



Réalisation de l'application web et mobile

—
NEKO

SOMMAIRE

Introduction	3
Présentation personnelle	3
Présentation du projet en anglais	3
Compétences couvertes par le projet	3
Organisation et cahier des charges	5
Présentation de l'entreprise	5
Les besoins client	5
Les fonctionnalités attendues	5
Application mobile	5
Application web	7
Contexte technique	8
Conception du projet	9
Choix de développement	9
Choix des langages	9
Choix des frameworks	9
Logiciels et autres outils	10
Organisation du projet	11
Architecture logicielle	12
Conception du front-end de l'application	13
Arborescence du projet	13
Charte graphique	14
Maquettage	15
Conception backend de l'application	16
La base de données	16
Mise en place de la base de données	16
Conception de la base de données	16
Modèle conceptuel de données	17
Modèle logique de données	17
Modèle physique de données	18
Développement du backend de l'application	20
Organisation	20
Arborescence	20
Fonctionnement de l'API	21
Middleware	22
Routage	22
Controller	24
Service	24

Model	24
Sécurité	27
Credential stuffing : vol du login et password.	27
Chiffrement des données sensibles	27
JWT	28
Gestion des Droits	28
Helmet	30
Exemple de Problématique rencontrée	31
Recherches anglophones	31
Exemple : Envois de données avec fichier image	31
Documentation	36
Tests	37
Postman	37
Newman	38
Reporter	39
Jest	39
Développement du front-end de l'application	40
Arborescence	40
Pages et composants	40
Sécurité	43
Problématiques rencontrées	43
Exemple navigation imbriquée :	44
Exemple de formulaire de mise à jour du profil	46
Conception de l'espace administrateur	48
Conception de la partie administration	48
User Story	48
Choix du langage et framework	49
Conception du frontend du site web	49
Charte graphique	49
Maquettage	49
Conception du back end du site web	49
Conclusion	50
ANNEXE	51

Introduction

Présentation personnelle

Je me nomme Khellaf RACHEDI, j'ai 28 ans. J'ai suivi le cursus Coding School de l'école La Plateforme sur Marseille en 2021-2022, en fin de première année, j'ai obtenu mon titre de développeur web et web mobile. Aujourd'hui en Coding School 2 je me prépare au passage du titre de concepteur développeur d'applications. Je suis en parallèle alternant au sein de l'entreprise CMA CGM en tant que développeur low code depuis septembre 2022.

Présentation du projet en anglais

As part of my training at La Plateforme_, I worked with several colleagues to create a mobile chat application for a fictitious company.

The aim of the application was to create an instant exchange platform that would enable the company's employees to register and then log in to communicate with each other easily and securely. In fact, the application is entirely dedicated to a single company, so that a certain conviviality is established during exchanges.

My target audience was therefore made up of several internal employees in need of a chat-based means of communication.

For the development of this application, I opted for the React Native framework for the front-end. In addition, to develop the API, I used Node.js in combination with Express.js. For the admin panel, I used the Vue.js framework.

The application has a registration page for new users of the application and a login page for those who already have an account.

Once logged in, users can access their profile, modify it and add a profile picture. They can also view the profiles of other users.

The application's main feature is real-time exchange via a collective chat.

In addition, a web interface has been developed to enable each user administrator to manage the application. The administrator has special rights to :

- Delete users.
 - Modify the status of certain users to promote them as administrators.
 - Delete and censor messages.
-

Compétences couvertes par le projet

Ce projet couvre les compétences du titre suivantes :

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

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 gratuite de chat, à des particuliers dans le but de pouvoir échanger librement sur différents sujets. 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 pages 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 Node Js 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 créent de nombreuses bibliothèques de code open source facilitant ainsi le développement de certaines fonctionnalités. Il est présent dans toutes les applications web mais aussi mobiles. Il n'existe à ce jour plus aucune page web qui n'utilise pas cette technologie pour dynamiser son contenu.

Choix des frameworks

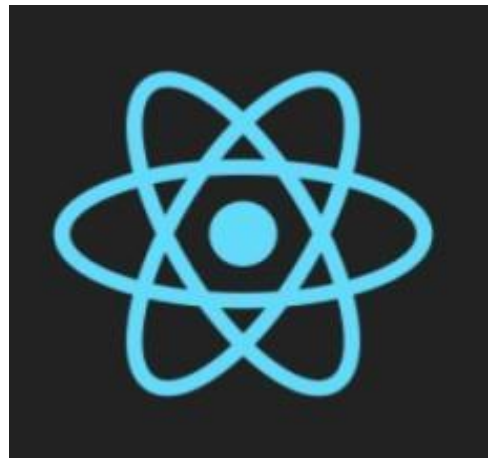
Pour la création de mon API j'ai choisi d'utiliser Expressjs qui est un framework de NodeJs.

expressJs est le framework le plus populaire pour NodeJs. C' est un framework minimaliste permettant de garder un certain contrôle dans le développement du projet et apporte

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 le framework react native car il est écrit en javascript et qu'il permet le développement d'un seul code pour les plateformes iOS et



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;
- Figma 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.

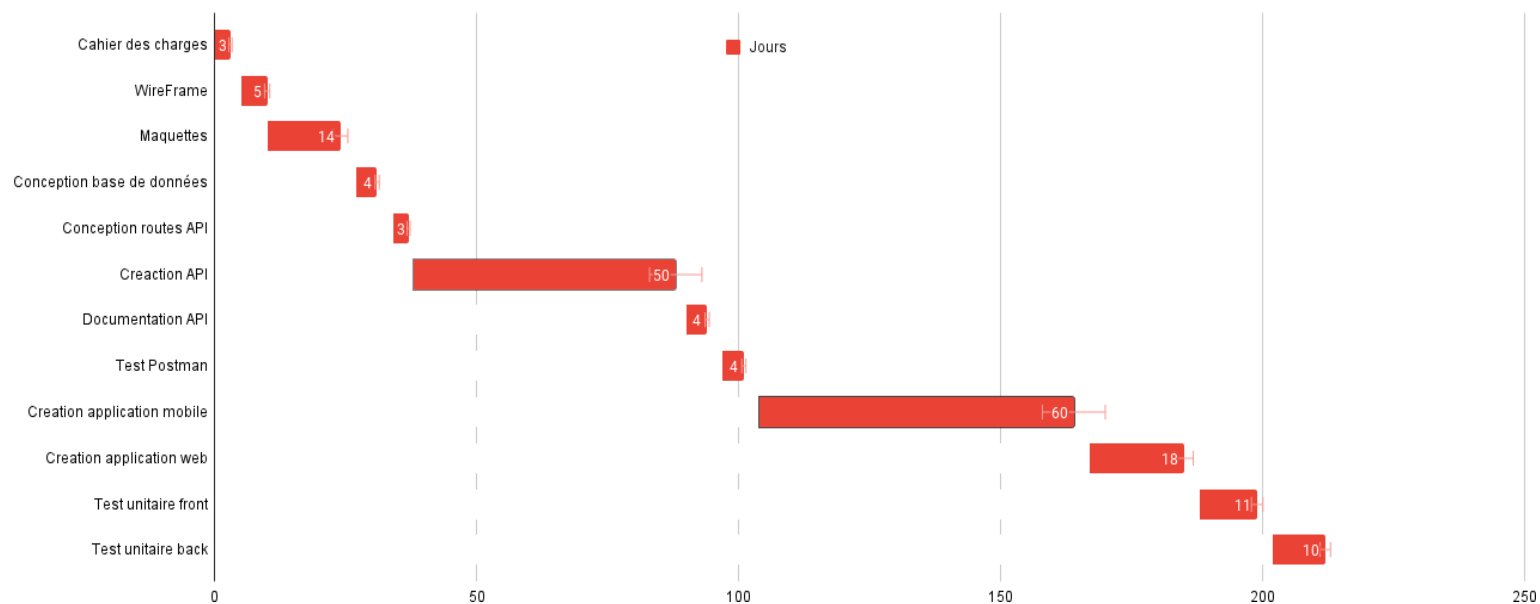
Organisation du projet

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.

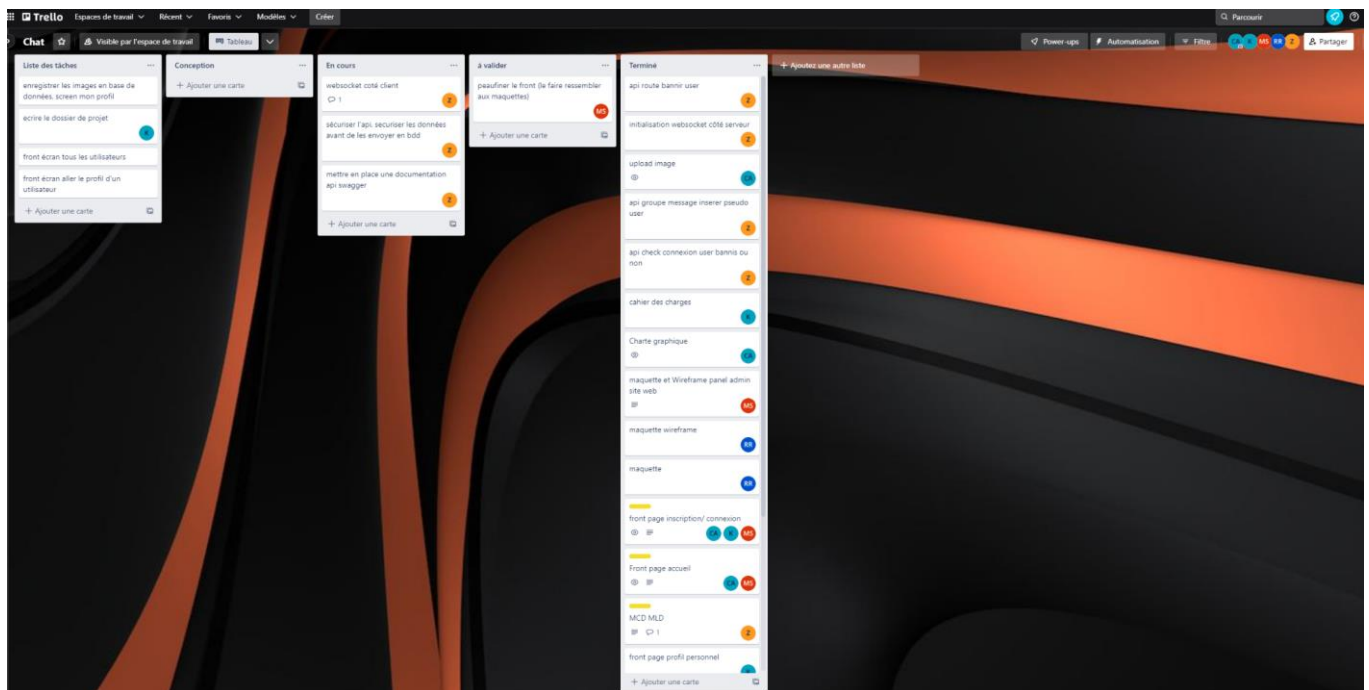
Premièrement, j'ai tout d'abord créé un diagramme de gantt.

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.

Diagramme de GANTT

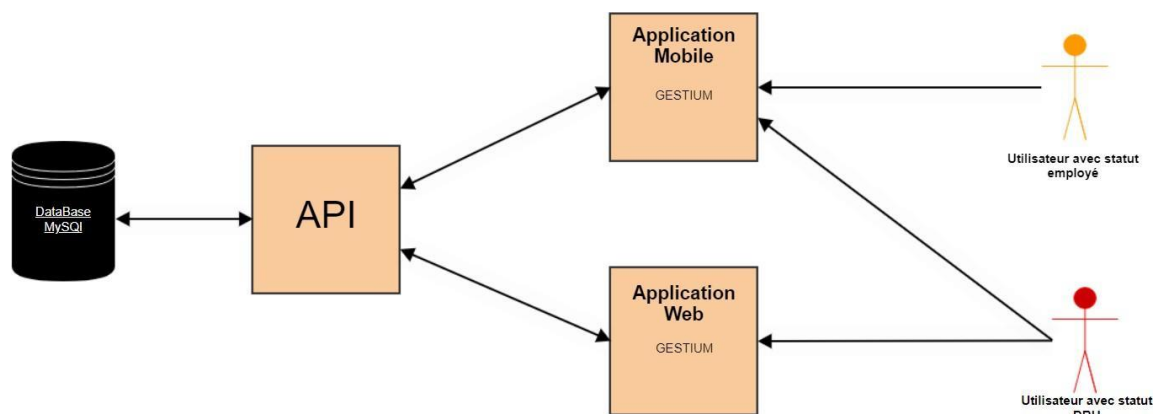


On a utilisé des outils comme trello afin de lister les tâches à effectuer et les attribuer à un membre.



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. L'API communique avec ma base de données MySQL.

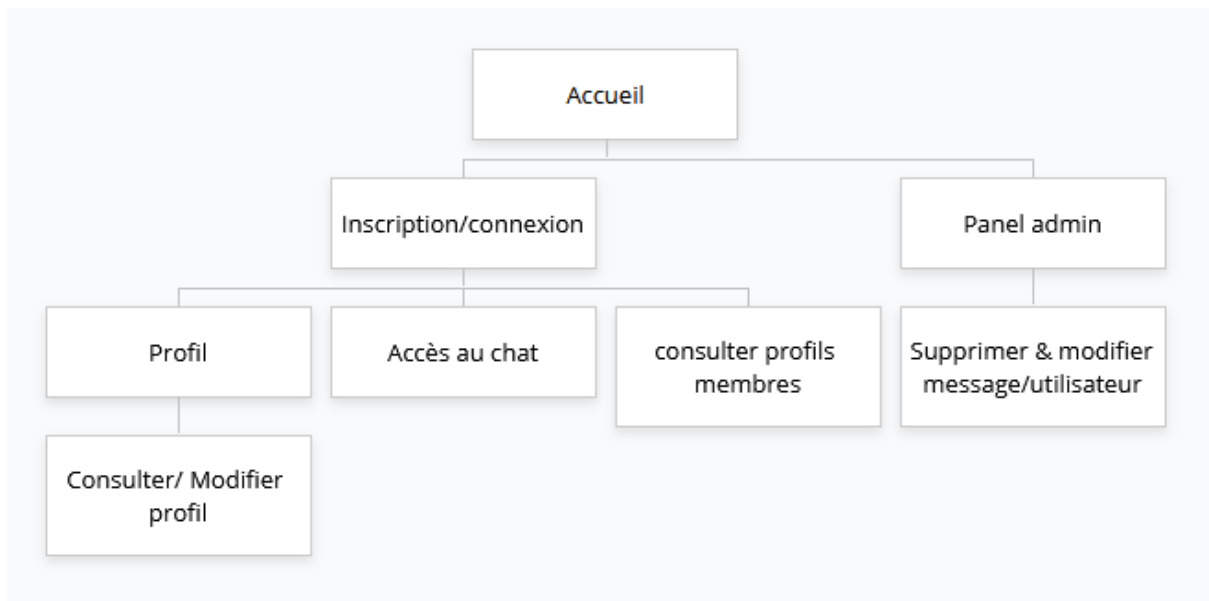


Schémas de l'architecture logiciel

Conception du front-end de l'application

Pour créer les maquettes 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 :



ffc13b



f5f0e1



ff6e40



1e3d59

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.

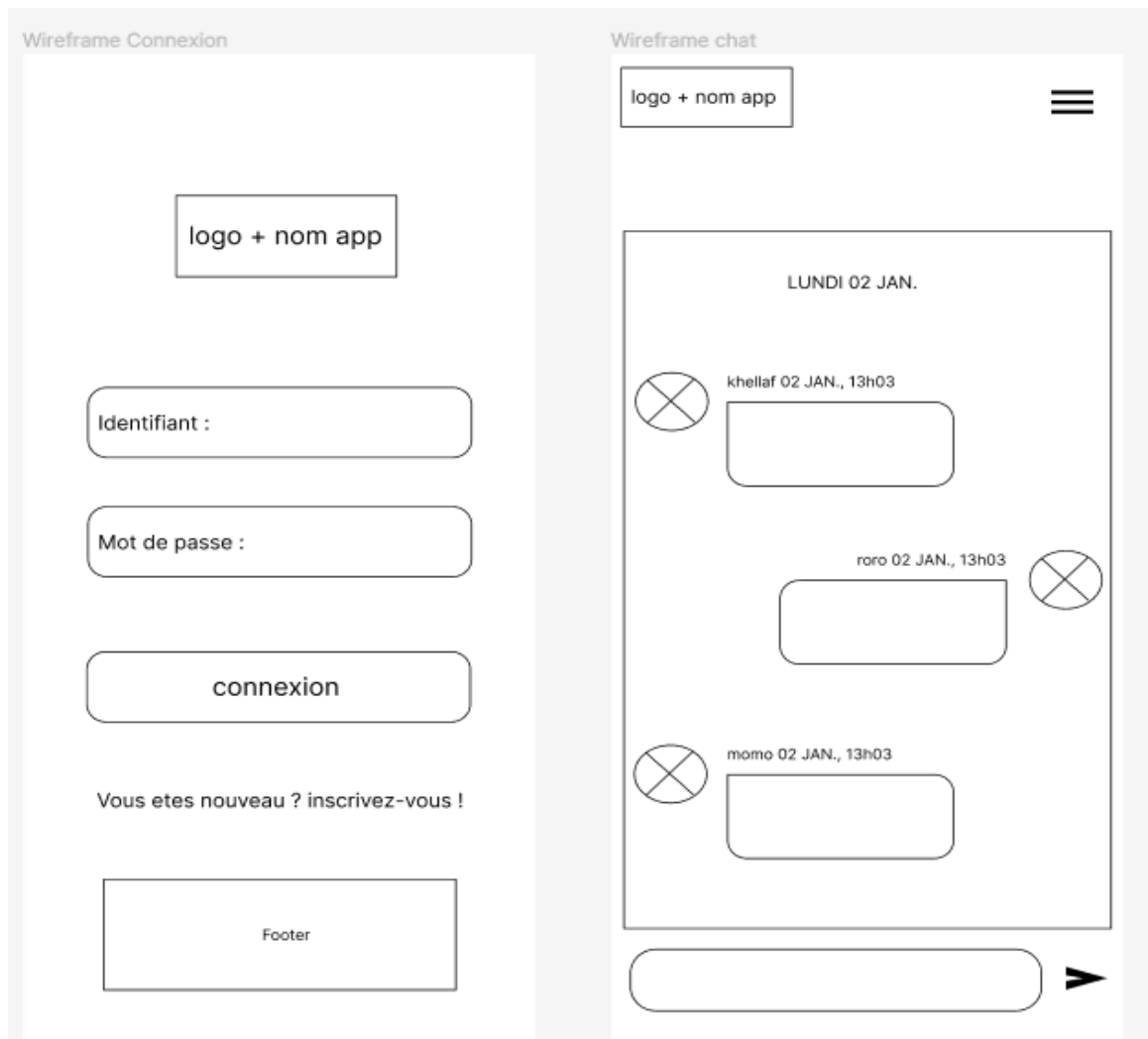
Maquettes :

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) :



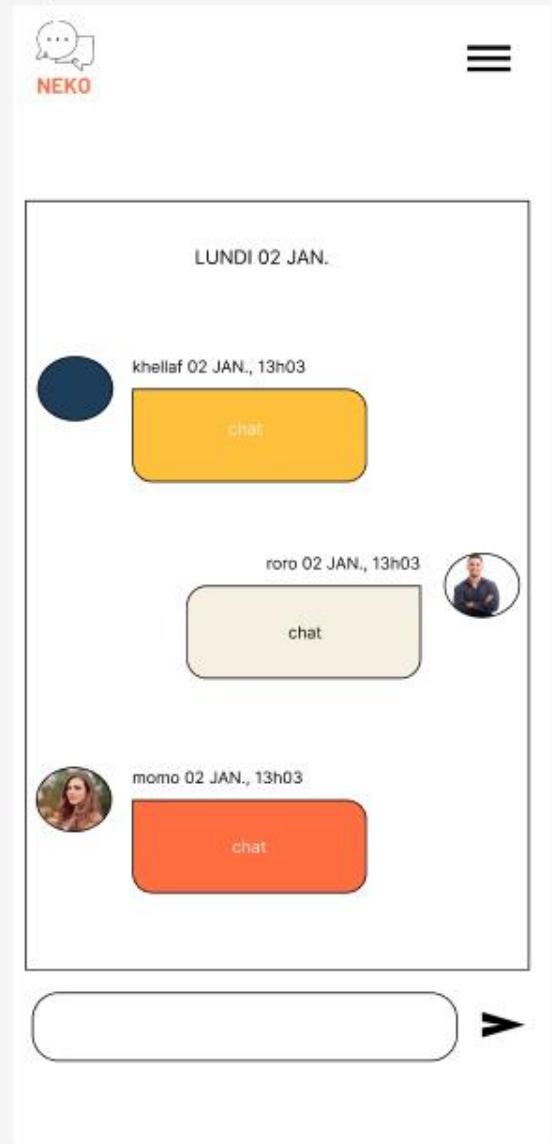
Par la suite, des maquettes **Design** ont été réalisées afin de mettre en avant une identité visuelle de l'application. Le but étant que l'utilisateur puisse trouver l'application attrayante et agréable car la première impression qu'un utilisateur fait sur un site est la partie front-end.

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

iPhone 14 - 2



Maquette 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

Flexible et peut associer & stocker plusieurs types de données

Stock et gère des **volumes de données** plus importants par rapport à une BDDR

Conception de la base de données

Pour concevoir ma base de données NoSql j'ai utilisé la méthode Merise. Dans un premier temps , j'ai fait un recueil 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 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.

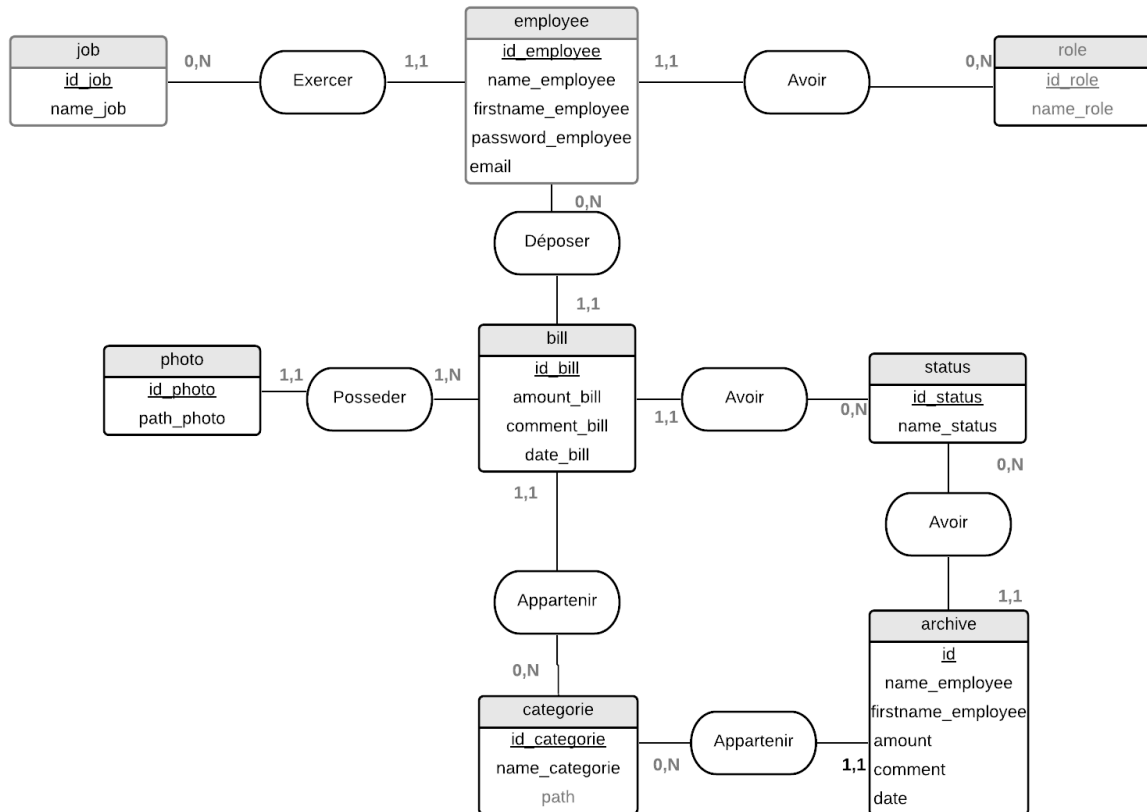
Voici un aperçu des données pertinentes qui m'ont aidé à construire une représentation claire des besoins:

- Pour les notes de frais : j'ai besoin de stocker le montant , la description , un statut , une date, sa catégorie et une photo.
- Pour les employés : j'ai besoin de stocker leurs noms , leurs prénoms, un mot de passe, un email, le poste occupé et son statut.
- Pour les archives : j'ai besoin du nom du salarié, son prénom , le montant de la note de frais, le montant , la description de la note de frais, un statut et une date.

J'ai construis ma base de données , en procédant en trois grandes étapes:

Modèle conceptuel de données

La première étape a été la création du modèle conceptuel des données. J'ai créé des entités en fonction du dictionnaire de données récoltés. J'ai donc dessiné dans un premier temps mes entités en leur donnant un nom.



Modèle logique de données

J'ai ensuite continué la création de ma maquette de base de données en créant le modèle logique de données.

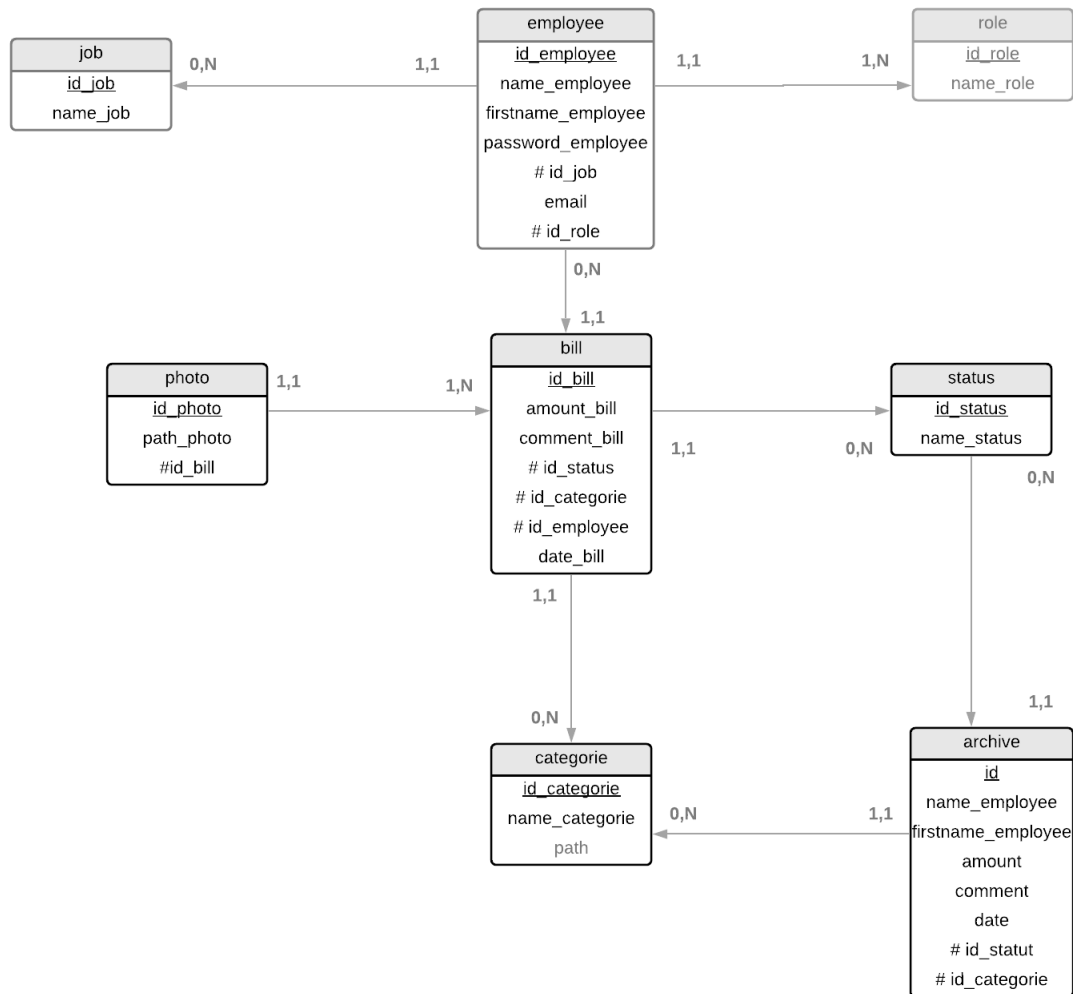
J'ai créé une clé primaire pour chaque entité qui permet d'identifier sans ambiguïté chaque occurrence de cette entité.

Ensuite, j'ai rempli chaque entité avec des attributs en reprenant les informations de mon recueil de données.

J'ai procédé à la création des cardinalités de chaque entité.

Exemple :

un employé peut avoir au minimum 0 bill ou au maximum N bill et un bill peut avoir au minimum 1 employée et au maximum 1 bill.



Modèle physique de données

Pour cette dernière étape , j'ai analysé les relations entre mes tables. Dans ce projet , je n'ai pas de cardinalité de type N/N , je n'ai donc pas de table de liaison. J'ai ensuite créé les clés étrangères. La clé étrangère est une contrainte qui garantit l'intégrité référentielle entre deux tables. J'ai ensuite typ

Développement du backend de l'application

Organisation

Mon back end a pour but d'être utilisé à la fois pour mon application web et mon application mobile. J'ai décidé de créer une API afin de ne pas coder deux fois ma logique métier.

Dans le but de rendre mon backend plus efficace , je me suis concentré sur la logique et l'optimisation de mon code.

J'ai donc fait des recherches dans ce sens.

Je divise donc mes programmes en différents modules, ainsi cela augmente la lisibilité du code et devient plus facile à maintenir pour les prochaines versions.

J'évite les répétitions en créant des fonctions et des services.

La logique de mon code est divisé en services et fichiers

Arborescence

J'ai suivi une architecture N-tier. Le principe de cette architecture est la séparation des préoccupations pour éloigner la logique métier des routes de l'API.

Les différentes couches de l'application sont séparées :

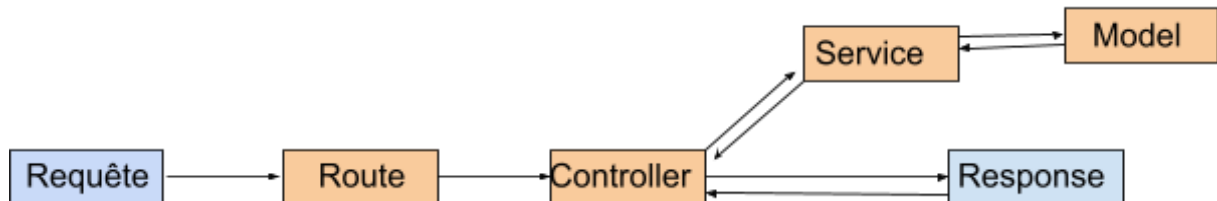
- Le routeur,
- La couche applicative qui est divisé deux couches :
 - controller,
 - services,
- La couche data (les modèles).

Mon back end est donc composer des dossiers suivants :

- **Routes** : regroupant tous les fichiers de mes routes (un fichier par CRUD d'une table de base de données)
 - **Controllers** : Regroupant tous les controller (un par route)
 - **Fonctions** : regroupant tous les fichiers contenant les fonctions
 - **Middleware** : contenant des sous dossier (un par route) contenant les fichiers avec les middlewares des routes
 - **model** : contenant les modèles de toutes mes tables (un par table et par fichier)
 - **services** : contenant des sous- dossier (un par table)
 - **test** : destiné aux test unitaire - Newman
 - **uploads** : images upload
-

Fonctionnement de l'API

Lorsque le client envoie une requête sur mon API, le routeur analyse l'URL, en fonction de la route et de la méthode un controller est appelé. Ce contrôleur va faire appel à un service qui va communiquer avec le modèle afin de récupérer des données. Ensuite, ses données sont analysées par le service puis une réponse est envoyée au format JSON avec un code statut.



Architecture de l'API

Les différents statuts utilisés dans se projets 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
- **PATCH** - Pour mettre à jour partiellement une donnée
- **DELETE** - Pour supprimer une donnée

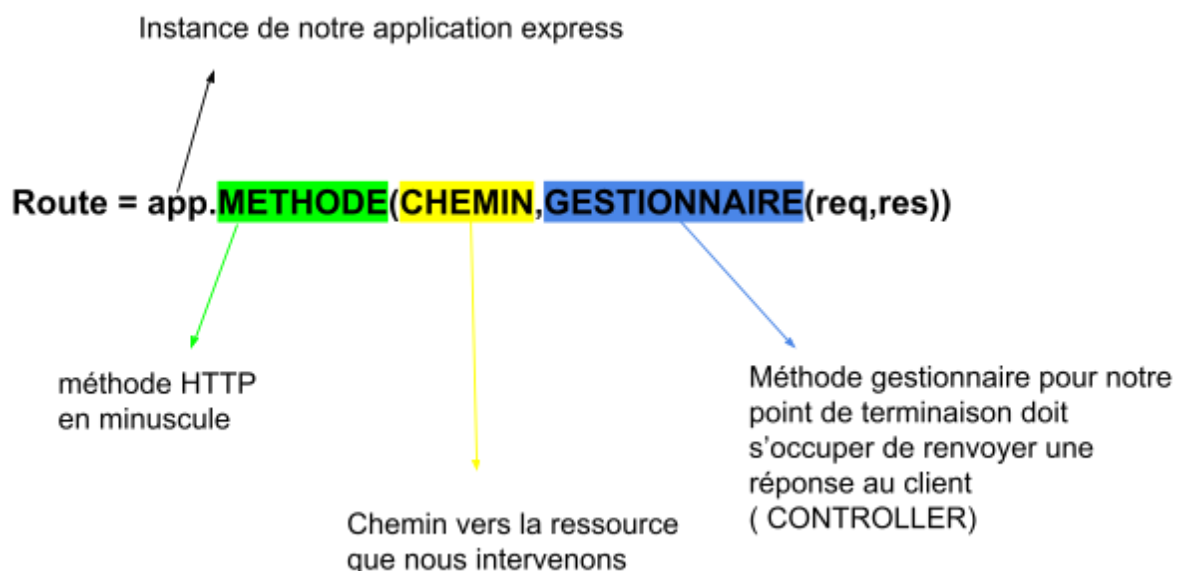
Middleware

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

Routing

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 expliqué plus haut mon but a été de rendre mon code facilement modifiable et maintenable j' ai donc diviser cette partie .

La première chose que j'ai faite a été la séparation des routes et le code d'implémentation de celles-ci.

```
// ROUTES
app.use('/v1/employees', employeeRoutes)
app.use('/v1/job', jobRoutes)
app.use('/v1/bill', billRoutes)
app.use('/v1/category',categoryRoutes)
app.use('/v1/status',statusRoutes)
app.use('/v1/login', connexionRoutes)
```

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

Comme on peut le voir dans cet exemple, si l'URL '/v1/employees' est appelée , mon routeur employeeRoutes est utilisé.

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

Dans les routeurs une classe express.Router est instancié. C'est un middleware niveau routeur.

Un middleware niveau routeur fonctionne de la même manière qu'un middleware. Grâce à la méthode route() de celui-ci le schéma de la route est défini. Ainsi je peux utiliser les méthode get , post, put, patch et delete afin de pouvoir effectuer différentes actions en fonction de la méthode choisie.

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

router.get('/',AuthenticateJWT, async
(req,res)=>{ try {
    if (req.user.role === 2){
        let result = await
EmployeeController.findAll()
res.setHeader('Content-Type', 'application/json')
SendResponse(result,200,res)
    }else{
        res.status(401).json({error : "Droits
insuffisant"})
    }
}
```

```
        res.status(500).json(error)
    }
});
```

Dans l'exemple ci-dessus, j'ai instancié la classe Router de express, grâce à cette classe, je peux faire appel aux différentes méthodes HTTP.

Comme on peut le voir, la méthode utilisée est le get.

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

Controller

Dans les controllers on y trouve uniquement les tâches de validations comme par exemple la validation des droits. Aucune logique ne se trouve dans le contrôleur.

Service

Cette couche est l'endroit où se trouve toute ma logique. L'objet request et response ne sont pas envoyés au service, seuls les paramètres de la requête sont envoyés après vérification du contrôleur.

Model

Dans les modèles on y trouve la connexion à la base de données mais aussi toutes requêtes SQL.

Chaque table possède un fichier contenant une classe avec des méthodes qui agissent sur celle-ci.

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

const db = mysql.createConnection({

    host:
    "localhost", user:
    "root",
    password: "",
```

```
});

db.connect(function(
err) { if (err)
throw err;
    console.log("Connecté à la base de données MySQL!");
});

module.exports = db
```

Dans cet exemple j'utilise un paquet npm 'mysql' qui permet de lier mon serveur Node.js à MySQL et ainsi pouvoir exécuter des requêtes. Il existe plusieurs paquets permettant une connexion à une base de données, j'ai choisi mysql. C'est la solution la plus simple et rapide à mettre en place pour interagir avec une base de données MySQL en nodeJs.

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

```
npm install mysql
```

Ensuite pour se connecter à la base, il faut dans un premier temps initialiser le module. Ensuite nous avons accès à des fonctions.

La première fonction CreateConnection() permet d'indiquer l'host, le login de la bdd, le mot de passe et le nom de la base.

Enfin, j'utilise la fonction connect pour me connecter à la base de données, une exception sera envoyée en cas d'erreur.

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

const db = mysql.createConnection({

    host:
    "localhost", user:
    "root",
    password: "",
    database : "api_node"
});

db.connect(function(er
r) { if (err) throw
err;
    console.log("Connecté à la base de données MySQL!");
});
```

Afin d'éviter les répétitions , j'ai créé une classe dbModel qui permet la connexion à la base de données.

```
class models{  
  
    constructor(table)  
    {  
        this.table = table  
        this.db = require('../models/dbConfig')  
    }  
}
```

Comme expliqué précédemment , chaque table possède une classe et celle-ci va s'étendre de la classe bdModel .

```
const Models = require('../models/Model')  
  
class StatusModel extends  
Models{ constructor(){  
    super()  
}  
  
findAll(){  
    return new Promise((resolve,reject) => {  
        this.db.query(`SELECT * FROM status` ,  
(error,result) => { if (error) {  
            return reject(error)  
        }  
        else{  
            return resolve(result)  
        }  
    })  
    })  
}
```

Dans cet exemple , la classe `StatusModel` s'étend de la classe `Models` , permettant ainsi de récupérer la connexion a la base de données qui est stockée dans la propriété `db` de la classe `models`.

La connexion est ouverte à l'exécution de la première requête.

La méthode `query` prend en paramètre une chaîne SQL et un tableau facultatif de paramètres qui seront transmis à la requête.

Cette fonction renvoie une `Promise Object`. La promesse sera résolue lorsque l'exécution de la requête sera terminée. Le résultat sera renvoyé si tout c' est bien passé sinon en cas d'erreur la promesse sera rejetée.

Une promesse en javascript est une complétion ou un échec d'une opération asynchrone.

Sécurité

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

Les différents risques sont :

- les injections SQL : Les injections consistent à insérer dans les codes un programme un autre programme malveillant permettant ainsi d'attaquer directement une base de données et y prendre le contrôle. L'attaquant peut alors se servir librement de toutes les informations récoltées.
- Credential stuffing : vol du login et password.
- Attaque DDOS : envoie de trafic en vue de surcharger le trafic d'une API
- Man-in-the-middle : consiste à pousser un utilisateur à se connecter a un service compromis, qui pourra alors s'emparer du jeton ou de la clés de l'utilisateur.

Pour rendre mon API sécurisé j'ai mis en place plusieurs actions que je vous décrirais dans ce chapitre.

Chiffrement des données sensibles

Les mots de passe des utilisateurs ne sont pas stockés en dur dans la base de données. Pour hacher des mots de passe , j'utilise `bcrypt`.

JWT

Dans le but d'effectuer une authentification sécurisée sur mon projet ainsi que stateless , j'utilise le Jeton Web Token.

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

Ce jeton est composé de trois parties:

- Un header : identifie quel algorithme a été utilisé pour générer la signature
- un payload : est la partie qui contient les informations de l'utilisateur, sous forme de chaîne de caractères hashé en base 64.
Pour des mesures de sécurité , je n'insère aucunes données sensibles telles que des mots de passe ou des informations personnellement identifiables.
- la signature : Elle est créée à partir du header et du payload générés et d'un secret. Une signature invalide implique systématiquement le rejet du token. La signature du jeton a une importance fondamentale , il sert à vérifier que les informations connues sont inchangées.

Lorsqu'un utilisateur essaie de se connecter à son espace , une demande est envoyée au serveur. Si les informations envoyées sont correctes , le serveur renvoie une réponse sous forme de JSON dans le quel s'y trouve le jeton . Celui-ci contient des informations concernant la personne connectée (son id , son mail et son rôle). Le client enverra ce jeton avec toutes les demandes qui suivront. Ainsi, le serveur n'aura pas à stocker d'informations sur la session.

Gestion des Droits

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

```
const jwt = require('jsonwebtoken');
const accessTokenSecret = 'youraccesstokensecret'

const AuthenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (authHeader) {
        const token = authHeader.split(' ')[1];

        jwt.verify(token, accessTokenSecret,
            (err, user) => { if (err) {
```

```

        return res.sendStatus(403);
    }
    else{
        r
        eq.user =
        user; next();
    }
    })
}else {
    res.sendStatus(401);
}
}

```

Dans ce middleware , j'ai utilisé un module jsonWebToken qui est utilisé pour générer et vérifier un jwt.

Le constante accessTokenSecret contient le secret pour signer le JWT. Ce code ne doit jamais être partagé . Il est important que ce secret soit complexe pour que pour que l'application soit le plus sécurisé.

Ensuite , je vérifie si le token est valable et je récupère toutes les données contenues dedans. Je renvoie les informations du user dans la requête vers ma route.

Par exemple , pour la route 'v1/employees/{id}', qui permet la suppression d'un salarié. Cette action est réservée aux DRH.

Pour pouvoir accéder à cette route , il faut avoir un rôle 2 qui correspond au statut DRH.

Ainsi a chaque appel de cette route le middleware de la route récupère le JWT , le vérifie et récupère les informations lié à l'utilisateur.

```

router.delete('/:id',AuthenticateJWT, async (req,res,next)
=> { try {
    if (req.user.role === 2){
        let result = await
EmployeeController.delete(req.params.id)
        SendResponse(result,204,res)
    }else{
        res.status(401).json(error)
    }
} catch (error) {

```

```
}  
})
```

Ici , si le rôle récupéré est 2 je peux avoir accès au controller sinon , une réponse de type 403 est renvoyée avec un message d' erreur.

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.

Exemple de Problématique rencontrée

Durant le développement de mon API , j'ai dû tester le fonctionnement de mes routes avec Postman.

J'ai été confronté a un erreur lors de l'appel des mes routes côté front:

```
✖ Access to fetch at 'http://192.168.1.6:8080/v1/login' from origin 'http://localhost:19006' has been blocked by CORS policy: Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. If an opaque response serves your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
```

Pour résoudre ce problème , j'ai dû comprendre d'où venait cette erreur.

Le CORS est un protocole de sécurité qui s'applique aux navigateurs. Son but ? Bloquer l'accès à vos ressources depuis une origine qui n'est pas la vôtre si vous ne l'avez pas autorisée spécifiquement.

Recherches anglophones

Pour résoudre le problème cité juste au dessus , j'ai parcouru stack overflow mais aussi la document d'express qui est en anglais.

The screenshot shows the Express.js website with a black header containing the text "Black Lives Matter. Support the Equal Justice Initiative." Below the header is a navigation bar with links: Home, Getting started, Guide, API reference, Advanced topics, and Resources. The main content area is titled "cors" and includes a note that the page was generated from the cors README. It features a sidebar with a list of modules: body-parser, compression, connect-redis, cookie-parser, cookie-session, cors, csrf, errorhandler, method-override, morgan, multer, response-time, serve-favicon, serve-index, serve-static, session, timeout, and vhost. The main content area for "cors" includes a note about the package being generated from the README, a list of npm statistics (v2.8.5, 75M/mo, CI passing, coverage 100%), a description of CORS as a node.js package for providing a Connect/Express middleware, and a list of links for installation, usage, configuration options, demo, license, and author. The installation section states that this is a Node.js module available through the npm registry and that installation is done using the npm install command.

Exemple : Envois de données avec fichier image

Dans cette exemple, je vous montre comment j'ai procédé pour la création d'une demande de remboursement de note de frais :

```

router.post('/create',AuthenticateJWT,middlewareUpload.upload.single(
'i mageBill') , middleware.middlewareBill(),async (req,res) => {

    try {
        let image = req.file.filename;
        let errors =
validationResult(req); let result
= await
BillController.add(errors,req.body.amount,req.body.comment,req.body.i
d_ categorie,req.body.id_employee,image);
        res.setHeader('Content-Type',
'multipart/form-data') res.setHeader('Content-
Type', 'application/json')
        if (result.error){
            SendResponse(result,400,re
s)
        }else{
            res.location(resu
lt[0].uri)
            SendResponse(result[0],201,re
s)
        }
    }
}

```

Comme on peut le voir , la route /create permet la création d'une demande de remboursement de note de frais.

La méthode utilisée est la méthode POST. Trois middlewares ont été utilisés , AuthenticateJWT, single et middlewareBill.

Middleware AuthenticateJWT :

```

const jwt = require('jsonwebtoken');
const accessTokenSecret = 'youraccesstokensecret'

const AuthenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (authHeader) {
        const token = authHeader.split(' ')[1];
        jwt.verify(token, accessTokenSecret, (err,user) => {
            if (err) {
                return res.sendStatus(403);
            }
            else{

```

```

                                r
    eq.user =
      user; next();
    }
  })
} else {
  res.sendStatus(401);
}
}

```

Ce middleware permet de vérifier si le JWT est correct. S'il est correct, les informations de l'utilisateur sont stockées dans user puis envoyées au contrôleur, sinon une réponse 403 est envoyée.

Middleware upload et single:

```

const upload =
multer({ storage:
storage , limits: {
//5 megabytes
fileSize: 1024 * 1024 * 100
},
fileFilter: filter
})

```

Permet de vérifier la taille de la photo envoyée.

Single est une fonction incluse dans le module multer qui me permet de gérer tout ce qui concerne l'envoi de fichier (le nom, les extensions ...).

Middleware middlewareBill :

```

const { body } = require('express-validator');

function middlewareBill() {
  return [ body('amount').isDecimal().withMessage('Le montant de la
facture doit être un chiffre.') ,
    body('comment').isLength({min:10, max: undefined}).withMessage("La
description doit faire au minimum 10 caractères."),
    body('id_categorie').isInt().withMessage("La catégorie doit être
sélectionnée."),
  ]
}

```

Permet la vérification des données envoyées dans le body de la requête. Body est un middleware inclus à express qui encapsule les fonctions de validation et de désinfection des données.

Je stocke dans errors les éventuelles erreurs des données du body grâce au middleware.

Ensuite , je fais appel au controller BillController et j'appelle la méthode Add .

`async add(errors,amount,comment,categorie,employee,path){ if`

```
(!errors.isEmpty()) {  
  
    return { error: errors.array() };  
}else {  
    //verif id categorie  
    let categoryExist = false  
    let employeeExist = false  
  
    let idCategoryExist = await  
FindOneService(this.CategoryModel,categorie,messages)  
    if (idCategoryExist.error){  
        categoryExist = false  
    }else {  
        categoryExist = true  
    }  
    //verif id employee  
    let idEmployeeExist = await  
FindOneService(this.EmployeeModel,employee,messages)  
    if (idEmployeeExist.error){  
        employeeExist = false  
    }else {  
        employeeExist = true  
    }  
  
    if (categoryExist === true && employeeExist === true){ let result = await  
AddBillService(this.BillModel,amount,comment,categorie,employee)  
  
        if (result !== false){  
            const id_bill = result.insertId; await  
AddPictureService(this.PictureModel,path,id_bill)  
            return [{success: true , uri: URI+id_bill }]  
        }else {  
            return {error : "une erreur est survenue !"}  
        }  
    }else {
```

```

                if (categoryExist === false){
                    return {error : 'Identifiant catégorie
: '+messages.dataError(categorie)}
                }else {
                    return { error: 'Identifiant salarié
: '+messages.dataError(employee)}
                }
            }
        }
    }
}

```

Add est une méthode asynchrone de la classe ControllerBill.

Ce contrôleur permet la vérification des données qui lui sont envoyées.

Si aucune erreur n'est détectée , le contrôleur fait appel à un service comme par exemple le service AddBillService.

Service addBillService :

```

const AddBillService = async
(BillModel,amount,comment,categorie,employee) => {
    let result = await
BillModel.add(amount,comment,categorie,employee)
        if (result.affectedRows
=== 1){ return result;
        }else {
            return false
        }
    }

module.exports = { AddBillService }

```

Ce service a pour but la communication avec la base de données.

Les requêtes avec la base de données sont gérées par une classe BillModel comme on peut le voir dans cet exemple.

Model add:

```

add(amount,comment,categorie,employee) {
    return new Promise((resolve,reject) => {

```

```
        this.db.query( `INSERT INTO bill ( amount_bill,
comment_bill, id_categorie, id_employee,date_bill) VALUES
(?,?,?,?,NOW())` ,
[amount,comment,categorie,employee], (error,result)
=> {

                                if(error){
                                    return reject(error)
                                }else {
                                    return resolve(result)
                                }
                            })
                        })
                    })
```

Add est une méthode de la classe BillModel, c'est une méthode permettant d'exécuter une requête. Elle est chargée d'enregistrer des informations , si tout c'est bien passé le résultat est renvoyé sinon une erreur est retournée , tout cela géré par une promesse.

Mon code étant séparé en responsabilité unique , pour pouvoir récupérer la donnée, j'ai dû utiliser les promesses.

Et pour finir , la réponse est envoyée au client grâce à un service. Son rôle est d'envoyer un code http en fonction de la réussite de la requête ou non. Dans cet exemple , si tout c'est bien passé le code 201 est envoyé avec un message de succès sinon un code 400 indiquant qu'un problème s'est produit.

J'ajoute un header à ma requête. Le multipart/form-data est un type d'encodage qui permet d'envoyer des fichiers.

Si un problème serveur intervient, une erreur 500 est envoyée.

Documentation

Le but d'une API est d'être utilisé par d'autres développeurs. Ainsi il est important d'avoir une documentation.

La documentation fait référence aux contenus techniques avec des instructions claires sur le fonctionnement de celle-ci.

Il est nécessaire qu'elle soit mise à jour quand le code évolue ou que d'autres fonctionnalités sont ajoutées.

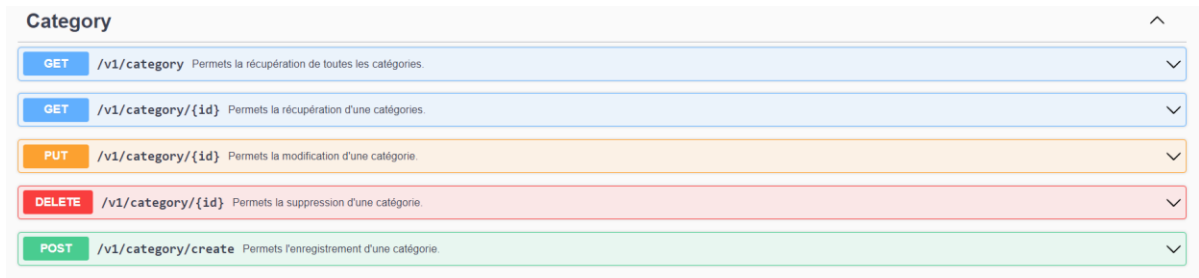
La documentation de l'API est écrite avec SWAGGER. Swagger est un ensemble d'outils permettant d'aider les développeurs dans la conception , le build, la documentation et la consommation d'API.

Swagger offre de nombreux avantages :

- la génération automatique de la documentation à partir du code;
- tout changement dans le code met à jour la documentations;
- un gain de temps.

Grâce à Swagger , j'ai pu avoir une interface graphique, Swagger UI, permettant de visualiser et d'interagir avec l'API. Cette interface permet aussi de tester son API.

Voici un exemple illustrant mes propos :

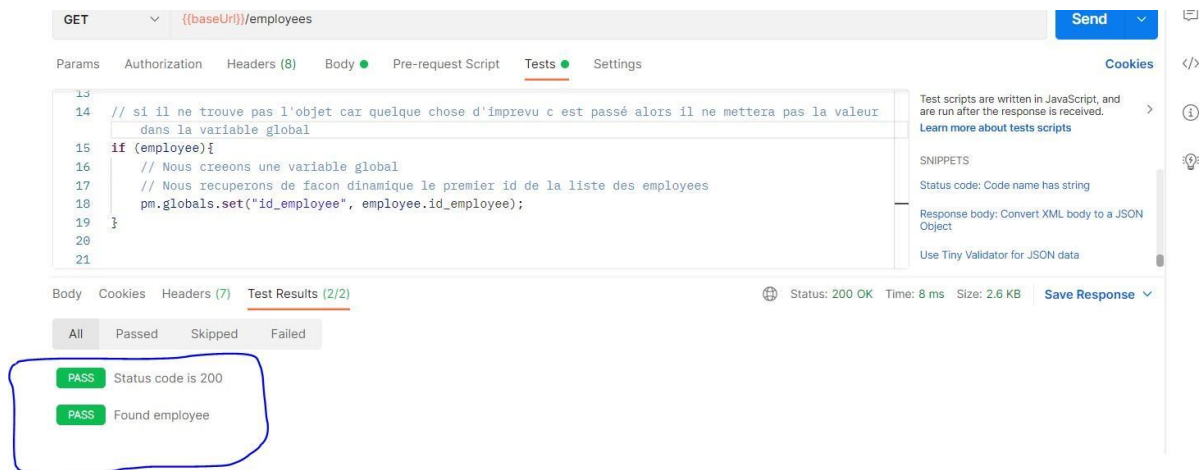


Tests

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 gérer.

Postman m'a aussi permis de créer des tests d'intégration.

Les tests sont exécutés à chaque fois que je lance la requête ainsi j'ai un visuel de tout ce qu'il se passe.

La syntaxe d'écriture des tests avec Postman est très verbeuse et intuitive. Elle ressemble beaucoup aux bibliothèques d'assertion communément utilisées en javascript. On peut facilement tester le statut de la réponse, on peut aussi tester les données reçues.

J'ai créé des variables d'environnement qui sont stockées et utilisées entre chaque requête.

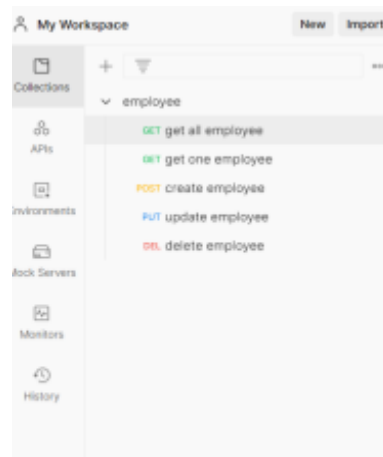
Newman

J'ai décidé d'automatiser les tests afin de pouvoir repérer plus rapidement les éventuels bugs. Cela me permet de pouvoir rajouter des fonctionnalités et d'exécuter les tests et voir si j'ai cassé quelque chose ou pas.

J'ai décidé d'utiliser Newman.

Newman est un utilitaire en ligne de commande qui permet d'exécuter des collections Postman.

Pour tester son api avec newman , il faut dans un premier temps importer la collection créer dans postman (fichier Json), il faut importer toutes les variable d'environnement créer dans postman. Puis l'exécution se fait en ligne de commande et on obtient dans la console différents résultats.

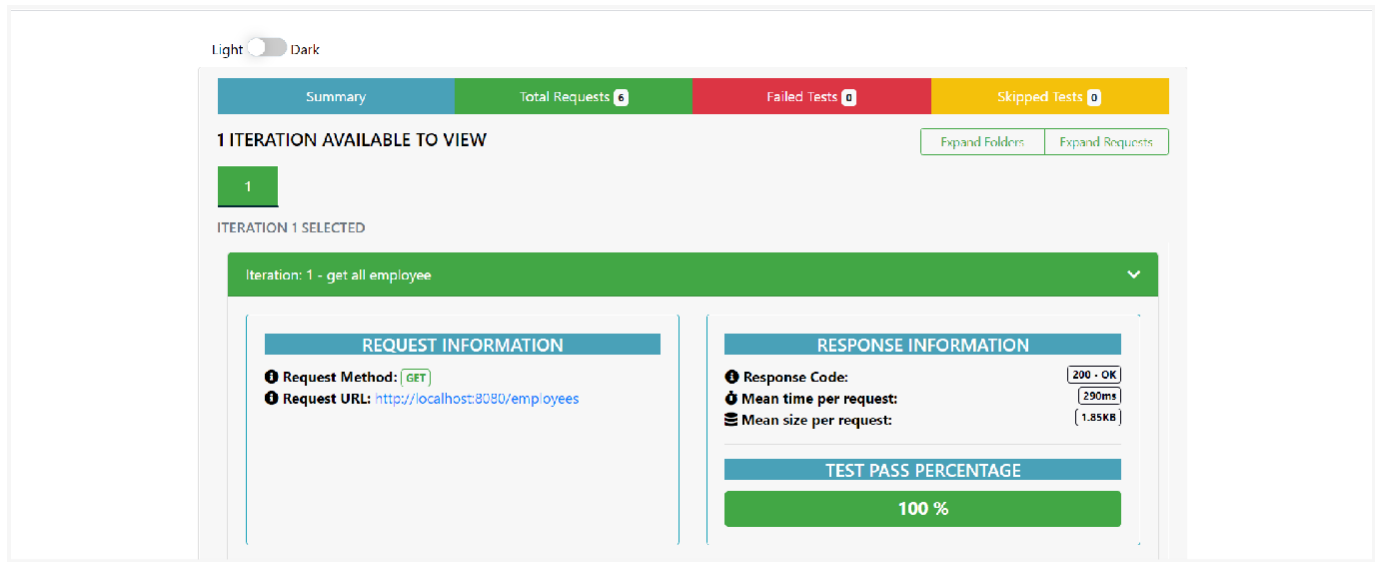


Collection Postman

Un résultat individuel de chaque requête et un résultat global avec notamment le temps d'exécution (très pratique pour effectuer des tests de performance dans le temps).

Reporter

Dans le but d'avoir un résultat plus facile à lire et à interpréter , j'ai décidé d'utiliser l'outil reporter de newman permettant à chaque lancement de test d'avoir un fichier HTML généré et ainsi permettre une interprétation plus simple des résultats.



Jest

Mon projet étant complètement écrit en javascript, il est important d'avoir une couverture de test suffisante.

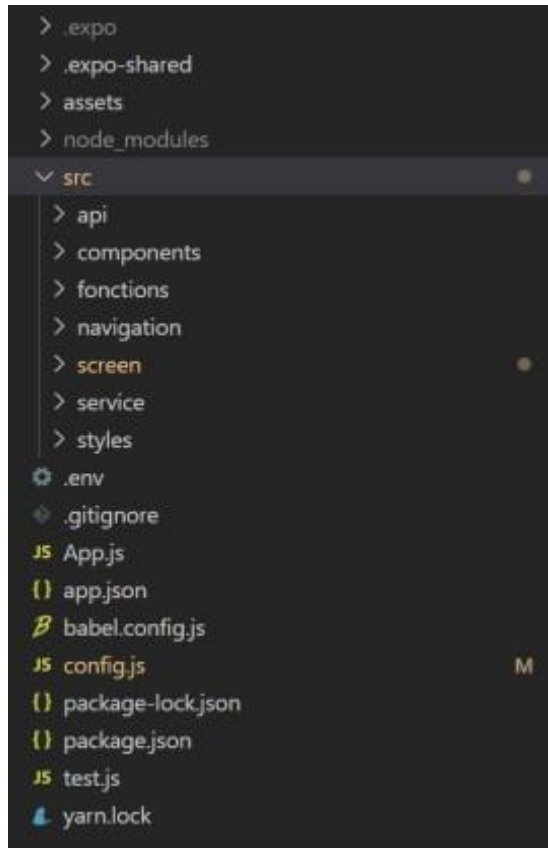
C'est pour cela que j'utilise Jest.

Jest est une librairie javascript conçue pour les tests front end et back end .

Je l'utilise donc pour tester les services de mon projet.

Développement du front-end de l'application

Arborescence



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

- Dans le dossier api , on y trouve tous les fichiers relatifs au appel api.

Chaque fichier contient une fonction générique pour les différentes méthodes évitant ainsi les répétitions.

- Dans le dossier components on y trouve toutes les composants réutilisables ou pas , triés par sous dossier, un sous-dossier par vue.

- Dans le dossier fonctions se trouvent les différentes fonctions utilisées dans ce projet.

- Dans le dossier navigation se trouve la navigation.

- Dans screen , se trouve tous les écrans de mon application mobile.

- Dans service , on y trouve les fichiers contenant la logique métier.

- Et dans styles , on y trouve le style global du projet.

Pages et composants

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

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

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

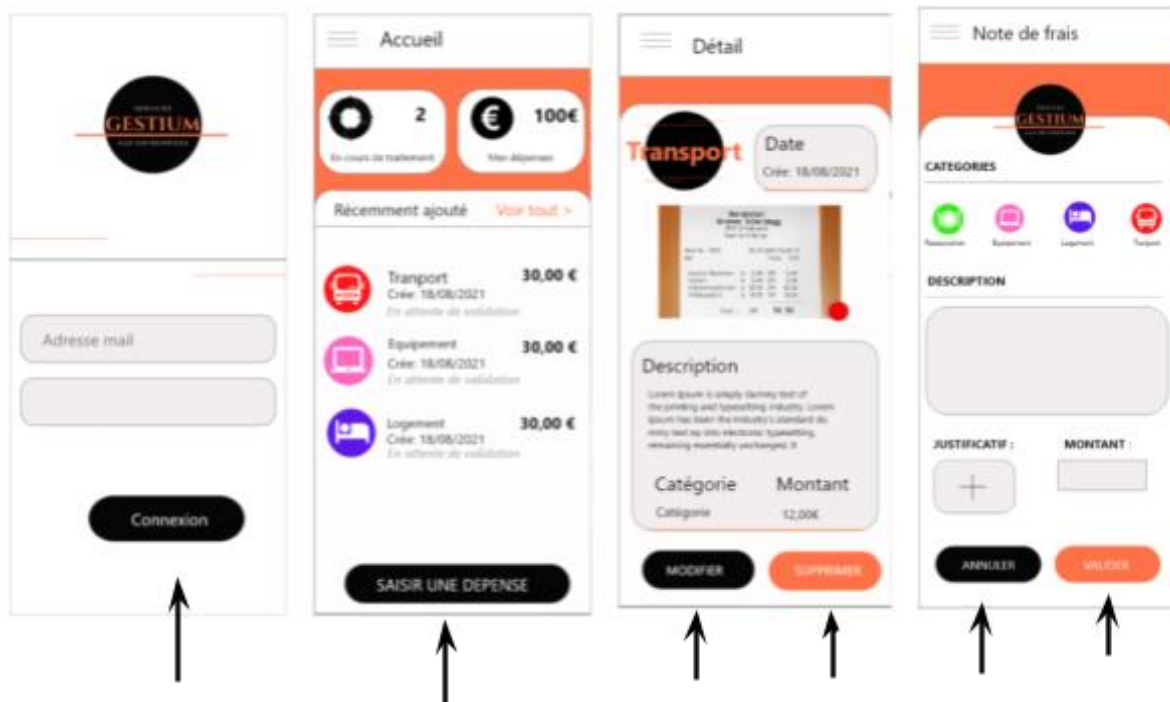
```
import React from 'react';
```

```
import {Text,TouchableOpacity} from 'react-native';

const ButtonCustom =
(props) =>{ return (
    <TouchableOpacity
style={props.styles1} onPress={props.fonction}>
        <Text
style={props.styles2}>{props.text}</Text>
    </TouchableOpacity>
)
}
```

Voici un exemple d'un composant réutilisable, il permet l'affichage d'un bouton. Dans mon projet , se trouvent de nombreux boutons avec des actions différentes . Mais ils sont composés des mêmes éléments , TouchableOpacity et un Text.

Pour éviter les répétitions , j' ai créé un composant nommé ButtonCustom que j'appelle à chaque fois que j'ai besoin d'un bouton. Je lui passe en props toutes les différences comme par exemple le style ou le nom du button.



Comme on peut le remarquer sur l'image ci- dessus , le composant ButtonCustom permet de créer les nombreux boutons de mon application.

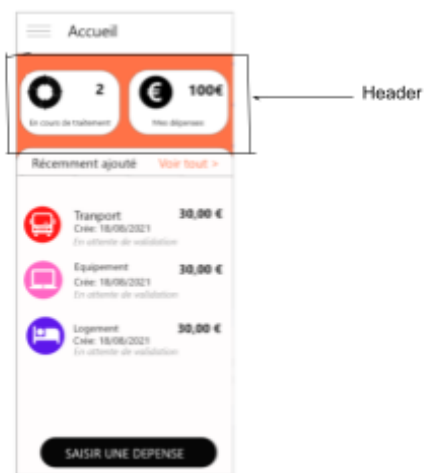
```

const HeaderInformations = () => {

    return (
        <View style={styles.container}>
            <Card
picture={require('../assets/Encours.png')}
compteur={compteur} text={'En cours de traitement'}/>
            <Card picture={require('../assets/Achat.png')}
compteur={ CountAmountBill(bill) + ' €' } text={'Mes dépenses ce
mois '}/>
        </View>
    )
}

const styles =
StyleSheet.create({
    container:{
        flexDirection: "row",
        justifyContent: 'space-
evenly', paddingTop: 25,
        paddingBottom:
25, backgroundColor:
'#FF7247',
    },
});

```



Cet exemple nous montre un composant qui n'est pas réutilisable , il représente le header d'une page .

Sécurité

Dans le l'objectif de rendre mon application la plus sécurisée possible en plus de la sécurité côté serveur, j'ai décidé de rajouter une couche sécurité sur le front de mon application. J'ai décidé d'utiliser les outils mis à disposition dans react native afin de me faire gagner du temps.

J'ai décidé d'utiliser la bibliothèque react-hook-form. L'avantage de cette bibliothèque est qu'elle possède une gestion des erreurs, après lui avoir paramétré les valeurs attendues de chaque champ. Elle se charge de vérifier les champs au clique sur le bouton de soumission du formulaire. Si un des champs ne correspond pas au pattern indiqué dans les paramètres, une erreur est affichée pour guider l'utilisateur et le bouton d'envoi sera complètement désactivé permettant ainsi l'envoi de données solides cotées serveurs.

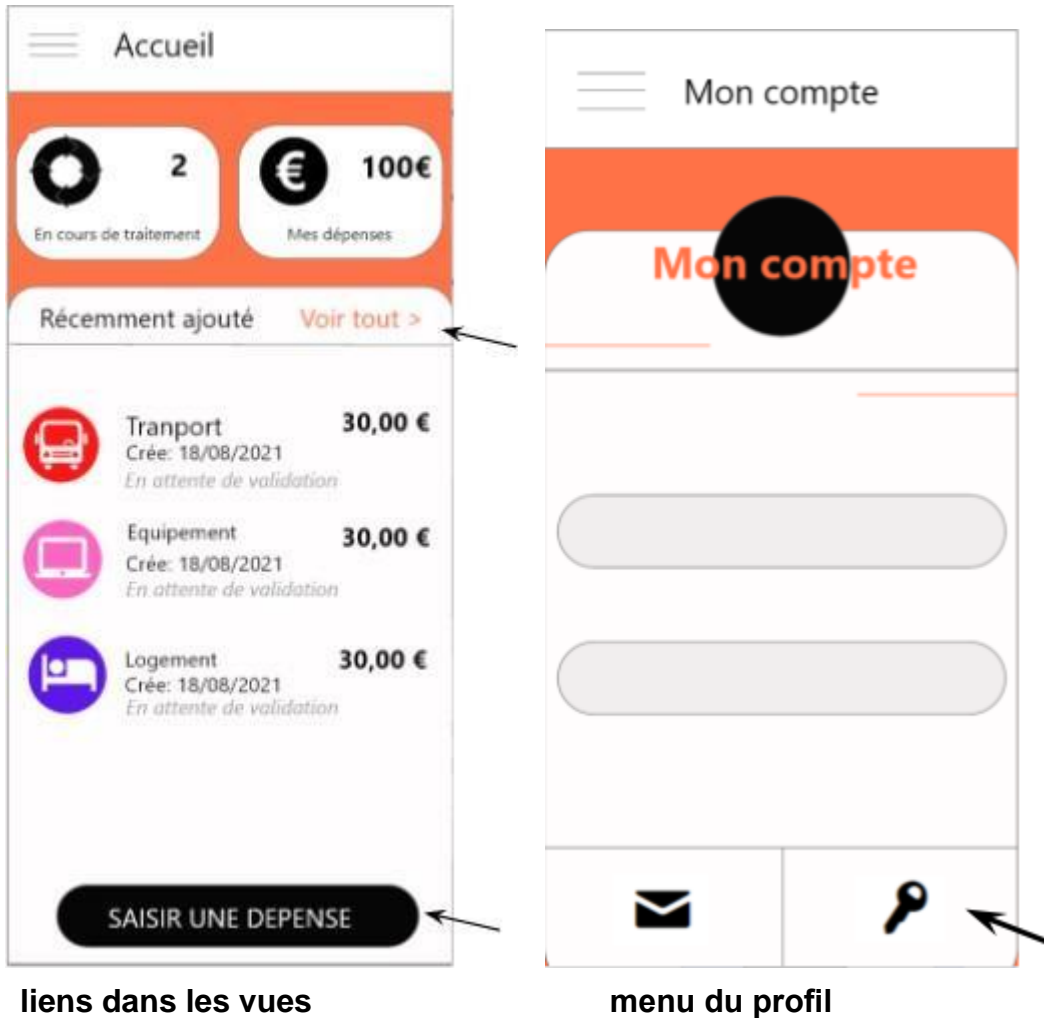
Problématiques rencontrées

Durant le développement de mon application mobile , j'ai rencontré différentes difficultés. L'une de ces difficultés à été la navigation au sein de mon application.

Comme on peut le remarquer sur les maquettes du projet , on voit que l'on peut naviguer entre les vues grâce à un menu sur le côté (drawer) , mais aussi grâce a des liens et grâce à un menu en bas de page sur la page profile.



==> Navigation



Pour ce faire , j'ai du comprendre la navigation dans react native mais aussi les différents navigations dans celle-ci.
J'ai utilisé une bibliothèque React Navigation qui est très populaire dans React Native. Cette bibliothèque à résoudre le problème de navigation entre les pages et le transfert des datas entre celles-ci.

Exemple navigation imbriquée :

```
import React from 'react';
import { createStackNavigator } from '@react-navigation/stack';

import Menu from '../navigation/Menu';
import DetailScreen from '../screen/DetailScreen';
import FormBillScreen from '../screen/FormBillScreen';
import ExportScreen from '../screen/ExportScreen';
```

```
const Navigation = () => {
  const Stack = createStackNavigator()

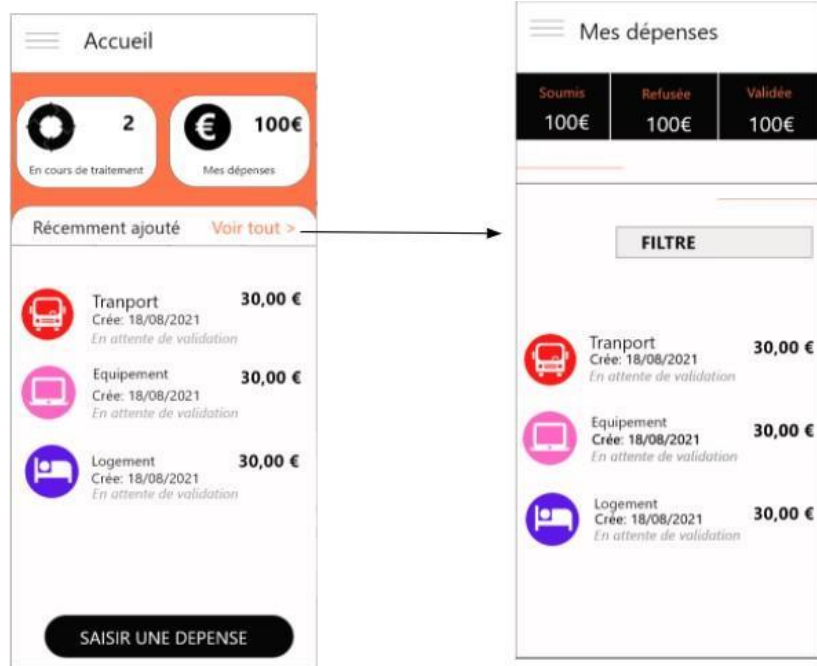
  return(
    <Stack.Navigator>
      <Stack.Screen name="Root" component={Menu}
options={{ headerShown: false }}/>
      <Stack.Screen name="Detail"
component={DetailScreen} />
      <Stack.Screen name="Formulaire note de frais"
component={FormBillScreen} options={{ headerShown: false }} />
      <Stack.Screen name="Export"
component={ExportScreen} />
    </Stack.Navigator>
  )
}
```

Ce composant est chargé de la navigation dans mon application mobile.
Pour qu'elle puisse fonctionner , il faut dans un premier temps importer react navigation ainsi que les différentes vues.

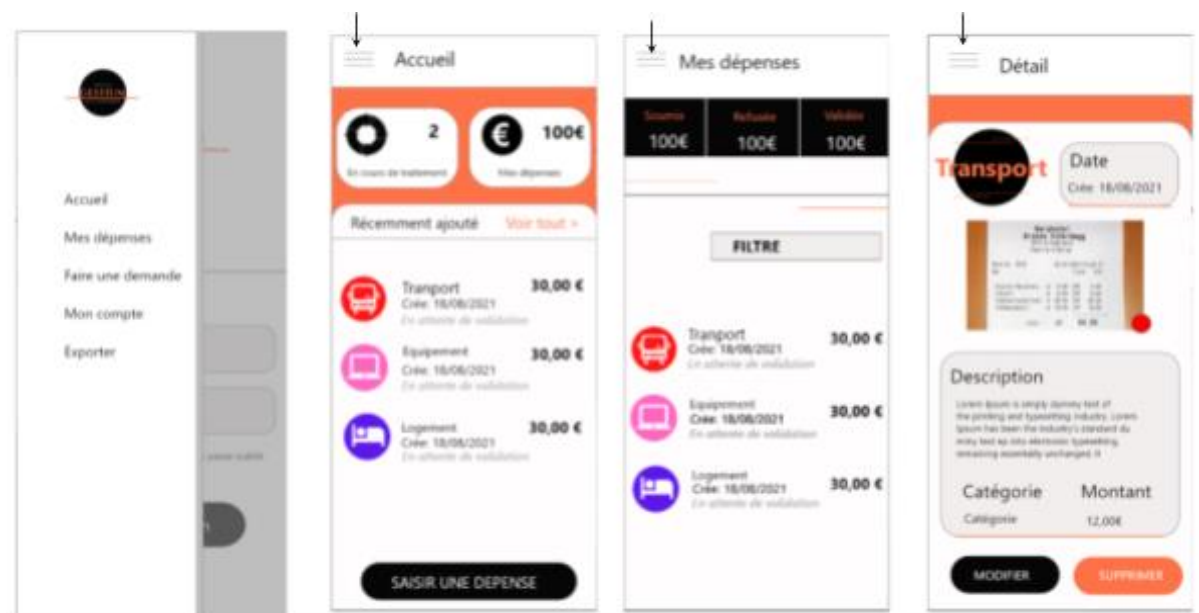
Dans la constante Stack , je stock l'objet createStackNavigator permettant d'avoir accès à toutes les méthodes de cet objet dont le Stack.Navigator .
Le Stack.Navigator permet le stockage des vues de l'application. L'ordre de celle-ci est important.

Ma première Stack Screen permet d'avoir accès à mon menu drawer sur toutes les vues de cette Stack Navigator.

Les autres Stack.Screen correspond à un composant



Stack Navigator



Menu Drawer

Exemple de formulaire de mise à jour du profil

L'utilisateur de l'application a la possibilité de modifier son profil.

La vérification des champs côté client est faite avec la bibliothèque react-hook-form comme annoncé plus haut.

```
const { control, handleSubmit, formState: { errors }, reset }
= useForm();
```

La première étape est d'aller chercher toutes les fonctions nécessaires dans l'objet useForm.

La constante control permet de mettre des paramètres à respecter sécurisant ainsi les données envoyées.

```
const InputComponent = ({style,name,
control,placeholder,secureTextEntry,editable,keyboardType,multiline})
=> {
    const {field} =
    useController({ control,
        defaultVa
    lue: '', name,
        rules:{
            required: true,
        }
    })
    return(
        <TextInput style={styles.input,style} value={field.value}
onBlur={field.onBlur} onChangeText={field.onChange}
placeholder={placeholder} secureTextEntry={secureTextEntry}
editable={editable} keyboardType={keyboardType} multiline={multiline}/>
    )
}
```

Comme on peut le voir sur ce composant , ce champ sera obligatoire, il n'a pas de valeur par défaut et possède un name permettant de récupérer sa valeur .

Si le champs est vide une erreur sera afficher , comme ceci:

```
<View style={styles.error}>
    {error ? <Text style={General_styles.G_Text_Error}>Le Champs est
requis</Text>: null}
</View>
```

Si une erreur est commise par l'utilisateur , la constante est commise par la bibliothèque. La condition ternaire vérifie si error est rempli . Si c'est le cas , le texte est affiché sinon rien ne se passe.

COncePTIOOn 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.

Dans l'optique de rester cohérente dans le choix de développement de celui-ci , j'ai choisi le framework react Js pour le développer. Ce choix m'a permis une montée en compétences sur react.

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

User Story

Une personne qui a les droits DRH doit pouvoir se connecter au site web, il a donc besoin d'un formulaire de connexion dans lequel il entre son login et mot de passe. Si ses identifiants sont corrects , il a accès à une page d'accueil sur laquelle se trouve plusieurs informations utiles , comme un graphique contenant l'intégralité des demandes de remboursement sur l'année. L'utilisateur doit pouvoir naviguer au sein de ce site , il faut donc une navigation. Un menu sur le côté est créé avec différents liens permettant l'accès aux pages du site. L'utilisateur doit pouvoir ajouter un salarié. Donc il est nécessaire de créer un espace sur lequel se trouve un formulaire qui doit être composé de différentes informations du salarié. Un champ prénom , nom , le poste occupé , ses droits doivent être renseigné. Ce formulaire est utilisé pour la création d'un espace pour l'application mobile. La personne à la possibilité de supprimer les droits d'un salarié , il pourra donc le faire en cliquant sur le bouton supprimer de la fiche du salarié.

Les notes de frais doivent être gérées , il est donc nécessaire de créer une page sur laquelle on y trouve l'intégralité des demandes. Il doit pouvoir faire une action sur chaque notes de frais comme par exemple accepter le remboursement ou le refuser. Il est important pour des raisons de comptabilité de garder un trace de celle-ci, la note doit pouvoir être archivée. Pour ce faire , il sélectionne une note de frais grâce à une case à cocher et appuie sur le bouton archiver. Bien sûr, la note de frais doit être traitée pour faire cette action. Aucune note de frais en cours de traitement ne peut être archivée.

L'utilisateur peut aussi gérer les postes au sein de son entreprise un formulaire est mis à disposition pour pouvoir en ajouter. Il est donc nécessaire d'avoir un formulaire avec un champ permettant d'enregistrer celui-ci. Il a aussi la possibilité de le supprimer grâce à un bouton .

Choix du langage et framework

Pour réaliser ce site j'ai utilisé ReactJs pour le front , pour le back end , j'utilise le framework ExpressJs.

Conception du frontend du site web

Charte graphique

Dans l'optique d'être un coherent , 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.

J'ai aussi utilisé adobe XD.

Voici un exemple :



Maquette de la page d'accueil du site

Conception du back end du site web

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

Des routes ont été créées spécialement pour la gestion côté administrateur comme par exemple la routes GET : 'v1/employees' qui récupère l'intégralité des utilisateurs inscrits. Cette route est protégée, il faut avoir le rôle DRH.

COncLUslOn

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

Ce projet qui s'est déroulé sur l'année de formation m'a appris à organiser mon travail.

En effet , durant cette année je n'ai pas seulement développé ce projet , j'ai aussi collaboré sur le développement d'une autre application mobile avec des personnes de ma promotion. Une application sur la gestion des humeurs développée avec symfony pour l'API et react Native pour l'application. Celle-ci m'a permis de développer des compétences transverses comme le travail d'équipe.

De plus, le projet présenté dans ce dossier m'a permis de voir que je pouvais m'adapter aux différentes situations.

ANNEXE

Annexe 1 : Maquette de l'application mobile - GESTIUM


Page de connexion




Adresse mail


Connexion


Accueil
Page d'accueil

**2**
En cours de traitement

**100€**
Mes dépenses

Récemment ajouté [Voir tout >](#)

**Transport** **30,00 €**
Crée: 18/08/2021
En attente de validation

**Equipement** **30,00 €**
Crée: 18/08/2021
En attente de validation

**Logement** **30,00 €**
Crée: 18/08/2021
En attente de validation

SAISIR UNE DEPENSE

Mes dépenses

Soumis

100€

Refusée

100€

Validée

100€

FILTRE

Transport

Crée: 18/08/2021

En attente de validation

30,00 €

Equipement

Crée: 18/08/2021

En attente de validation

30,00 €

Logement

Crée: 18/08/2021

En attente de validation

30,00 €

Pages mes dépenses

Détail

Transport

Date

Crée: 18/08/2021

Berghotel
Grosse Scheidegg
3818 Grindelwald
Familie R. Müller

Rech.Nr. 4572

30.07.2007/13:29:17

Bar

Tisch 7/01

2xLatte Macchiato

à 4.50 CHF

9.00

1xGlögi

à 5.00 CHF

5.00

1xSchweinsschnitzel

à 22.00 CHF

22.00

1xChasapätzli

à 18.50 CHF

18.50

Total :

CHF

54.50

Description

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and composed it to produce a sample of the typefaces available at the time. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the early 1960s by the Letraset team, who used it to produce a sample of the typefaces available at the time. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the early 1960s by the Letraset team, who used it to produce a sample of the typefaces available at the time.

Catégorie

Montant

Catégorie

12,00€

MODIFIER

SUPPRIMER

Page détail

Note de frais

SERVICES
GESTIUM
AUX ENTREPRISES

CATEGORIES

Restauration

Equiptement

Logement

Tranport

DESCRIPTION

JUSTIFICATIF :

+

MONTANT :

ANNULER

VALIDER

Exporter

Période

A partir de

12/12/2021

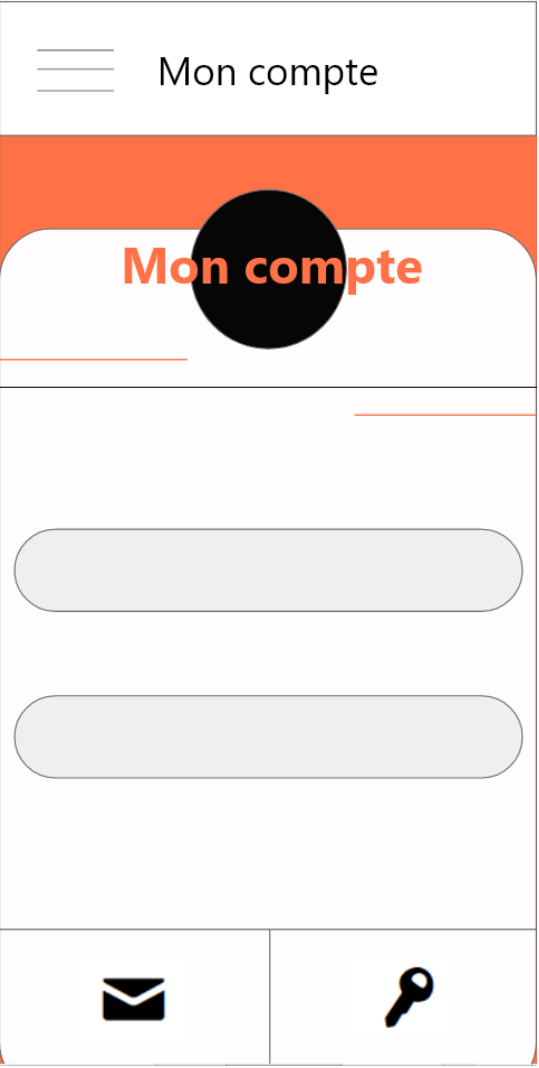
Jusqu'a

12/12/2021

VALIDER

Page demande de remboursement

Page exportation



Page mon compte