier le code, d'éviter les répétitions et plus encore. Le code SASS doit être dans des fichiers ayant l'extension **.scss**. À l'aide d'un programme, les fichiers SASS sont compilés et convertis en CSS. Une fois compilé, tout le code est regroupé dans un seul fichier CSS et **minifié**.

4.1.2 - Frameworks

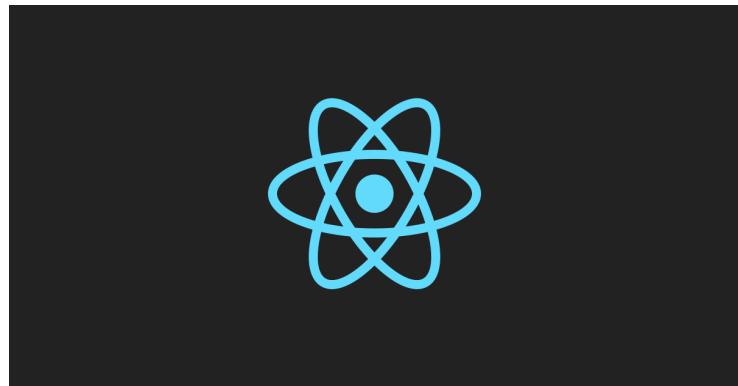
express

Express est une infrastructure d'applications Web **Node.js** minimaliste et flexible qui fournit un ensemble de fonctionnalités robuste pour les applications Web et mobiles.

Le framework disposant de multiples méthodes **HTTP** et de **middleware**, il permet la création d'API robuste, simple et rapide.

Ce qui explique notre choix, ne disposant que d'un court délai pour la réalisation du projet, la simplicité et la rapidité de développement mis à disposition par Express correspondait parfaitement aux contraintes de délai.

¹ Un préprocesseur CSS est un outil informatique permettant de générer dynamiquement des fichiers CSS. L'objectif est d'améliorer l'écriture de ces fichiers, en apportant plus de flexibilité au développeur web.



React est une bibliothèque **JavaScript** libre développée par Facebook permettant de faciliter la création d'**applications Web et Mobile** (**React native**) à l'aide de **composants** et d'**états** permettant d'afficher / rafraîchir une page ou du contenu **HTML** à chaque changement.

Nous avons donc choisi d'utiliser **React Js** pour le panel administrateur. Ce choix s'explique par l'interface de développement et le langage de codage optimisés de la bibliothèque permettant ainsi un développement plus rapide de l'application Web.

Pour le développement de la partie mobile, nous avons donc utilisé **React Native**. Il permet de développer et déployer simultanément votre application pour les plateformes **iOS** et **Android** avec une base de code unique. Il permet d'optimiser la vitesse de développement notamment grâce à sa facilité d'utilisation et sa librairie de composants concise.

Il dispose de beaucoup de documentations / ressources dues à une communauté active et nombreuse.

4.1.3 - Autres logiciels et outils

Durant toute la durée du développement nous avons été amené à travailler avec de nombreux logiciels / outils :

- **Git / GitHub** pour le versionning
- **Trello** pour l'organisation et la répartition des tâches
- **Figma** pour la création des maquettes graphiques
- **LucidChart** pour la mise en place des modèles de données
- **Postman** pour les test API
- **Npm** pour la gestion de packages
- **Expo** pour émuler l'application sur les différents device
- **Android Studio** pour ouvrir l'application sur le même 'type' de device depuis différents ordinateurs
- **Canva** pour la création du Logo
- **Coloris** pour la charte graphique

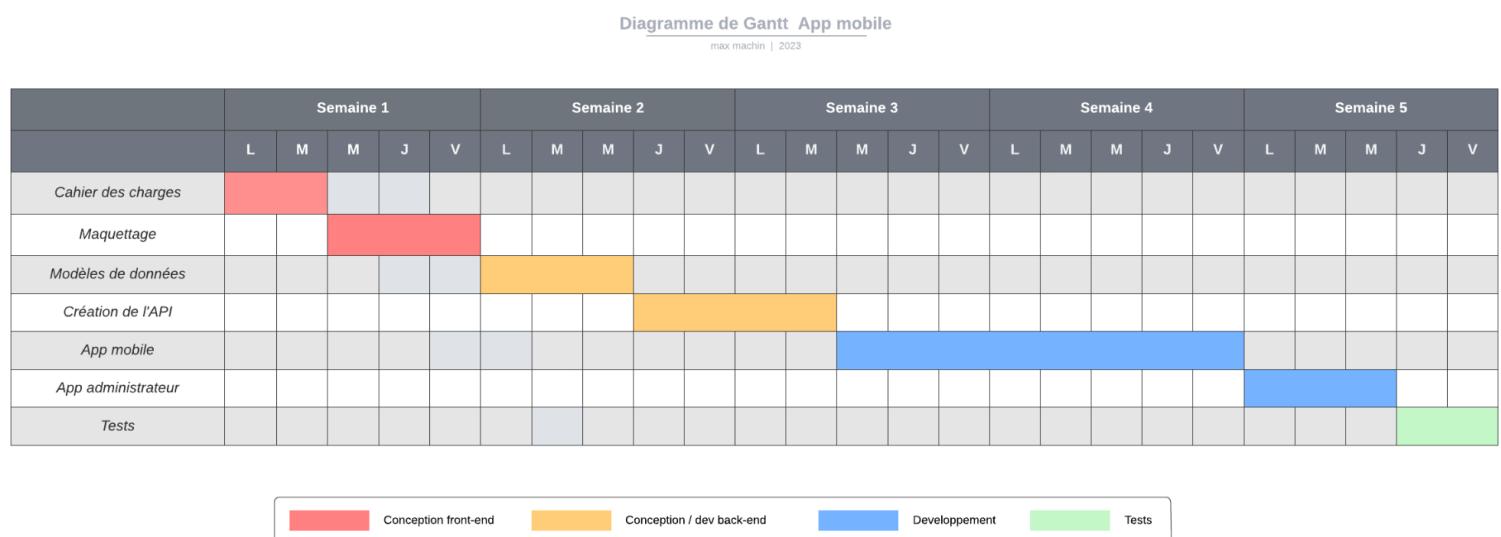
4.2 - Organisation du projet

Ce projet a été réalisé dans le cadre de mon année de formation, impliquant une charge de travail importante entre les projets développés durant mes semaines d'alternance et ceux à rendre dans le cadre scolaire.

Il était donc nécessaire d'avoir une **organisation rigoureuse** pour parvenir à développer une application de cette envergure. Pour cela nous avons décidé d'adopter une méthode de gestion de projet dite **Agile**².

Nous avons donc commencé par lister les différentes étapes de développement du projet dans un **diagramme de Gantt**.

Ce diagramme nous a permis de mieux visualiser le travail selon les contraintes de délai imparties.



Toujours dans le but de maintenir une bonne organisation, nous avons mis en place un tableau **"Kanban"** à l'aide de l'outil **Trello**.

Cela nous a permis de mieux répartir le travail entre les différents collaborateurs à l'aide de tickets.

Ces tickets sont rangés par "étape de développement" : conception, dev, back, front etc...

Cette méthode permet notamment d'avoir une vision globale sur l'avancement du projet, mais aussi sur les différents soucis rencontrés.

Sur ce tableau nous partagions également des fichiers / ressources tout au long du projet.

² Méthodes agiles :

Concernant le repository **Github**, nous avons mis en place un **workflow³** permettant de maintenir l'organisation jusqu'au versionning de l'application.

Premièrement, nous ne faisions pas de push depuis la branche dite 'finale' ou main. Chaque collaborateur travaillant sur une fonctionnalité précise doit créer un branche relative à cette fonctionnalité / ticket en question. Puis un **merge** de sa branche sera effectué vers la branche main un fois le code fonctionnel. Cette méthode permet notamment de maintenir une version fonctionnelle de l'application à n'importe quel moment du développement (pratique pour des démonstrations par exemple). Mais également de ne push le code que par bloc plus concis et donc minimiser les conflits tout en facilitant le débug. Les **commits** doivent contenir des messages pertinents permettant de revenir à une version précise plus simplement et maintenir un historique de l'avancement.

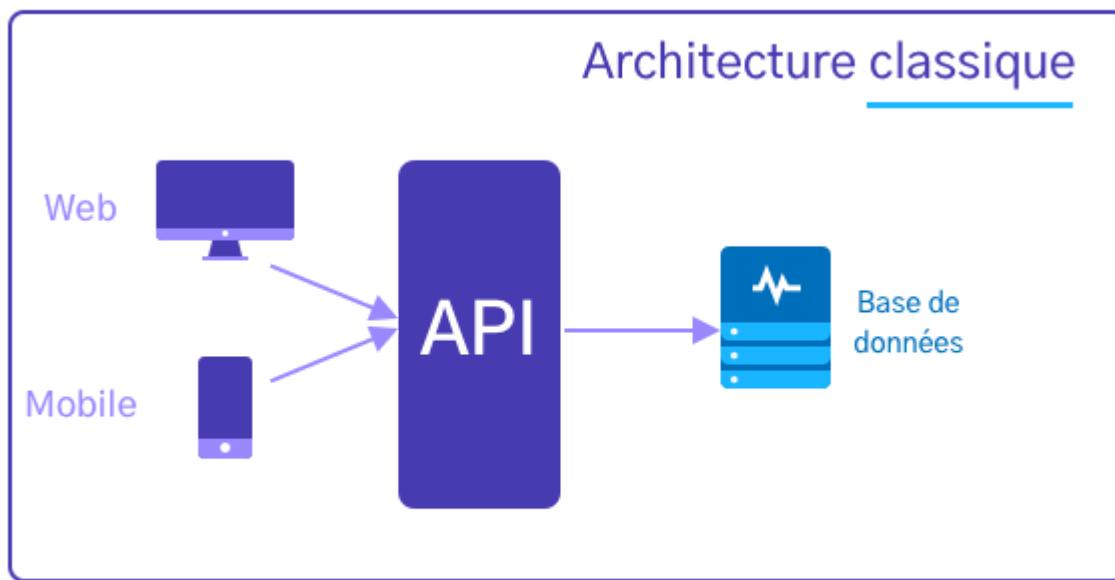


³ automatisation des différentes opérations et étapes de validation d'une tâche plus ou moins complexe.

4.3 - Architecture logicielle

Concernant l'architecture logicielle, nous avons opté pour une architecture **multicouche ou n-tiers**.

Plus précisément **3-tiers** dans notre cas.



On peut remarquer les trois couches de l'architecture sur le schéma ci-dessus :

- **La couche de présentation** correspond à l'affichage, la restitution sur le poste de travail
- **La couche de traitement métier** correspond à la mise en œuvre de l'ensemble des règles de gestion et de la logique applicative
- **La couche d'accès aux données** correspondant aux données qui sont destinées à être conservées / récupérées

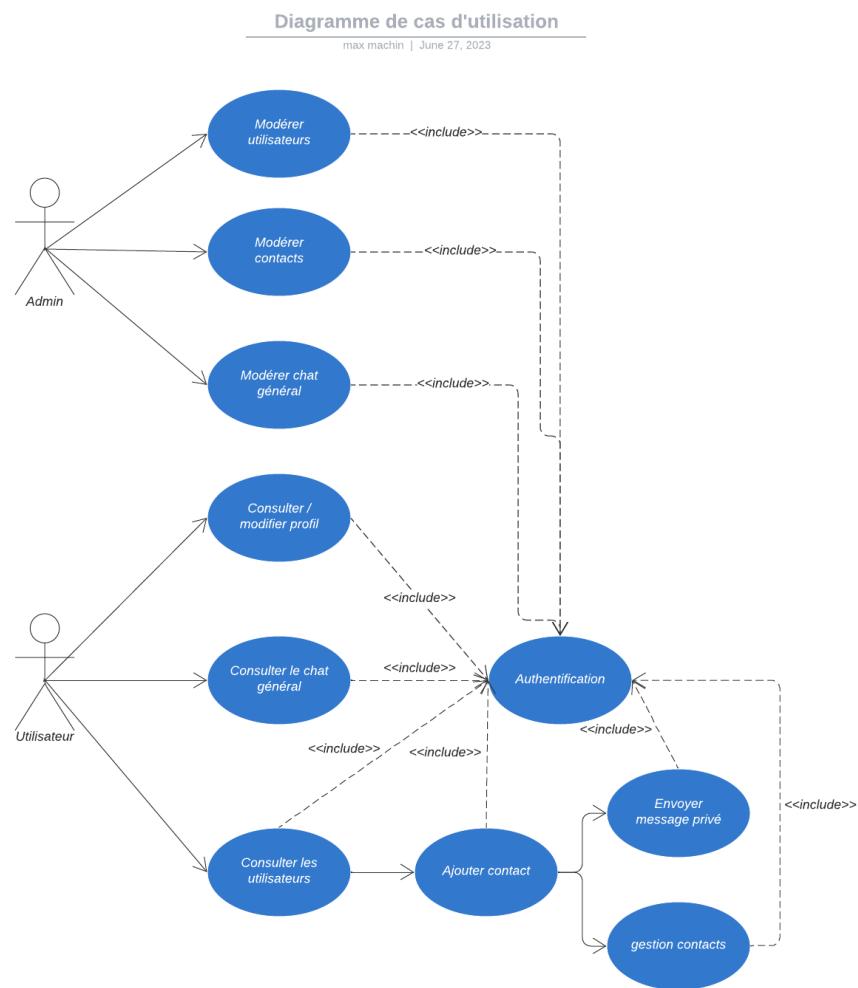
Dans notre projet, l'application mobile et la Web app communiquent avec la même API qui elle interagit avec notre base de données.

5 - Conception : Front-end

5.1 - Cas d'utilisation et arborescence

Pour commencer la **conception** de notre application, nous avons établi un **diagramme de cas d'utilisation** nous permettant de visualiser les différentes fonctionnalités attendues ainsi que leurs contraintes (authentification).

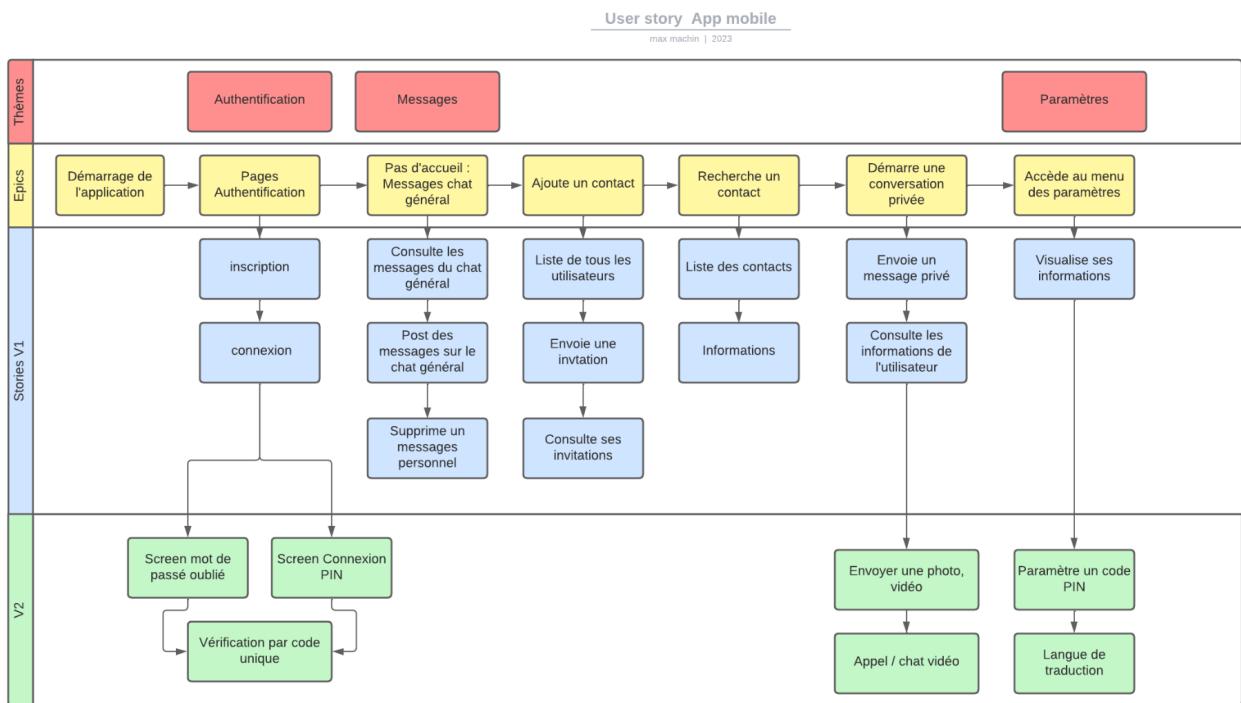
Nous les avons réparties entre les différents acteurs de l'application : les **utilisateurs** et les **administrateurs**.



A l'aide du diagramme établi précédemment, nous avons poursuivi la conception de l'application.

Nous avons donc mis en place une "**User story**".

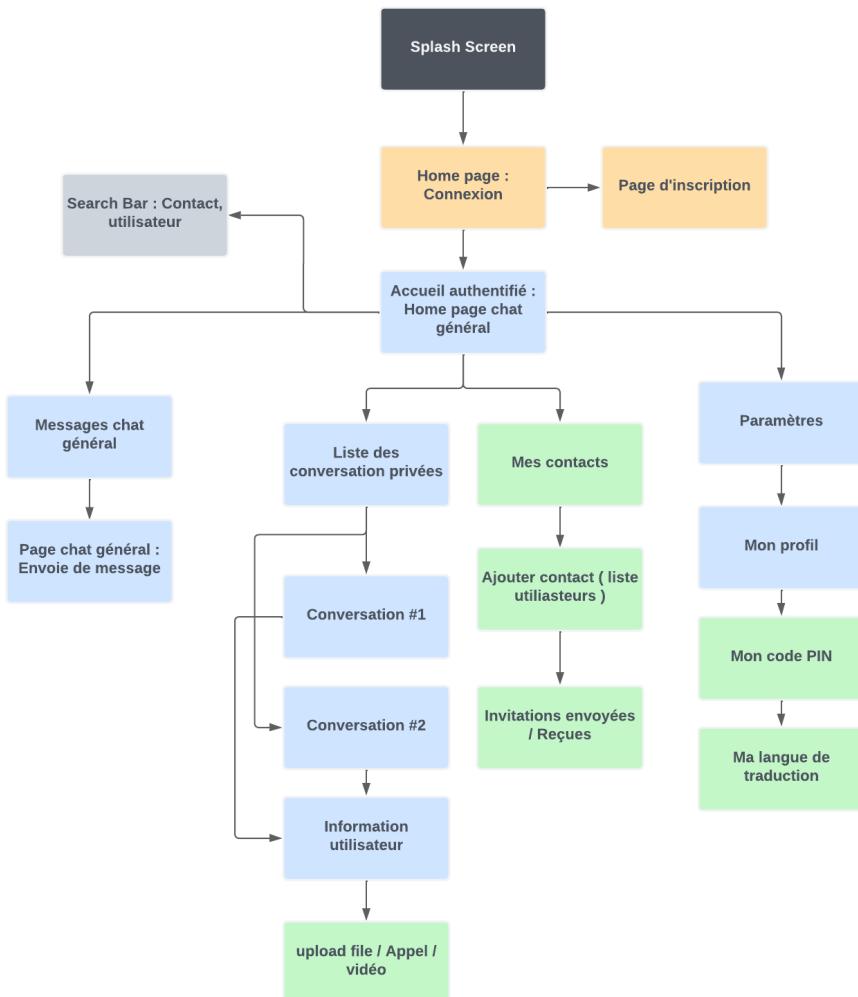
Dossier de Projet Professionnel - Machin Max



Sur ce schéma, on peut voir les différentes **actions** de l'utilisateur en fonction de son **avancée** dans l'application.

Il nous a permis de mieux visualiser et définir les screens et différents éléments qui devront être présents dans l'application mobile en fonction du parcours utilisateur.

Ce premier schéma m'a donc permis de mettre en place **l'arborescence** de l'application React native.



5.2 - Identité visuelle

La **charte graphique** représente une étape cruciale lors de la création d'un projet, elle a pour but de représenter visuellement une entreprise.

Elle comprend plusieurs éléments notamment :

- Le logo
- Le thème couleur / palette
- Les typographies

Elle est généralement fournie avec le cahier des charges, il faut donc porter une attention particulière à la respecter..

Ne disposant pas de charte graphique / identité visuelle, nous l'avons donc créé dans le but d'obtenir un résultat final des plus cohérents et respectant les manières de faire professionnelles.

Nous avons commencé par définir une palette de couleurs à l'aide de l'outil en ligne **Coolors**, avant de créer un logo sur **Canva**.

Suite à cela, nous avons choisi les **typographies**.

Enfin, grâce à l'outil collaboratif **Figma** nous avons rassemblé ces éléments dans le but de pouvoir les utiliser rapidement pendant le maquettage. Nous avons également pensé en amont les éléments qui seraient amenés à apparaître le plus souvent dans l'application afin d'obtenir un résultat visuel plus harmonieux et gagner en rapidité de production.



Eléments de base	Inputs	Nav
Submit 	Adresse e-mail 	
	Confirmez mot de passe 	
	Checkbox / PIN <input type="checkbox"/> J'accepte les termes et conditions	
		Icons

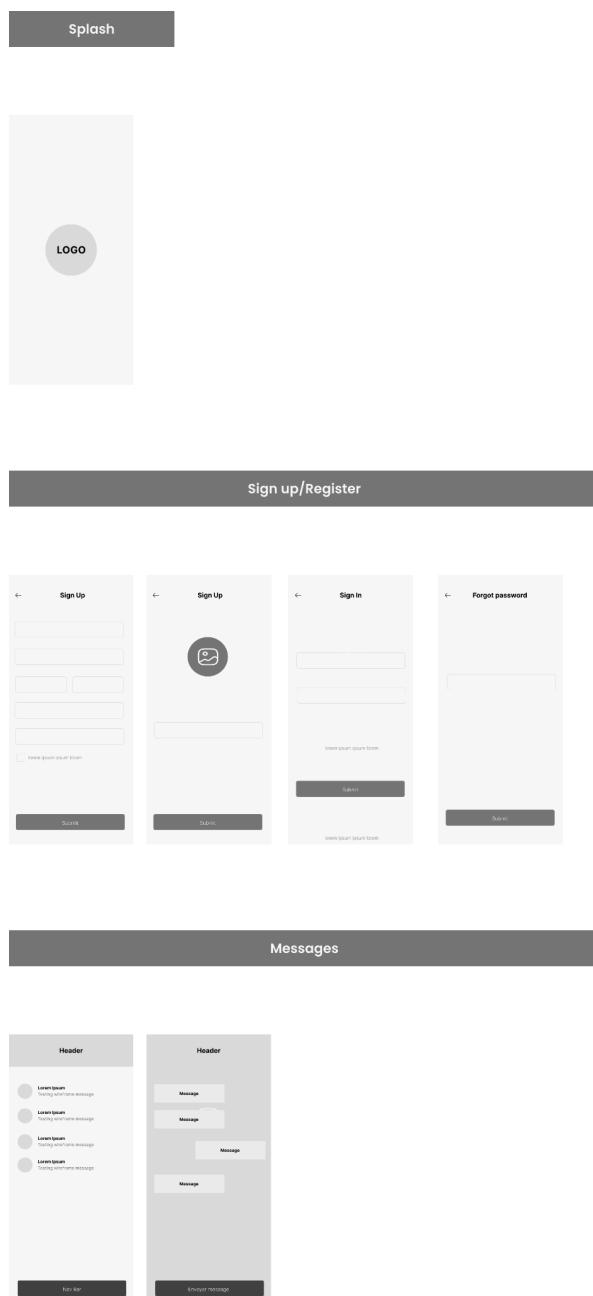
5.3 - Maquettes de l'application

Grâce aux diagrammes / arborescence en place, nous avons pu maquetter visuellement les différents screens répertoriés.

Ces maquettes ont été réalisées à l'aide de **Figma**.

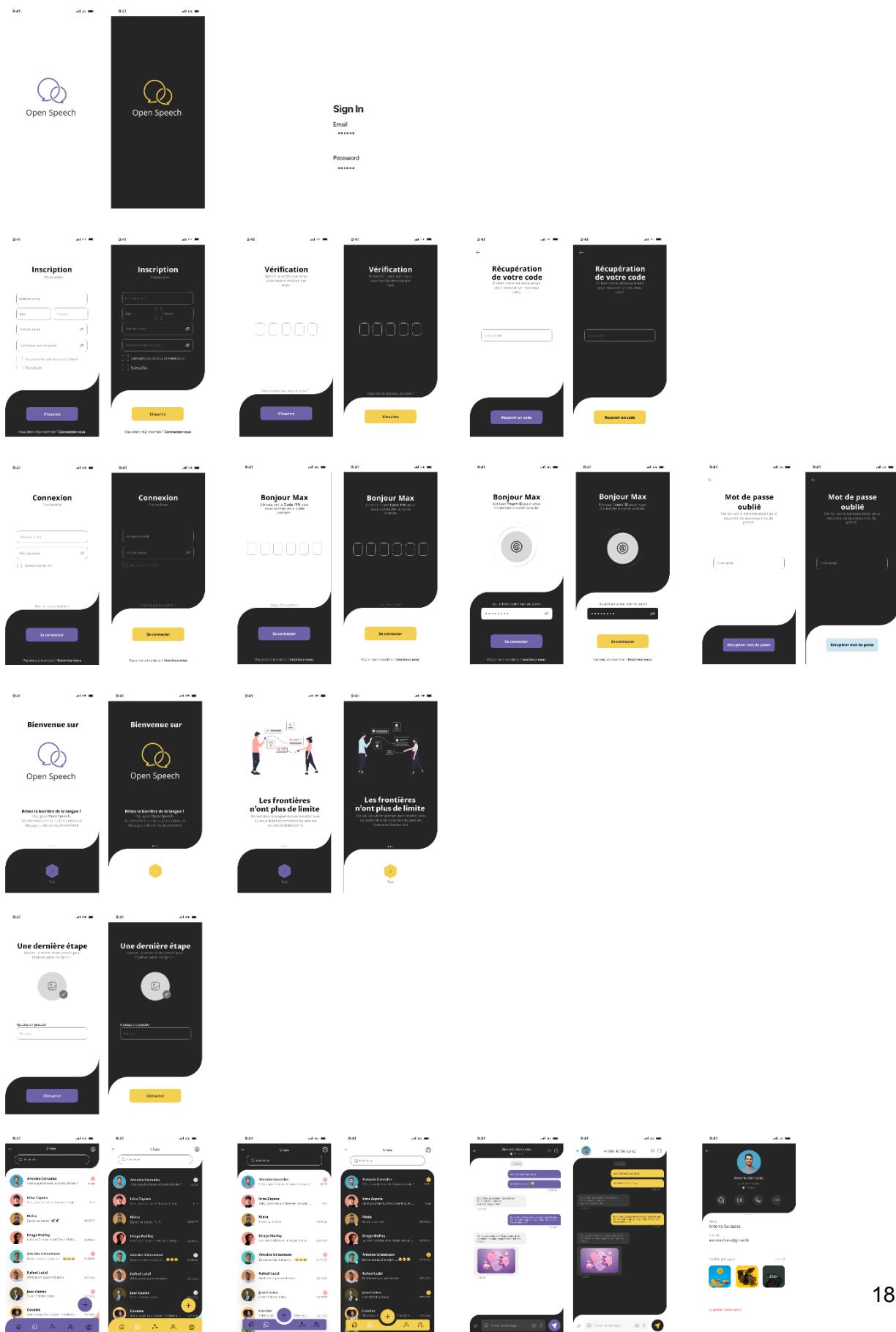
Dans un premier temps le **Zoning** "ou low fidelity" a été créé. Il s'agit de la mise en place et le positionnement des différents blocs visuels de chaque page.

Nous avons préféré rester simple sur cette partie du maquettage pour pouvoir accorder plus de temps à la **maquette** "high Fidelity"



Dossier de Projet Professionnel - Machin Max

Nous avons ensuite, à l'aide de la charte graphique et du zoning poursuivis par le **Wireframe**. Le wireframe se rapproche du rendu final du visuel. Il intègre donc l'identité visuelle, les photos, les textes, et tous les éléments dans leur style final.



6 - Conception : Back-end

6.1 - Règles de données

Après de nombreuses recherches concernant les bases de données des applications de messagerie, nous avons remarqué que le **NoSQL** était le plus souvent utilisé avec NodeJS. N'ayant pas de contraintes à ce niveau dans le cahier des charges, nous avons décidé de partir sur une **base de données relationnelle MySQL**.

Ce choix s'explique par le fait que l'équipe a été formée sur ces bases de données et que nous souhaitions développer nos compétences / connaissances sur le sujet.

Pour la conception de la base de données, nous avons utilisé la méthode **Merise**⁴.

Afin de pouvoir construire nos modèles de données, nous avons recueilli les données pertinentes afin de préciser notre '**dictionnaire des données**' :

- Pour les utilisateurs : informations personnelles permettant l'identification mais aussi la présentation du profil. Nom, prénom, email, password, avatar, pseudo, date d'inscription, role
- Pour les messages : Contenu du message, date d'envoi mais également pseudo de l'utilisateur l'envoyant.
- Pour les contacts : les identifiants des deux utilisateurs en contact.
- Pour les rôles : Un nom de rôle

Cela nous a notamment permis de mettre en place les **règles de données** qui sont :

- Un utilisateur peut poster aucun ou plusieurs messages.
- Un message n'est posté que par un utilisateur.
- Un utilisateur peut ajouter aucun ou plusieurs contacts.
- Un contact peut être créé par un utilisateur.
- Un utilisateur peut créer aucune ou plusieurs conversations privées.
- Une conversation privée peut contenir 1 ou plusieurs messages.

⁴

6.2 - Modèles de données

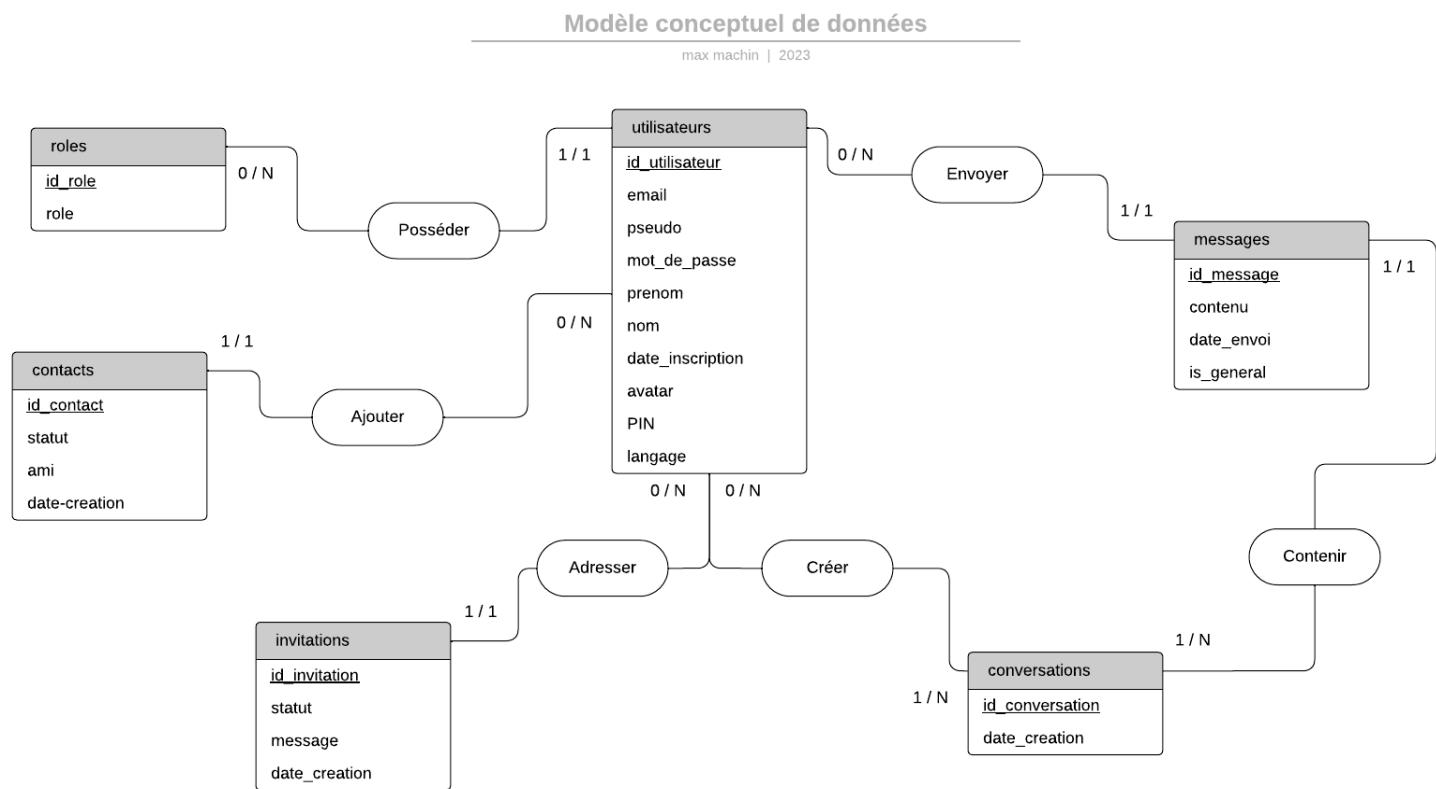
Une fois les différentes règles de données mises en place et les principales données à stocker répertoriées, nous avons pu réaliser la conception des **modèles de données**.

6.2.1 - Modèle conceptuel de données

La première étape consiste à créer le modèle conceptuel de données ou **MCD**.

Il est une représentation claire des données du système d'information à concevoir.

Les données sont représentées sous forme **d'entités** et **d'associations** entre entités.



6.2.2 - Modèle logique de données

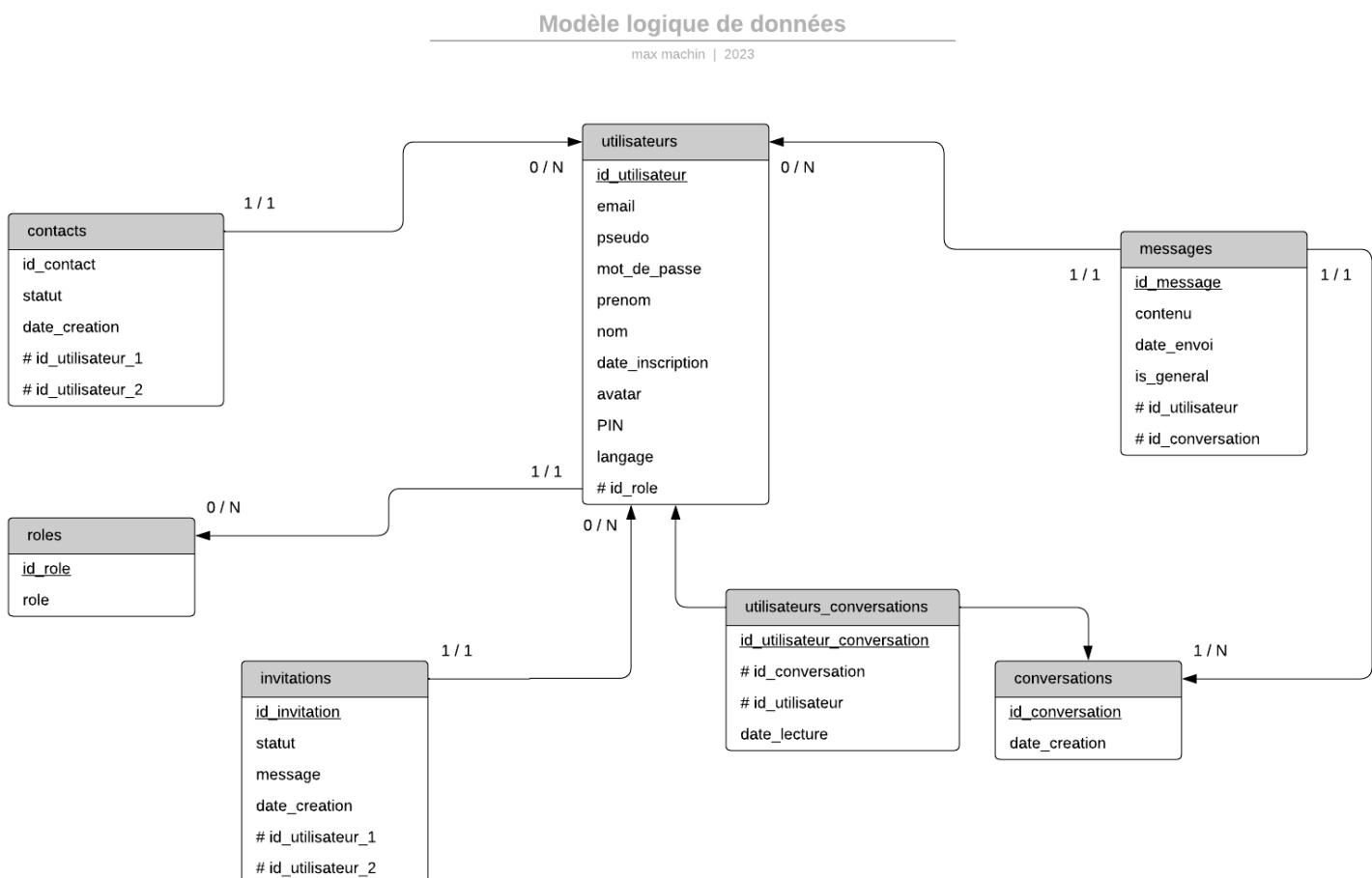
Le **MCD** (Modèle Conceptuel de Données) ne peut pas être implanté dans une base de données sans modification.

Il est obligatoire de transformer ce modèle. On dit qu'on effectue un passage du modèle conceptuel de données vers le **modèle logique de données**.

Le **MLD** pourra être implanté dans une base de données relationnelle.

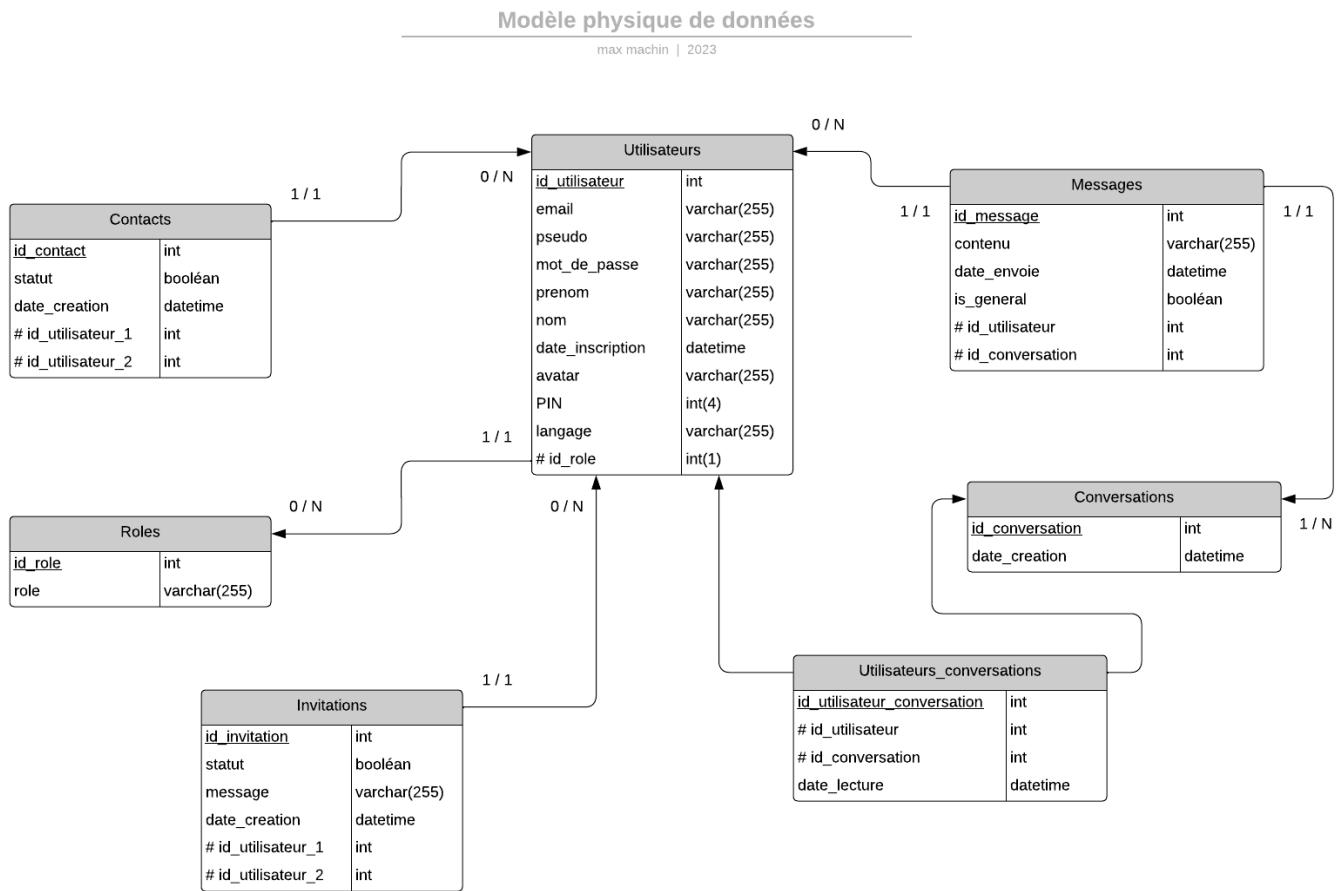
Lors du passage du MCD au MLD on applique **differentes règles** :

- Une entité du MCD deviendra une relation / table
Son identifiant devient la clef primaire de la relation.
Les autres propriétés deviennent des attributs
- Une relation de type 1:N se traduit par la création d'une clef étrangère dans la relation 1.
- Une relation de type N:N se traduit par la création d'une table dont la clef primaire sera composée des deux clefs étrangères.



6.2.3 - Modèle physique de données

Le modèle physique de données contient les informations détaillées nécessaires à la phase de mise en œuvre. Il intègre le typage des données et permet donc de créer un script de **création de la base de données SQL**.



La phase de conception front / back et la réalisation de ces documents nous a permis d'optimiser et réduire notre temps de développement.

Nous pouvions désormais visualiser les différentes classes et leurs attributs à implémenter, ainsi que les éléments visuels et screens définis dans les maquettes et parcours utilisateur. Le but étant de se concentrer uniquement sur le développement du code et de la logique métier.

7 - Développement : Back-end

7.1 - Organisation et fonctionnement

Nous avons mis en place une **API Node Js** qui sert tant pour l'application mobile que pour l'application web pour l'administration afin de ne pas avoir à développer deux API distinctes. Dans le but d'optimiser notre code, nous avons divisé nos programmes en plusieurs modules.

Cette méthodologie a pour but de faciliter la lisibilité du code et le maintient par la suite.

La logique du code est organisé dans différents dossiers -> fichiers :

- **Routes** : contient la totalité des routes de l'API, une route par entité / table en base de données.
- **Controllers** : contient la totalité des controllers, un par route.
- **middlewares** : contient les fichiers middlewares permettant notamment de vérifier qu'un utilisateur est bien connecté ou possède le rôle d'administrateur.
- **services** : contient la logique de connexion à la base de données
- **images** : contiendra les images uploadé par les utilisateurs.
- **tests** : contient les fichiers de test.

Concernant le fonctionnement de l'API, lorsqu'une requête est envoyée par l'utilisateur l'url est analysée par le routeur.

Par rapport à la route et à la fonction indiquée, un controller sera appelé à son tour permettant de faire le lien avec la couche de données.

Par la suite une réponse sous format JSON ainsi qu'un statut seront retournés.

Voici une liste des différents statuts HTTP utilisés :

- **200 : Ok**
La demande a réussi
- **201 : Created**
La demande a réussi et une nouvelle ressource a été créée en conséquence.
- **204 : No content**
Il n'y a pas de contenu à envoyer pour cette requête, mais les en-têtes peuvent être utiles.
- **400 : Bad request**
Le serveur ne peut pas ou ne veut pas traiter la demande en raison de quelque chose qui est perçu comme une erreur du client
- **403 : Forbidden**
Le client n'a pas les droits d'accès au contenu ; c'est-à-dire qu'il n'est pas autorisé, donc le serveur refuse de donner la ressource demandée.
- **500 : Internal server error**
Le serveur a rencontré une situation qu'il ne sait pas gérer.

Ainsi que des différentes méthodes HTTP utilisées :

- **GET** : permet de récupérer des données
- **POST** : permet d'enregistrer des données
- **PUT** : permet de modifier les informations d'une donnée dans leur intégralité
- **PATCH** : permet de modifier les informations d'une donnée partiellement
- **DELETE** : permet la suppression de données

7.2 - Routes

Pour ce qui concerne le **routeur** de l'application, nous nous sommes servis du **middleware** et du système de routage complet proposé par le routeur d'**Express Js**.

Voici comment s'articule une route :

<code>app</code>	<i>Instance of an Express application</i>
<code>app.get</code>	<i>The HTTP request method</i>
<code>"/test"</code>	<i>The URL path for this route</i>
<code>(req, res) => {}</code>	<i>The handler function</i>

Nous avons donc divisé les routes en fonction des tables de la base de données les concernant. Cela nous a permis d'avoir une meilleure lisibilité et un maintien plus facile du code.

```
app.use('/user', usersRouter)
app.use('/conversations', conversationsRouter)
app.use('/admin', adminRouter)
app.use('/invitations', invitationsRouter)
app.use('/messages', messagesRouter)
```

En fonction de la route utilisée, un routeur contenant des sous-chemins ainsi que des middlewares / fonctions associés sera à son tour appelé de part le middleware `.use()`

A l'intérieur des fichiers de routeurs, une instanciation de `express.Router()` est faite.

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

Permettant de faire appel aux différentes **méthodes HTTP** proposées par Express. On peut donc choisir une méthode spécifique pour effectuer une action en base de données.

```
router.post('/register', userController.register);
router.post('/login', userController.login)
router.get("/details/:id", isAuthenticated, userController.getDetails)
router.delete("/deleteContact/:id", isAuthenticated, userController.deleteContact)
```

On peut voir sur l'image ci-dessus différentes méthodes : post, get et delete.

On peut également noter la présence de middleware sur certaines routes nécessitant l'authentification de l'utilisateur. ("/details/:id" , "/deleteContact/:id")

Ces middlewares seront présentés plus bas.

La syntaxe présente sur les deux dernières routes : "":"/id" permet de préciser au routeur qu'un paramètre id est attendu dans l'URL.

7.3 - Controllers

Dans les fichiers **controllers** on peut retrouver la logique de traitement de la donnée transmise ou à transmettre mais également la logique permettant d'interagir avec la base de données.

En fonction des routes utilisées par le routeur, une fonction du controller sera exécutée à son tour.

Ils font appel à une instanciation de la base de données faite dans le dossier de services.

```
/* create database connection */
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USERNAME || 'root',
  password: process.env.DB_PASSWORD || '',
  database: process.env.DB_DBNAME || 'chat-nodejs',
  queueLimit : 100,
  connectionLimit : 100,
  multipleStatements: false
});

module.exports = pool.promise()
```

Nous avons fait appel au package NPM **mysql2** permettant de lier notre serveur Node.js avec notre base de données MySQL pouvant ainsi effectuer des requêtes.

Les variables de connexion à la base de données sont définies dans un fichier .env par souci de sécurité. On exporte ensuite une 'pool' de connexion permettant de requêter notre base de données.

Nous importons ensuite l'instance de connexion ouverte afin de pouvoir l'utiliser.

```
/**Import Base de données */
const pool = require('../services/database')
```

```

getAllUsers: async (req, res, next) => {

  try {
    const sql = "SELECT id, firstname, lastname, pseudo, urlAvatar, mail FROM users ORDER BY firstname ASC"

    const [query] = await pool.query(sql)

    return res.status(200)
      .send({data: query})
      .end()
  } catch (error) {
    return res.status(403)
      .send({message: "Une erreur est survenue durant le traitement"})
      .end()
  }
},

```

Grâce à la méthode `.query()` nous pouvons envoyer une promesse de requête qui sera retourné lors de la résolution de cette dernière. Elle prend en paramètre une requête SQL ainsi qu'un tableau de données facultatif à transmettre dans à la requête.

En cas de bon déroulement, on retourne la donnée récupérée en base vers la couche de présentation, sinon on retourne un statut et un message d'erreur.

7.4 - Middlewares

Comme notifié plus haut, le middleware se place entre deux couches de logiciels. Dans le cas d'Express il se place entre la requête et la réponse.

Il s'agit d'une fonction permettant d'appliquer une logique à une requête.

Elle dispose d'accès aux paramètres de requête / réponse et permet donc un grand nombre de d'actions sur ces dernières.

Nous nous sommes notamment servis de middleware pour vérifier que l'utilisateur était bien connecté ou possédait bien le rôle d'administrateur.

Il peut servir à une route unique ou à plusieurs.

7.5 - Sécurité

7.5.1 - Exemple de failles

Sur les documentations officielles de Node Js et Express on peut retrouver une section portant sur la sécurité. Elle apporte des explications sur les bons procédés mais aussi les différentes vulnérabilités à prévenir afin de sécuriser une application Web Node / Express. On peut notamment trouvé :

- Les failles **Cross-Site-Scripting (XSS)** : Elle consiste à injecter du code dans une page. Pour s'en protéger, il est indispensable d'encoder les données provenant de l'utilisateur.

- Les **injections SQL** : Il s'agit d'ajouter une requête non prévue à une requête SQL pour interagir avec la base de données. Une faille SQL permet de voler ou modifier des données, voire exécuter du code à distance.
- les **failles CSRF** : se produit lorsqu'un attaquant arrive à obliger des utilisateurs finaux à exécuter des actions non désirées et sans qu'ils ne s'en rendent compte. Les attaques CSRF visent les demandes de création, modification et suppression de données et sont menées sur les utilisateurs authentifiés.
- Les **attaques par bruteforce** : il s'agit d'une pratique courante pour essayer d'obtenir un mot de passe ou une clé. Cela consiste à tester toutes les possibilités une à une.
- Les **attaques DDOS** : qui consiste à surcharger le trafic de l'application ayant pour but de rendre indisponible le service.

Parmi cette documentation, Node Js et Express mettent à disposition des pratiques mais également toutes sortes d'outils permettant de sécuriser les différentes failles citées précédemment.

Voici les pratiques / outils à utiliser pour prévenir les attaques :

- Pour commencer, la documentation d'Express nous précise de n'utiliser que des versions à jour du module. Les versions 2x et 3x n'étant plus maintenus, les problèmes de sécurité et de performances ne sont donc plus corrigés.
- Pour la prévention de failles type XSS, Express conseille de mettre en place un **TLS** (transport layer security).
De façon simplifiée, le protocole TLS permet au serveur de s'authentifier et génère ensuite une clé de chiffrement. Ce chiffrement des données assure une plus grande sécurisation de la navigation (confidentialité et intégrité des données).
On parle alors de HTTPS, combinaison entre HTTP et TLS ou SSL (prédecesseur).

Outil : **Helmet.Js** est une bibliothèque JavaScript open source qui vous aide à sécuriser votre Node.js en définissant plusieurs en-têtes HTTP. Il agit comme un middleware pour Express et les technologies similaires, ajoutant ou supprimant automatiquement des en-têtes HTTP pour se conformer aux normes de sécurité Web

- Concernant les **injections SQL** : Il existe quelques méthodes courantes permettant d'empêcher les attaques par injection SQL.
Ne pas autoriser plusieurs déclarations SQL pour chaque requête.

multipleStatements: false

Un autre moyen de prévenir de ce type de faille concerne l'utilisation de **requêtes dites préparées**. Il s'agit de définir en amont une requête avec des paramètres dans la requête au lieu de valeurs constantes.

Il faut également bien vérifier les entrées de l'utilisateur à chaque requête.

Outil : On peut notamment utiliser un **ORM** pour les requêtes. Cela ne veut pas dire que la solution proposée par l'ORM est entièrement sécurisée contre ces failles. Il faut quand même effectuer des vérifications.

- Pour les **failles CSRF** : le principe est de sécuriser les requêtes et de s'assurer qu'elles proviennent bien de l'application web.
Pour cela nous avons mis en place un **système d'authentification** de l'utilisateur à l'aide d'un jeton **JWT**. Il est ensuite transmis dans les headers de la requête.
« Authorization: Bearer <jeton> ».
Il est également possible d'utiliser des outils anti-CSRF comme **csurf**.
- Pour ce qui concerne le **bruteforce** : nous avons mis en place un regex obligeant l'utilisateur à enregistrer un mot de passe avec un minimum de caractères permettant de le rendre plus long à "craquer". Nous avons également ajouté une limite de nombre de requêtes simultanées. Il est également possible de mettre en place un nombre limité de requête par minute par IP à l'aide de package node : express-brute par exemple
- Même cas pour le **DDOS**.

7.5.2 - JSON Web Token

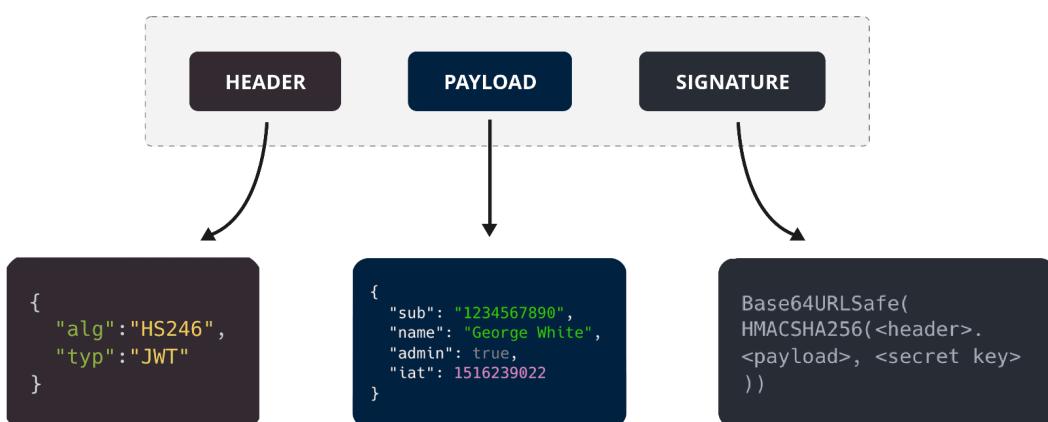
JWT est un jeton autonome sous forme de chaîne de caractères qui se compose de trois parties :

- L'en-tête (header) contient les informations sur le jeton et la façon dont la signature est générée (algorithme utilisé pour la signature)
- La charge utile (payload) contient les déclarations sur l'utilisateur nécessaires pour lui accorder l'autorisation d'accès. En général sous forme de chaîne de caractères hashé en base 64.
- La signature valide l'utilisateur. Elle sert également à vérifier que les données n'aient pas changé. En cas de mauvaise signature, le token sera rejeté.

Le jeton peut être signé avec une clé secrète qui permet de valider l'origine des données transférées.

Ainsi, le JWT peut-être transmis aux différentes requêtes nécessitant une authentification de l'utilisateur et effectuer cette requête en sécurité.

Structure of a JSON Web Token (JWT)



7.5.3 - Gestion des rôles

Toujours dans le contexte de sécurité, nous avons créé deux **middleware** permettant d'abord de vérifier qu'un utilisateur est bien **authentifié** afin d'accéder à certaines requêtes.

La différence entre les deux se base sur le **rôle** de l'utilisateur en question.

Le premier middleware permettra aux utilisateurs 'lambda' d'effectuer des actions comme par exemple :

- poster un message
- consulter / mettre à jour ces données

Le second vérifiera que le rôle de l'utilisateur correspond bien à celui d'un administrateur afin de lui laisser accès à la partie modération.

Cette séparation permet un maintien de la sécurité sur la base de données ne donnant pas accès aux utilisateurs à toutes les fonctions interagissant avec la base de données.

7.5.4 - Quelques outils

Il existe une multitude d'outils mis à disposition sur le web permettant de renforcer la partie sécurité d'une application web / mobile.

Voici une liste de quelques exemple intéressant :

- **Snyk** : c'est un outil pouvant être intégré à GitHub, Jenkins pour trouver et corriger les vulnérabilités connues.
À un niveau élevé, Snyk fournit une protection de sécurité complète, y compris les éléments suivants :
 - Trouver des vulnérabilités dans le code
 - Surveillez le code en temps réel
 - Corrigez les dépendances vulnérables
 - Soyez averti lorsqu'une nouvelle faiblesse affecte votre application.
 - Collaborez avec les membres de votre équipe

Snyk maintient sa propre base de données de vulnérabilités, et actuellement, il prend en charge Node.js, Ruby, Scala, Python, PHP, .NET, Go, etc.

- **Jscrambler** : Jscrambler rend votre application Web auto-défensif pour lutter contre la fraude, éviter la modification du code en cours d'exécution et les fuites de données, et protéger de la perte de réputation et des affaires.

Une autre fonctionnalité intéressante est la logique d'application, et les données sont transformées de sorte qu'elles sont difficiles à comprendre et cachées du côté client. Cela rend difficile de deviner l'algorithme, technologies utilisées dans l'application.

Jscrambler prend en charge la plupart des Frameworks JavaScript tels que Angular, Ionic, Meteor, Vue.js, React, Express, Socket, React, Koa, etc.

- **Rate Limit Flexible** : Utilisez ce petit paquet pour limiter le taux et déclencher une fonction sur l'événement. Ce sera pratique pour se protéger des attaques DDoS et par force brute.

7.5 - Situation : upload de données avec fichier

Les utilisateurs ont la possibilité sur Open Speech d'**ajouter une image d'avatar** lors de leur inscription mais peuvent également l'ajouter depuis l'édition du profil.

Voici comment nous avons procédé pour mettre en place l'upload d'image depuis l'application.

Pour l'inscription d'un utilisateur : Après avoir rempli la première étape du formulaire, ce dernier aura la possibilité d'ajouter un avatar ou photo de profil.

A ce moment la **couche de traitement** va effectuer deux logiques distinctes.

Après avoir vérifié les différentes entrées de l'utilisateur et avoir opéré les traitements de sécurité nécessaire (hachage de mot de passe par exemple) l'utilisateur est enregistré en base de données mais sans son image de profil.

Cela permet de ne pas ajouter d'images dans le dossier public/images du serveur au cas où l'inscription ne se serait pas bien déroulée ou ne serait pas finie.

Nous évitons également de développer une logique de suppression des images provenant d'inscriptions inaccomplies.

Dans le cas où l'inscription s'est déroulée sans erreur, une seconde logique permettant l'upload de l'image vers le serveur sera déclenchée.

En voici quelques détails.

Nous avons décidé d'utiliser **multer** qui est un middleware Node Js permettant la gestion **multipart / formdata** principalement utilisé pour télécharger des fichiers.

```
var multer = require('multer');

const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "public/images");
  },
  filename: function (req, file, cb) {
    cb(null, Math.random().toString(36).slice(2, 7) + file.originalname);
  },
});
```

On commence donc par créer un “**disk Storage**”.

Le moteur de stockage sur disque vous donne un contrôle total sur le stockage des fichiers sur le disque.

Deux options sont disponibles, destination et filename. Ce sont deux fonctions qui déterminent où le fichier doit être stocké.

Destination : est utilisé pour déterminer dans quel dossier les fichiers téléchargés doivent être stockés. Cela peut également être donné sous la forme d'un string(par exemple '/tmp/uploads'). Si aucune destination n'est donnée, le répertoire par défaut du système d'exploitation pour les fichiers temporaires est utilisé.

Filename : est utilisé pour déterminer le nom du fichier dans le dossier. Si aucun filename n'est donné, chaque fichier recevra un nom aléatoire qui n'inclut aucune extension de fichier.

```
const upload = multer({
  storage: storage ,
  limits: {
    fileSize: 1024 * 1024 * 5
  },
  fileFilter: function (req, file, cb) {
    if (file.mimetype === "image/jpeg" || file.mimetype === "image/png") {
      cb(null, true);
    } else {
      cb(new Error("Image uploaded is not of type jpg/jpeg or png"), false);
    }
  }
});
```

On va ensuite utiliser le moteur de stockage sur disque précédemment créé avec multer.

On déclare également une **taille maximale** aux fichiers à l'aide de l'attribut limits.

Puis on utilise la fonction fileFilter qui aura accès aux différents éléments de la requête notamment le fichier à télécharger. On va donc pouvoir effectuer une restriction / vérification du **mimeType**.

Nous déclarons une route au userRouter : “**/upload**”.

Cette route aura pour but de modifier le champ contenant l'url de l'avatar de l'utilisateur en base de données et de stocker le fichier sur le serveur.

```
router.put("/upload", upload.single("profile"), userController.avatar, (req, res, next) => {
  const file = req.file;
  if (!file) {
    const error = new Error("Please upload a file");
    error.statusCode = 400;
    return next(error);
  }
  res.status(200).send(file).end();
});
```

On utilise ensuite la fonction **single()** du stockage multer paramétré précédemment et contenu dans la constante upload. Cette fonction permet de n'accepter qu'un seul fichier et de le stocker au format **req.file** pour pouvoir y accéder par la suite.

On fera par la suite appel à la fonction **avatar()** du controller.

C'est dans cette fonction qu'est mis à jour le champ en base de données contenant le chemin vers l'image de profil de l'utilisateur.

```
const filename = req.file.filename

const sql = `UPDATE users SET urlAvatar = '${filename}' WHERE
```

Un statut **201** sera retourné si le téléchargement s'est effectué sans problème.

En cas d'erreur ou si aucun fichier n'a été ajouté à la requête, on retourne un statut **400** indiquant qu'un problème est survenu.

7.6 - Swagger UI : Documentation de l'API

Nous avons mis en place une documentation de l'API à l'aide de **Swagger** :

- Swagger est un langage de description d'interface permettant de décrire des API exprimées à l'aide de JSON. Swagger est utilisé avec toute une série d'outils logiciels open source pour concevoir, créer, documenter et utiliser des services Web.

Cette documentation est importante pour plusieurs raisons :

- Elle permet aux développeurs ne connaissant pas le projet de pouvoir mieux se repérer et comprendre plus rapidement le fonctionnement de l'API.
- Elle facilite le maintien de l'application en servant de rappel sur les différents processus.

Nous avons donc créé un interface graphique à l'aide de Swagger permettant de consulter et tester les routes renseignées dans l'API.

The screenshot shows the Swagger UI interface for the 'Open Speech' API. At the top, it says 'Documentation API : Open Speech 1.0.0 OAS 3.0'. Below that, it says 'Documentation de l'API de l'application mobile et Web Open Speech'. A dropdown menu 'Servers' is set to 'http://localhost:3000/'. The main area is divided into sections: 'Utilisateurs' and 'Conversations'. Under 'Utilisateurs', there are several endpoints listed with their methods and URLs: GET /user/allUsers, POST /user/register, POST /user/login, PUT /user/profile/update/{id}, GET /user/mesContacts, POST /user/contact, and DELETE /user/deleteContact/{id}. Under 'Conversations', there is one endpoint listed: GET /conversations/mine.

7.7 - Phase de tests

Lors de la phase de développement d'une API, Un banc de test permet aux développeurs de déterminer si les API répondent aux attentes en matière de fonctionnalité, de performance, de fiabilité et de sécurité. L'objectif est d'identifier les bugs et tout autre comportement suspect afin que vos utilisateurs ne se retrouvent pas avec un produit de mauvaise qualité ou manquant de fiabilité.

Il existe un certain nombre de tests à réaliser pour une API, parmi eux notamment :

- les **tests fonctionnels**, il teste différentes fonctionnalités de votre codebase. Les tests consistent en des scénarios spécifiques visant à garantir que les fonctionnalités de l'API correspondent aux paramètres définis et attendus.
- les **tests de fiabilité**, ce type de test permet de vérifier que l'API peut opérer sans défaillance pendant une durée déterminée dans un environnement spécifique.
- les **tests de flux**, ce test permet de voir si les performances de l'API sont optimales dans des conditions normales et de plus forte amplitude.
- les **tests de sécurité**, il est utilisé pour garantir que l'API soit en mesure de faire face aux menaces externes. Les tests portent notamment sur les méthodes de cryptage, le contrôle d'accès aux API, la gestion des droits des utilisateurs et la validation des autorisations.
- les **tests négatifs**, l'objectif des tests négatifs est de voir ce qui se passe lorsque l'application est confrontée à une entrée non valide ou non intentionnelle. Il s'agit de déterminer ce qui pourrait provoquer la défaillance du système afin de développer des solutions plus appropriées.

Il existe de nombreux outils permettant d'effectuer des tests sur une application.

Le premier outil que nous avons utilisé est **Postman**. Il permet de tester des requêtes http, des API, utiliser des environnements, exporter en JSON ... Depuis l'interface graphique proposée.

Il nous a donc servi à tester chaque route implémenter dans le projet, dans le but d'observer les retours de réponses, mais également d'ajouter des fonctions de test.



```

POST http://192.168.1.34:3000/messages/translate
Params Authorization Headers (8) Body • Pre-request Script Tests • Settings
1 pm.test("Status code is 200", function () {
2   | pm.response.to.have.status(200);
3 });
4 pm.test("Body matches string", function () {
5   | pm.expect(pm.response.text()).to.include("translation");
6 });
7
  
```

Test Results (2/2)

All	Passed	Skipped	Failed
PASS	Status code is 200		
PASS	Body matches string		

Dans l'exemple présenté ci-dessus, on peut voir que deux fonctions `.test()` ont été ajoutées. Un test sera chargé de vérifier que le statut de retour de la réponse est bien 200. Le deuxième vérifie que la réponse inclut la chaîne de caractères : "translation".

Jest : Il s'agit de l'outil de test le plus populaire et le plus recommandé pour React.

Il s'agit d'un véritable cadre de test, créé par facebook.

On peut notamment utiliser Jest pour :

- Tester un composant de manière isolée
- Tester l'API publique d'un composant (propriétés et méthodes `@api`, événements)
- Tester l'interaction utilisateur de base (comme -clics)
- Vérifier la sortie DOM d'un composant
- Vérifiez que les événements se déclenchent lorsque prévu

Il s'adapte à tous les frameworks et librairies Javascript, il est facilement configurable et utilisable. Il permet de mettre en place des tests unitaires, fonctionnels et end to end par exemple.

8 - Développement : Front-end

Le front-end d'une application mobile se pense différemment d'une application Web.

Dans un premier temps, l'appareil de l'utilisateur n'offre pas le même espace d'affichage que sur un desktop. Nous avons donc pris ce paramètre en compte en affichant seulement le nécessaire afin de ne pas surcharger les screens.

Enfin la navigation n'est pas la même que sur un site internet (empilement d'écrans), il faut penser le parcours utilisateur en fonction. Nous avons trouvé important de ne pas ajouter de screen inutile dans le but de faciliter l'accès aux fonctionnalités et informations importantes.

8.1 - Organisation et arborescence

Les phases de conception de la partie front nous ont permis de mettre rapidement en évidence les éléments aux apparitions les plus récurrentes sur nos maquettes / screens.

Nous avons réfléchi aux différents **composants génériques** / réutilisables que nous pouvions développer en amont afin d'éviter la répétition de code et optimiser notre temps de production.

Nous souhaitions également proposer un dark / light thème sur l'application. C'est pourquoi nous avons défini deux feuilles de style (une pour chaque thème) en amont afin de pouvoir utiliser des variables qui changeraient de valeur en fonction du thème actuel.

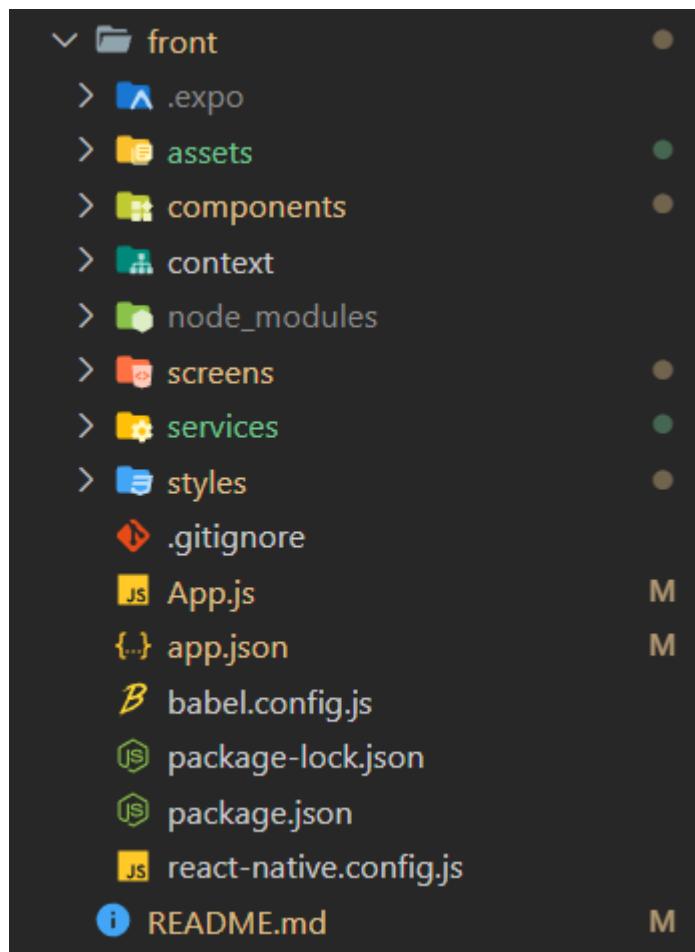
Nous avons commencé par initialiser un nouveau projet **React native** intégrant **Expo** à l'aide de la commande : `npx create-expo-app "app name"`.

Il créera un nouveau répertoire de projet et installera toutes les dépendances nécessaires pour que le projet soit opérationnel localement.

Voici la structure du projet après initialisation :

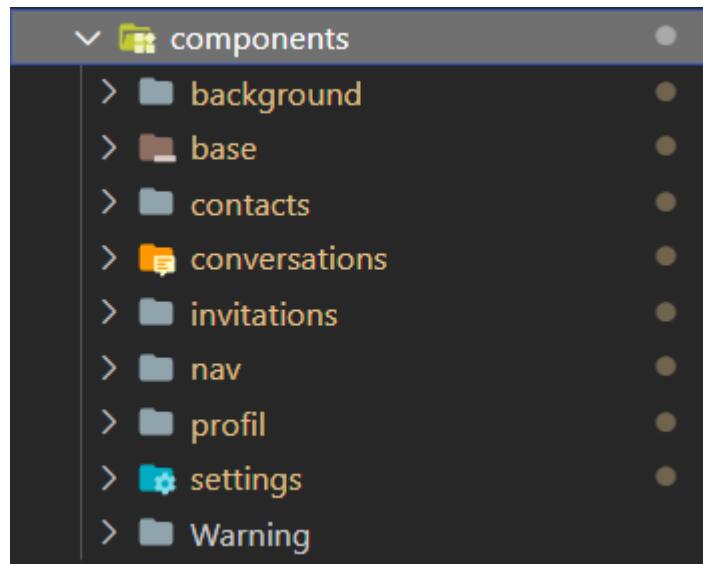
- **App.js** : C'est le fichier racine que vous chargerez après avoir exécuté le serveur de développement avec npx expo start.. Généralement, ce fichier contiendra les contextes React et la navigation au niveau racine. Il s'agit de l'écran par défaut.
- Le fichier **app.json** contient des options de configuration pour le projet. Ces options modifient le comportement de votre projet lors du développement, ainsi que lors de la création, de la soumission et de la mise à jour de notre application.
- Le dossier **assets** contient adaptive-icon.png utilisé pour Android et un icon.png utilisé pour iOS comme icône d'application. Il contient également splash.png qui est une image pour l'écran de démarrage du projet et un favicon.png si l'application s'exécute dans un navigateur. C'est dans ce dossier que l'on peut stocker les images, icônes à utiliser dans l'application côté client.
- Le dossier contient également des fichiers de base comme un .gitignore - package.json ou encore babel.config.js

Suite au développement du projet, voici la structure '**finale**'



- Le dossier **Expo** contient des informations concernant le dernier device à avoir utilisé le projet (relatif à cet appareil). Mais également des configurations (serveur par exemple) .

- Le dossier **assets** contient les fichiers, images, icônes
- Le dossier **components** contient les composants réutilisables / génériques rangés par catégorie ou screen.



Nous pouvons notamment y retrouver les boutons d'envoi de formulaire (submit), les headers des screens avec un titre défini, des listes, des éléments de liste, des messages d'alerte / d'aide. Ce sont les éléments les plus utilisés dans l'application.

- Le dossier **context** contient le context mis en place dans l'application. Il sert à stocker des données et les rendre accessibles depuis tous les screens.
- Le dossier **screens** qui contient la totalité des screens de l'application.
- Le dossier **services** qui contient un fichier de config renseignant la totalité des URL de base de l'API (exemple API_USERS = BASE_API + /users).
- Le dossier **styles** contient des éléments de style mis en place au début du projet afin de pouvoir les réutiliser tout au long du développement. Nous y trouvons un fichier Base qui contient des espacements définis, des calculs de dimensions etc.. Mais également un fichier dark / light theme définissant les couleurs en fonction du thème.

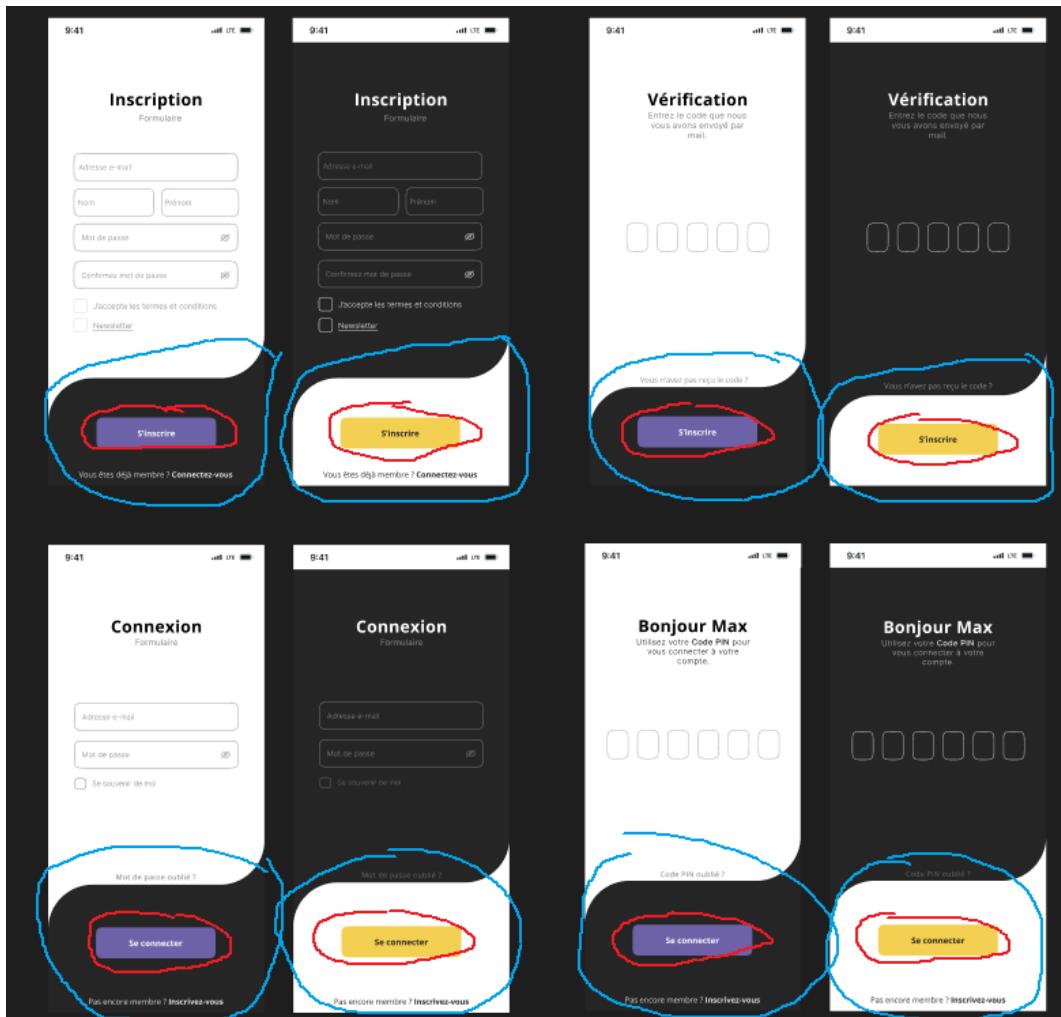
8.2 - Exemple de composants génériques

Un des grands principes de React concerne les **composants**.

Ils permettent d'éviter la répétition de code et d'avoir un dossier de projet propre et bien organisé.

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

Comme expliqué précédemment, l'étape de maquettage permet donc de visualiser les éléments les plus répétitifs à l'écran dans le but d'en faire des composants.



Voici un exemple parlant d'éléments quasi similaires apparaissant à de nombreuses reprises dans l'application :

On peut noter la présence sur chaque screen d'un **bouton** (de couleur différente selon le thème de l'appareil) d'action - En rouge.

On peut également voir que le **background** se répète sur les différents screens - En bleu. Il était donc intéressant de créer ces composants en amont afin de pouvoir les réutiliser.

Voici comment nous avons procédé pour le bouton d'action :

```
const Button_submit = ({titre , onPress}) => {
    const { colors } = useTheme();
```

Il s'agit d'un composant **fonctionnel** (puisqu'il est construit à partir d'une fonction et non pas d'une classe) auquel nous avons ajouté deux **props** / propriétés.

Titre contiendra le texte qui sera affiché sur le bouton.

OnPress permet de transmettre l'état du bouton au composant parent en cas de changement d'état, dans notre cas au clic..

```
return (
  <TouchableOpacity style={styles.button} onPress={onPress}>
    <Text style={styles.buttonLabel}>{titre}</Text>
  </TouchableOpacity>
)
```

On retourne notre élément visuellement dans la fonction de retour du composant. React Native met à disposition un certain nombre de composants, parmi eux les "touchables". Ils offrent la possibilité de capturer un événement de tapotement sur l'écran pour pouvoir y assimiler une logique particulière.

Nous avons utilisé Touchable Opacity qui est une zone dans laquelle une pression peut être détectée.

Lors de cette pression, le composant parent sera mis à jour de part le changement d'état transmis par onPress.

Enfin, voici comment utiliser ce composant :

```
// Base components
import Titre_display from "../../components/base/Titre_display";
import Button_submit from "../../components/base/Button_submit";
import ValideWarning from "../../components/Warning/ValideWarning";
```

Il faut d'abord l'importer.
Ensuite on peut appeler notre fonction Button_submit() en lui passant en propriété le titre et la fonction qui devra être déclenchée au clic.

```
<Button_submit titre="Se connecter" onPress={onPress}/>
```

Nous avons donc défini plusieurs composants qui nous ont permis un gain de temps lors du développement. En ajoutant le background, le titre de la page mais également le bouton d'action nous pouvions rapidement construire la base d'un screen avec les éléments principaux et leur style déjà défini et appliqué.

Les dossiers de composants contiennent également des listes, leurs éléments 'individuels', des messages d'alerte, de réussite. Permettant notamment de garder le fichier de screen lisible et de ne pas le surcharger de code.

8.3 - Mode sombre

Nous avons implémenté un **mode sombre** à l'application.

Le mode sombre est utile pour diverses raisons :

- **Amélioration de la concentration**, il est plus facile de visualiser la structure d'un texte sur un fond sombre. Exemple avec les éditeurs de code en mode sombre par défaut.
- **Réduction de la fatigue visuelle**, il permet de limiter la diffusion de lumière bleue émise par un fond plus clair. Cela a pour effet de réduire l'inconfort visuel.
- **Amélioration de la qualité de sommeil**.
- **Réduction de la consommation d'énergie**.

A l'aide du hook react **useColorScheme()**, il est possible de savoir le thème couleur actuel de l'appareil de l'utilisateur. En fonction de la valeur de ce dernier : light ou dark, on passera le thème couleur concerné au container principal de navigation. Les valeurs des variables de couleurs seront alors mis à jour en fonction du thème utilisé.

Le composant **NavigationContainer()** proposé par react-navigation contient un prop nommé '**thème**'.

Ce dernier permet de transmettre un thème couleur à différents éléments de react-navigation ainsi qu'à l'application. Tous les composants seront ainsi mis à jour à chaque changement de mode de l'appareil.

```
const colorScheme = useColorScheme();
```

```
<NavigationContainer theme={colorScheme === 'light' ? AppLightTheme : AppDarkTheme}>
```

8.4 - Navigation

Dans React Native, la navigation se gère différemment que pour un site Web.

Il s'agit d'une série **d'écrans empilés** les uns sur les autres, pour naviguer de l'écran actuel vers un nouveau, il faut ajouter l'écran de destination en haut de la pile d'écran, et il faut supprimer l'écran en haut de la pile pour pouvoir revenir au précédent.

Il met également à disposition le **Stack Navigator** ainsi que le **Tab Navigator** qui sont l'équivalent d'un header / footer de navigation.

A l'intérieur de ces derniers, on peut définir différents screens ainsi que le composant qui lui sera rattaché

```
<HomeStack.Navigator screenOptions={{ headerShown: false }} >
  <HomeStack.Screen name="Connexion" component={HomeScreen} />
  <HomeStack.Screen name="Inscription" component={Inscription_screen} />
  <HomeStack.Screen name="Conversations" component={Tabs} />
```

. On peut voir sur la photo ci-dessus le **HomeStack.Navigator** ou navigateur de pile. Il contient les différents screens vers lesquels l'utilisateur peut naviguer.

Nous avons paramétré le navigateur afin de ne pas afficher le header de navigation sur les screens. Sur nos maquettes nous avions défini un menu de navigation en bas de page, c'est donc un autre composant qui sera chargé de gérer la navigation entre les écrans proposés dans le menu de navigation. Certains écrans ne sont accessibles qu'à partir d'autres écrans et ne doivent donc pas apparaître dans le menu.

Nous avons donc procédé comme suit :

On peut noter l'écran nommé "conversations" faisant appel au composant Tabs. Dans ce même composant, on pourra retrouver les écrans accessibles depuis le menu de navigation seulement.

Nous avons utilisé un le **Tab.Navigator** de react-navigation permettant la navigation par onglet puis avons défini le style de la **TabBar** dans les options d'écrans.

```
function Tabs(){
```

```
<Tab.Group>
  <Tab.Screen name="Home" component={Home_general} />
  <Tab.Screen name="Messages" component={Conversations_screen} />
  <Tab.Screen name="Contacts" component={Contacts_screen} />
  <Tab.Screen name="Settings" component={Settings_screen} />
</Tab.Group>
```

8.5 - Persistance de données

Par soucis de confort pour l'utilisateur mais également de sécurité, nous avons intégré une fonctionnalité de connexion à l'application par code **PIN**.

En effet, l'utilisateur aura la possibilité une fois inscrit, de paramétrier un code PIN à 4 chiffres lui permettant de se connecter plus rapidement et facilement à l'application tout en gardant une part de sécurité.

Lorsque l'utilisateur a renseigné un code PIN, et a coché la case 'se souvenir de moi' après son passage sur l'écran de connexion, il pourra s'authentifier à l'aide de son code lors de sa prochaine connexion.



Nous sommes passés par le **secure store** de **Expo** qui fournit un moyen de chiffrer et de stocker en toute sécurité des paires clé-valeur localement sur l'appareil. Chaque projet Expo a un système de stockage séparé et n'a pas accès au stockage des autres projets Expo.

```
import * as SecureStore from 'expo-secure-store';
```

Si la checkbox a été coché, on va enregistrer les données de l'utilisateur qui se connecte dans le secure store :

```
if (isChecked === true){
  save("user", JSON.stringify({id: currentId, token: tok, PIN: PIN, langage: langage}))
}
```

```
async function save(key, value){
  await SecureStore.setItemAsync(key, value)
}
```

La fonction **save()** sera chargée de sauvegarder une nouvelle paire de clef / valeur dans le secure store.

On va alors sauvegarder un nouvel élément “user” qui contiendra l'identifiant de l'utilisateur connecté, son token, son code PIN ainsi que le langage de traduction qu'il a défini à l'inscription ou dans son profil.

Le hook **useEffect()** est un **hook** qui va permettre de déclencher une fonction de manière asynchrone lorsque l'état du composant change.

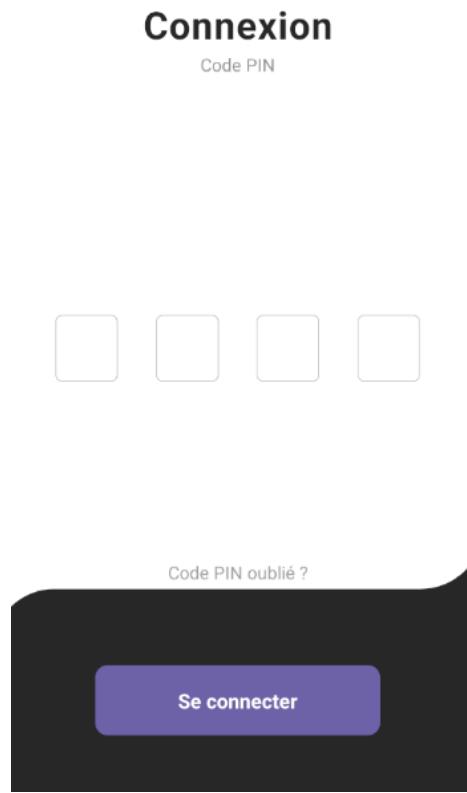
Nous allons donc l'utiliser dans l'écran de connexion afin de lancer la fonction **getValueFor()** à chaque changement d'état du composant.

```
useEffect(() => {
  getValueFor('user')
}, [])
```

On passera en paramètre de la fonction la clef que l'on veut récupérer dans le secure store à savoir “user”.

```
async function getValueFor(key) {
  let result = await SecureStore.getItemAsync(key);
  if (result) {
    let PIN = JSON.parse(result).PIN
    if (PIN)
      navigation.navigate('LoginByPin')
```

Enfin, si un code PIN est reconnu dans le store on va rediriger l'utilisateur vers la page de connexion par code PIN.



8.6 - Traduction

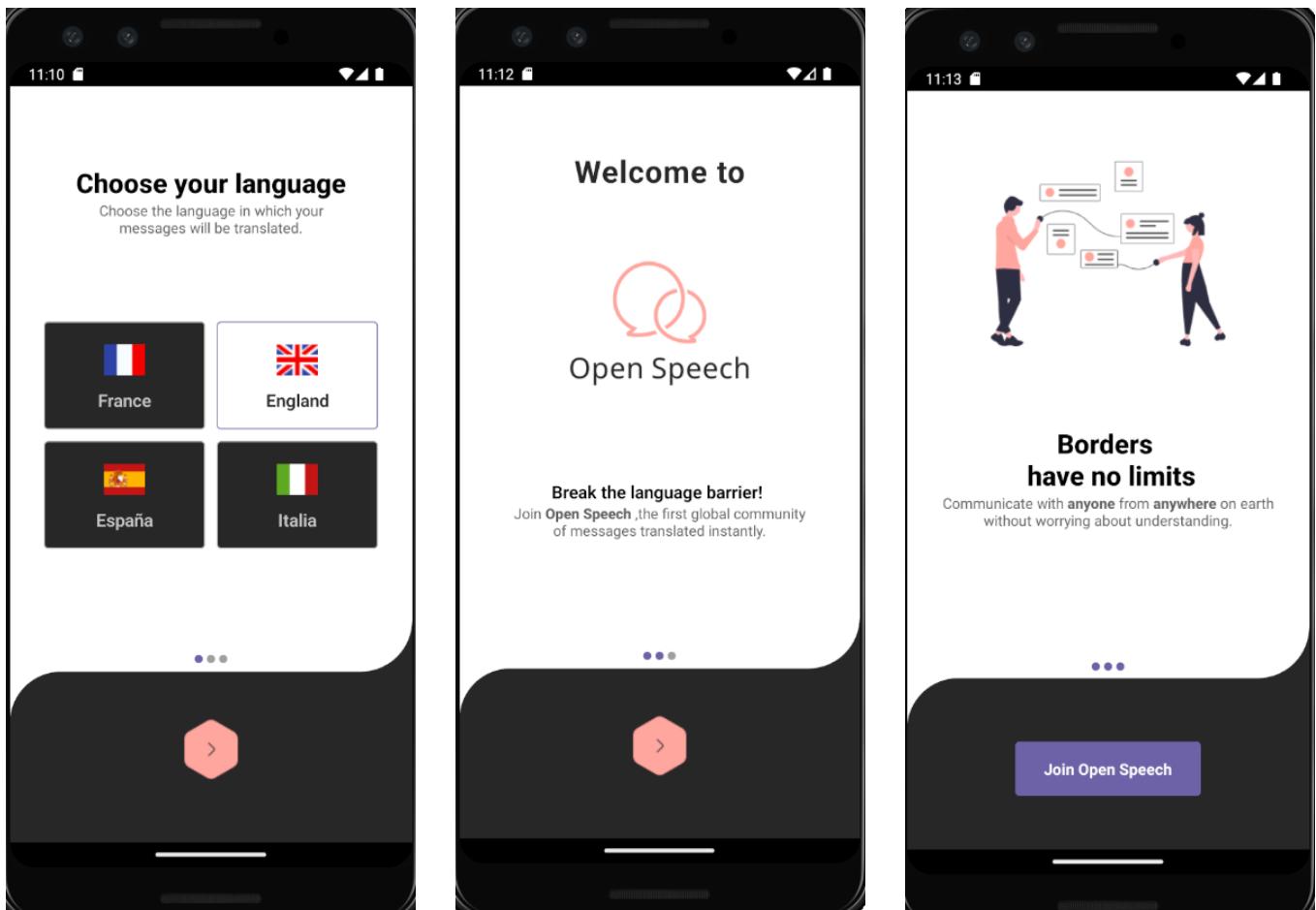
Une des fonctionnalités principales proposées par l'application est la **traduction**.

Après inscription de l'utilisateur, ce dernier va pouvoir se connecter.

Lors de sa première connexion (et première seulement) il aura accès à une navigation particulière entre 3 screens.

Le premier lui permettra de choisir son langage de traduction, les deux autres seront chargés de faire une brève présentation de Open Speech.

Son choix sera alors enregistré en base de données, dans le champ langage de la table utilisateur.



En fonction du choix du langage de l'utilisateur, l'application sera traduite.

Pour ce qui est des textes écrits en dur, nous avons plusieurs fichiers json contenant la traduction de l'application dans les langues disponibles.

La complexité se trouvait dans la traduction des messages généraux / privés reçus ou envoyés. Pour mettre en place cette fonctionnalité, nous avons fait appel à une API de google :

```
const translate = require('translate-google')
```

```
const [messages] = await pool.query(sql)

const translateMessages = await Promise.all(messages.map( async (msg, index) => {
    const translation = await translate(msg.content, {from: "auto", to: langue})
    msg.content = translation
    return msg
}))
```

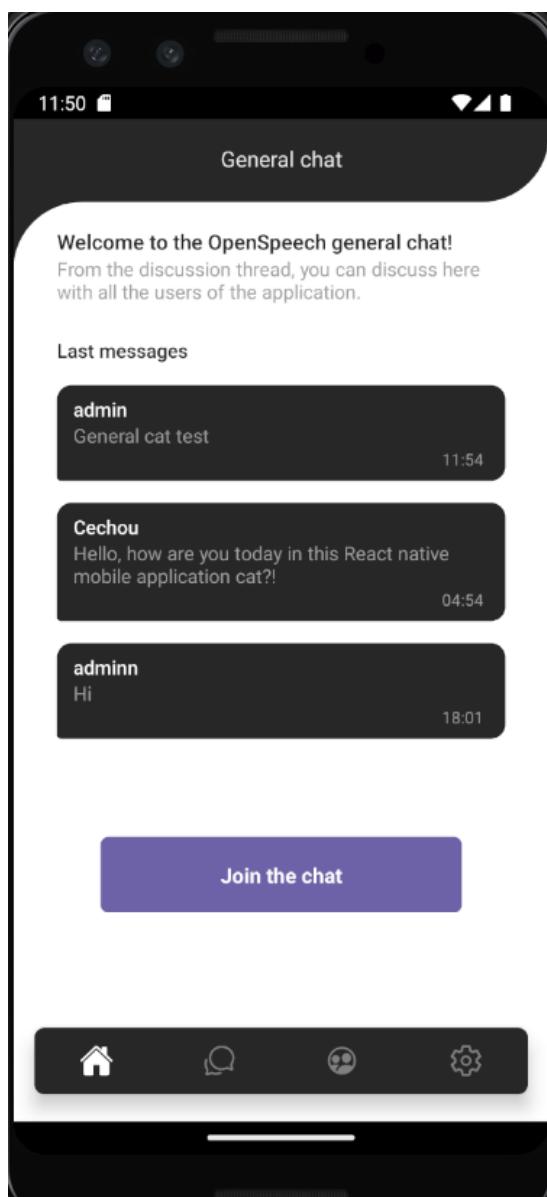
Après avoir récupéré les messages (du salon général, ou de conversations privés selon le contexte) nous allons utiliser un Promise.all() qui permet d'exécuter plusieurs promesses à la suite et retourne une erreur si une des promesses n'a pas été résolue dans le bloc.

Nous bouclons donc sur les messages afin d'effectuer un appel API vers la traduction de Google pour chaque message du salon / conversation.

Dans cet appel nous envoyons le contenu du message ainsi qu'un objet détenant deux paramètres :

- **from** : Langue dans laquelle le message à traduire est écrit. La valeur "auto" permet de laisser l'API reconnaître la langue.
- **to** : Langue dans laquelle le message doit être traduit. En fonction de la langue choisie par l'utilisateur, cette dernière sera transmise à la requête puis envoyée vers l'API de Google.

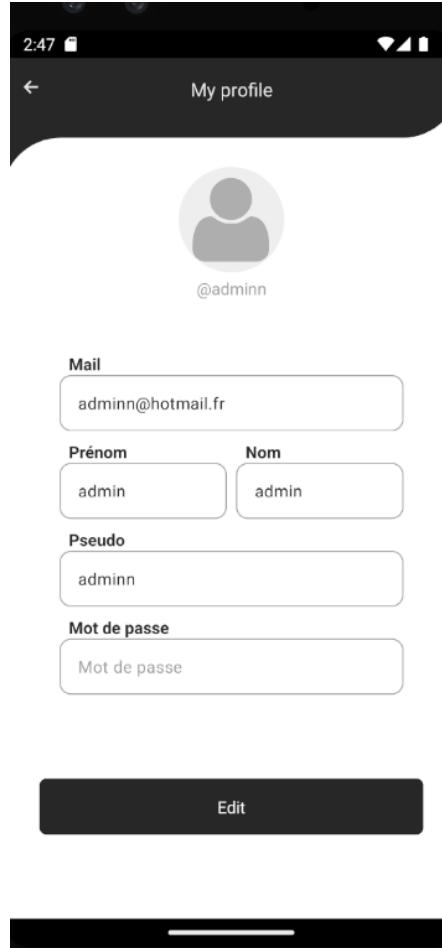
Nous remplaçons ensuite le contenu des messages par leur traduction avant de les retourner à la vue.



L'utilisateur a également la possibilité de modifier son langage de traduction depuis son profil.

8.7 - Mise à jour du profil

Depuis son profil, l'utilisateur a la possibilité de modifier ses informations à l'aide d'un formulaire. Il pourra également modifier sa photo de profil (indépendamment des informations).



Côté **front-end**, nous allons d'abord effectuer des vérifications sur les informations renseignées par l'utilisateur.

Nous gardons un tableau contenant les informations actuelles de l'utilisateur, lors de la soumission du formulaire, si une information a été modifiée et n'est plus similaire à celle connue avant pour l'utilisateur, alors elle devra être mise à jour. Sinon il n'est pas nécessaire de transmettre l'information au back-end étant donné qu'elle ne sera pas modifiée en base de données.

Côté **Back-end**, nous allons récupérer le corps de la requête, contenant les informations à modifier. Après avoir bouclé sur ces informations, nous allons remplir un tableau avec les données à modifier en base de données directement formater en syntaxe de requête SQL. Ce tableau sera ensuite transmis dans la requête à la base de données modifiant automatiquement les champs reçus.

```

var newArray = []

for (const [key, value] of Object.entries(array)) {
    if (key === "password"){
        if (key.length < 5){
            return res
                .status(400)
                .json({message: "Le mot de passe doit contenir au moins 5 caractères."})
        } else {
            const salt = await bcrypt.genSalt()
            const passwordHash = await bcrypt.hash(req.body.password, salt)

            newArray.push(` ${key} = '${passwordHash}' `)
        }
    } else {
        newArray.push(` ${key} = '${value}' `)
    }
}

if (newArray.length === 0)
    return

```

9 - Espace administration

Comme énoncé précédemment, notre application est également composée d'un panel d'administration. Ce dernier va permettre aux administrateurs du site d'effectuer des actions de modérations sur différents éléments de la base de données.

Notre application Web administrateur est développée à partir de la bibliothèque Javascript **React Js**.

Le but principal de cette bibliothèque est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page HTML à chaque changement d'état.

Nous avons choisi React Js pour la **cohérence technologique** avec l'application mobile, développée elle à l'aide de React Native.

Cette application Web va interagir avec la même API que l'application, seulement les routes nécessitent une authentification en tant qu'administrateur.

Nous n'avons pas réalisé de maquette pour la partie administration, préférant nous concentrer sur les fonctionnalités qu'il proposerait. Nous avons donc décidé de mettre en place un style minimaliste mettant en avant les fonctionnalités du panel.

L'application Web offre la possibilité aux administrateurs de modérer :

- Les **Utilisateurs**, il peut notamment éditer des informations (en cas de perte de mot de passe + email, autres cas particuliers)
- Le **chat général**, il peut éditer les messages et les supprimer au cas où ils seraient nuisibles pour le chat.
- Les **conversations**, il peut visualiser la totalité des conversations privées entre utilisateurs. Après avoir sélectionné une conversation, les messages relatifs à cette conversation pourront être supprimés. (à la demande particulière d'un utilisateur, nous partons du principe qu'un administrateur ne peut pas modifier de message privé entre deux utilisateurs : vie privée).
- Les **contacts** et les **invitations**, l'administrateur pourra les supprimer. L'édition n'a pas été rendue disponible par choix. Les utilisateurs ont la possibilité de gérer leurs invitations et leurs contacts depuis l'application mobile.

Pour ce qui est de l'arborescence, elle s'organise comme cela :

Nous avons 2 dossiers principaux

- **Public** : qui contient les images / icônes ainsi que l'index.html de React Js.
- **Src** : qui contient différents sous-dossiers :
 - **Components**, qui contient les sections du panel.
 - **Config**, qui contient les chemins d'accès à l'API.
 - **Functions**, qui contient des fonctions réutilisables.

Un point sur le sous-dossier components.

The screenshot shows the 'Utilisateurs' (Users) section of the OpenSpeech application. The left sidebar has a 'Gestion du site' menu with links for Utilisateurs, Chat général, Conversations, Contacts, and Invitations. Below this is a 'Paramètres' section with links for Graphiques, Tendances, and Abonnements. A green footer bar at the bottom has a question mark icon. The main content area displays a table of user data:

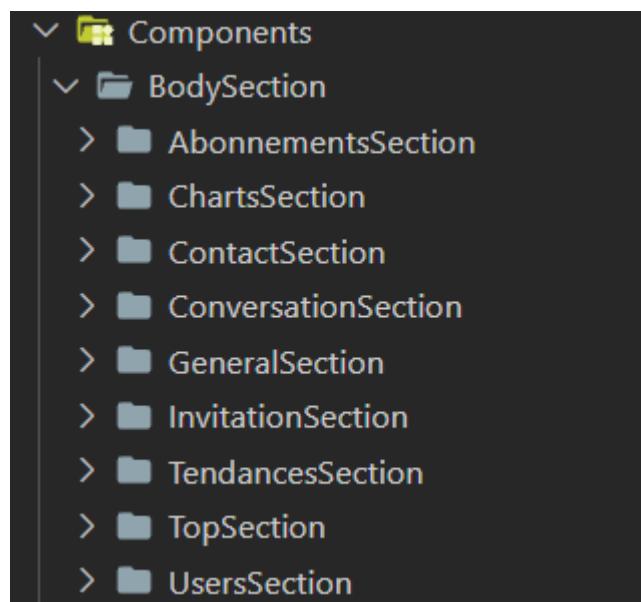
ID	Nom	Prénom	Email
17	admin	admin	admin@hotmail.fr
38	Ano	Dédé	Les@hotmail.fr
25	Test	Cecilia	Cechou@hotmail.fr
36	Test	Test	Test@hotmail.fr
39	Max	Max	Le@hotmail.fr
40	Max	Max	Tessst@gmail.fr
41	Max	Max	Tieis@hotmail.fr
42	admin	admin	adminnn@hotmail.fr
43	max	max	max@hotmail.fr
54	max	max	maxx@hotmail.fr

Plusieurs sous-dossier sont présents, organisé comme tel :

- Login
- BodySection
- SideBarSection

Dans le dossier Login, on retrouve le fichier `jsx` contenant le HTML ainsi que les appels API. On peut également retrouver le style du composant dans un fichier `.scss` ainsi qu'un `.css` et un `.css.map`.

BodySection contient toutes les parties disponibles depuis la SideBarSection. Comme pour le dossier Login, chaque partie / section est rangée dans un dossier qui contient le `.jsx`, le `.scss`, le `.css` et le `.css.map`.



Dans les fichiers `jsx`, on importe le fichier `css` qui lui est rattaché. Ce fichier contiendra le code re compilé contenu dans le `.scss`

Scss nous a permis notamment de paramétriser les différentes couleurs à utiliser ainsi que la taille de polices d'écritures par exemple depuis une variable root.

```

--textColor: hsl(240, 1%, 48%);
--bgColor: hsl(220, 10%, 94%);
--greyText: rgb(190, 190, 190);
--inputColor: hsl(330, 12%, 97%);

--biggestFontSize: 2.5rem;
--h1FontSize: 1.5rem;
--h2FontSize: 1.25rem;
--h3FontSize: 1rem;
--normalFontSize: .938rem;
  
```

Pour la partie back-end, une route spécifique a été créée pour les administrateurs.

Elle est couverte par un middleware confirmant que l'utilisateur dispose du rôle nécessaire.

```
router.delete('/message/:id', isAuthenticated, adminController.deleteMessage)  
router.patch("/user/:id", isAuthenticated, adminController.updateUser)  
router.delete("/user/:id", isAuthenticated, adminController.deleteUser)
```

10 - Problématiques rencontrées

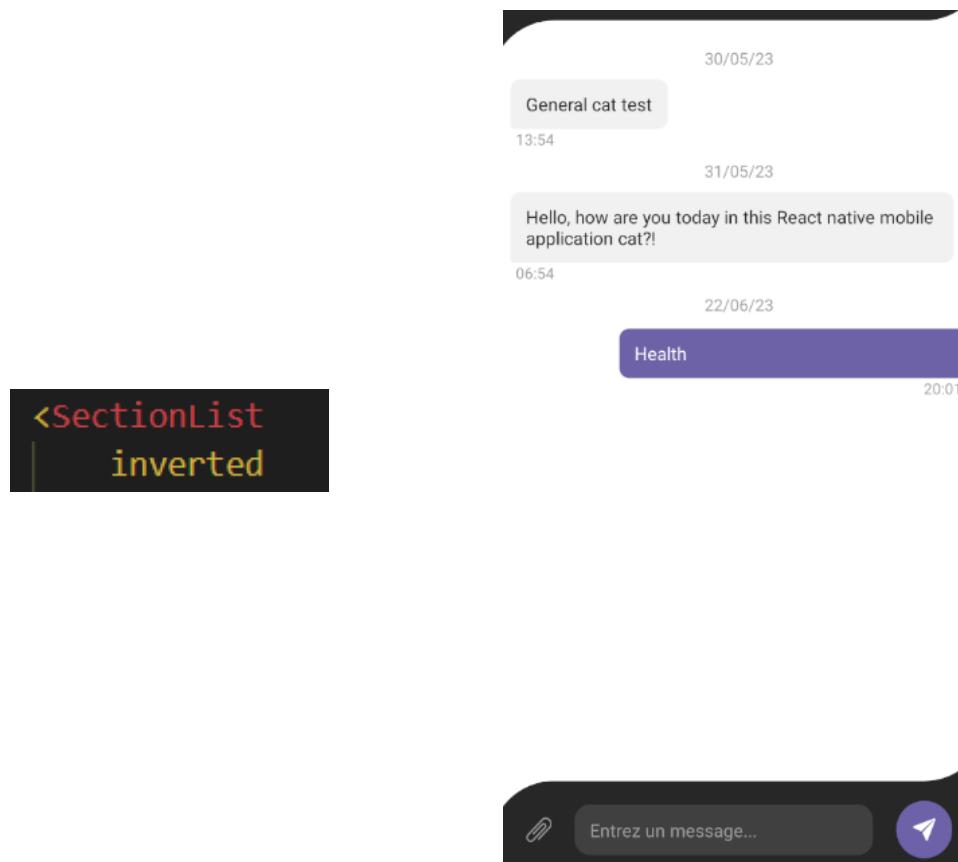
Durant le développement du front-end de l'application mobile, nous avons rencontré un problème majeur concernant l'affichage des messages pour les conversations privés de l'utilisateur.

Sur les applications de messagerie disponibles sur le marché, le dernier message reçu est affiché tout en bas de la page. Et le scroll est inversé (vers le haut) permettant de visualiser les messages les plus anciens.

Les messages sont également séparés et ordonnés par date d'envoi.

Pour cela, nous nous sommes documentés sur les possibles manières de faire et avons trouvé une solution.

Elle consiste à utiliser le composant **SectionList** de React native. Ce dernier détient une propriété “**inverted**” qu'il est possible de paramétrier. Sur true, l'ordre d'affichage des éléments sera inversé permettant d'obtenir l'effet désiré. Cette liste contient également des titres de section (qui seront donc les différentes dates d'envoi / réception des messages).



Une autre problématique que nous avons rencontré concerne l'accès à la base de données depuis un appareil émulant l'application mobile sur Expo.

En effet, il est possible depuis postman d'effectuer une requête vers notre serveur disponible sur le port **3000** du **localhost**. Hors, depuis un appareil externe le localhost n'est pas disponible. Nous avons donc dû trouver une solution pour pouvoir connecter cet appareil au serveur de la base de données.

Pour cela, nous nous sommes encore une fois documentés sur différents forums et avons trouvé des pistes intéressantes.

En modifiant l'adresse ip du localhost par **l'IPV4** de l'ordinateur.

11 - Recherche anglophone

La plupart des documentations que nous avons consultées pour le projet sont en anglais par défaut, notamment Express, React, Node Js.

Nous avons également été amené à consulter des forums ou des réponses à différentes problématiques rencontrées, le plus souvent rédigées en anglais. (StackOverflow)

Un exemple concret concerna le téléchargement de fichiers vers le serveur. Nous nous sommes documentés sur les possibilités mais également les bonnes pratiques.

how to upload image from expo app to node js api multer

Vidéos Images Livres Maps Actualités Flights Finance

Environ 57 000 résultats (0,32 secondes)

Conseil : Limiter cette recherche aux résultats en **anglais**. En savoir plus sur le filtrage par langue

Stack Overflow
https://stackoverflow.com › questions · Traduire cette page

uploading image to nodejs backend (express multer) from ...
1 août 2022 — I have a backend and a create ticket route, in my web react app the image upload works fine but in react native it was not working.

How to upload react-native image from expo-image-picker to ... 10 avr. 2022
How to upload image to node js server from react native (expo ...) 25 août 2021
React native image upload to node server with multer 30 nov. 2020
File upload from React Native (expo) to Node (multer) 2 janv. 2021

Autres résultats sur stackoverflow.com

How to upload image to node js server from react native (expo) using fetch

Asked 1 year, 10 months ago Modified 1 year, 10 months ago Viewed 5k times

1 I am not receiving data on the server side. Sometimes I get data as an object. I have tried different ways but the image upload gets failed. Below is my code. I have tried different method but the result was the same image folder remain empty as am using multer middleware

Image is getting displayed using this snippet

```
<TouchableHighlight
  style={[
    styles.profileImgContainer,
    { borderColor: "#4632a1", borderWidth: 1 },
  ]}
  onPress={openImagePickerAsync}
>
  <Image source={{ uri: selectedImage.localUri }} style={styles.thumbnail} />
</TouchableHighlight>
```

This is the Image Picker Section

```
function PickImage() {
  let [selectedImage, setSelectedImage] = useState("");
  let openImagePickerAsync = async () => {
    let permissionResult =
      await ImagePicker.requestMediaLibraryPermissionsAsync();

    if (permissionResult.granted === false) {
      alert("Permission to access camera roll is required!");
      return;
    }
}
```

Sur cet exemple, l'utilisateur dit qu'il ne reçoit pas de données côté serveur. Parfois il en reçoit en tant qu'objet. Il a essayé de différentes manières mais le téléchargement d'images a échoué. Malgré les essais, le dossier d'image reste vide. Il utilise le middleware multer. Ensuite, il présente son code.

En dessous, une réponse parle de diskStorage() (que nous avons donc utilisé).

Il précise d'abord comment définir l'objet image à télécharger.

Avant de dire qu'il faut passer par un formData.

```
setSelectedImage({
  uri: result.uri,
  name: 'SomeImageName.jpg',
  type: 'image/jpg',
});
```

Enfin il montre l'exemple de code pour l'upload via le diskStorage()

The Overflow Blog

How Bloomberg's era of knowledge sharing

Making computer science at Carnegie Mellon (

Featured on Meta

Starting the Prompt Home in our Stack Exchange Neighborhood

Colors update: A major UI change

Temporary policy: Generating ChatGPT is banned

Testing native, sponsored by Stack Overflow (start)

Related

41 How can I upload a file to a node.js server using React Native?

2 Image upload in React Native using fetch results in 400 error

1 Unable to upload image to node.js server using react-native

12 - Compétences et axe d'amélioration

Tout au long du projet, nous avons pu acquérir et développer un certain nombre de compétences.

Dans un premier temps, un projet d'une telle ampleur nécessite une organisation rigoureuse.

Nous avons donc veillé à ce qu'elle soit respectée tout au long du projet. Workflow GitHub / Git, tickets Trello, nommage cohérent des fonctions / variables. Cela nous a permis de nous rapprocher de pratiques plus professionnalisautes.

Nous avons également développé notre communication au sein de l'équipe, que ce soit lors de la phase de réflexion (identité de l'entreprise, maquettage) ou bien lors du développement.

Les nombreuses documentations consultées nous ont également apporté des connaissances supplémentaires sur différents aspects.

Nous avons également acquis diverses compétences techniques durant tout le développement du projet.

Des connaissances / compétences back-end avec la mise en place de l'API, des retour de réponse - statuts, de la sécurisation de cette dernière...

Mais aussi des compétences front-end (UX / UI) avec le travail effectué en amont tant sur la phase de réflexion à propos de la navigation et de l'enchaînement des écrans, que sur le maquettage de l'application.

Un exemple que l'on peut citer concerne le choix d'avoir développé un menu de navigation en bas de l'écran, facilitant son accès à l'utilisateur qui utilisera son appareil probablement à une main.

Nous avons poussé la conception de l'application afin de nous rapprocher d'un rendu plus réaliste et augmenter nos compétences.

Avec du recul, nous réalisons que notre projet pourrait être amélioré sur divers points. Les ressources disponibles pour le développement s'enrichissent de jour en jour et il est possible qu'entre la production de deux applications, un nouvel outil plus performant ou simple à utiliser voit le jour.

Nous pourrions prendre l'exemple de Strapi, qui propose un grand nombre de fonctionnalités permettant d'interagir avec une base de données. Il prend le rôle d'une API et permet de gagner un certain temps comparé à une API entièrement conçue à la main.

Parmi les améliorations on peut noter l'optimisation des requêtes sql (pour un gros trafic comme pour une application de messagerie par exemple), mais également que le modèle de base de données choisie aurait pu être différent (NoSQL car plus adapté). Il serait intéressant de comparer avec le développement que nous avons réalisé.