

Dossier de projet professionnel



Réalisation de l'application web et mobile

Quiz Connect

Présenté par **Fatima EL MOUHINE**

SOMMAIRE

Résumé.....	3
Introduction.....	4
Présentation personnelle.....	4
Présentation du projet en anglais.....	4
Compétences couvertes par le projet.....	5
Organisation et cahier des charges.....	6
Analyse de l'existant.....	6
Les utilisateurs du projet.....	6
Les fonctionnalités attendues.....	6
Application Mobile.....	6
Application Web (Panel admin).....	8
Contexte technique.....	9
Conception du projet.....	9
Logiciels Et Autres Outils.....	12
Organisation Du Projet.....	12
Architecture logiciel.....	15
Conception du Front-end de l'application.....	16
Arborescence du projet.....	16
Charte graphique.....	17
Maquettage (voir annexe 1).....	17
Wireframe.....	17
Haute fidélité.....	18
Conception du Backend de l'application.....	19
Mise en place de la base de données.....	19
Modélisation de la base de données.....	19
Méthode MERISE.....	19
Modèle conceptuel de données (MCD)(voir annexe 2).....	19
Modèle Logique de données (MLD) (voir annexe 3).....	21
Développement du backend de l'application.....	23
Organisation.....	23
Arborescence.....	23
Utilisation d'une API.....	24
Fonctionnement d'un middleware.....	28
Les Controllers.....	30
Conteneurisation du Backend.....	31

Connexion à la base de données.....	32
Sequelize.....	32
Définition des relations.....	33
Model.....	34
Fixtures.....	35
Sécurité.....	37
Injection SQL.....	38
Credential stuffing : vol du login et password.....	39
Sécurisation avec TOKEN.....	40
Gestion des rôles.....	40
Utilisation de Helmet.....	41
Autorisation des CORS Policy.....	42
Documentation.....	43
Test unitaire.....	44
Développement Du Front-end de l'application.....	48
Arborescence.....	48
Page et composant.....	50
Navigation.....	51
Développement du Panel d'administration.....	54
Utilisation de Next.js.....	55
Page de connexion.....	56
Page de gestion des utilisateurs.....	57
Explication de la logique postUser.....	58
Explication de la logique deleteUser.....	60
Explication de la logique putUser.....	62
Recherche anglophone.....	62
Context.....	62
Traduction.....	63
Conclusion.....	65
Annexes.....	66
Maquettage.....	66
Wireframe.....	66
Maquette haute fidélité.....	67
Modèle conceptuel de données.....	69
Modèle logique de donnée.....	70

Résumé

Quiz Connect est une application développée pour répondre aux besoins spécifiques des étudiants et des tuteurs dans les prépas de médecine. Cette plateforme innovante vise à améliorer l'expérience d'apprentissage des étudiants en première année en proposant une solution complète pour leurs révisions.

Cette application offre une variété de fonctionnalités pratiques. Les étudiants peuvent accéder à des leçons, rechercher des sujets spécifiques, s'entraîner avec des questions à choix multiples (QCM) et interagir sur un forum dédié aux questions et au soutien. Ces fonctionnalités sont spécialement développées pour les aider à surmonter la charge de travail importante liée à leur formation.

Pour une expérience fluide et accessible, Quiz Connect est construite à l'aide de React Native, une plateforme polyvalente compatible avec différents appareils mobiles. Ainsi, les étudiants peuvent bénéficier des fonctionnalités de l'application où qu'ils se trouvent, à tout moment, en utilisant leur appareil mobile préféré.

L'application intègre également une interface d'administration développée avec Next.js, un framework React.js réputé pour sa performance et son efficacité. Cette interface permet aux tuteurs et aux administrateurs de gérer l'application de manière efficace, de suivre les progrès des étudiants et de leur apporter le soutien nécessaire.

Quiz Connect a pour objectif de révolutionner l'approche de l'apprentissage médical pour les étudiants. En proposant une plateforme intuitive et rentable, elle donne aux étudiants les moyens d'améliorer leur apprentissage, d'augmenter leurs chances de réussite et de surmonter le taux d'échec élevé associé aux examens de première année.

Introduction

Présentation personnelle

Je m'appelle Fatima EL MOUHINE, j'ai vingt-sept ans.

J'ai découvert la programmation il y a maintenant quelques années avec mes premiers pas sur html, css et php en option à la fac. Cela m'a permis de découvrir tous les métiers liés au numérique. J'ai suivi le cursus de la Coding School de La Plateforme en 2021/2022, j'y ai obtenu mon titre de développeur web et web mobile . Aujourd' hui en Coding School 2, je prépare mon titre de concepteur/développeur d'application web et en alternance à L'Atelier de La Plateforme.

Présentation du projet en anglais

Quiz Connect is a mobile application designed to address the high failure rate experienced by first-year medical students in their final exams. The project aims to tackle the challenges faced by these students in the first year, known as PACES. The difficulty of PACES is not solely attributed to the complexity of the subjects but rather the overwhelming amount of content that needs to be learned within a limited timeframe.

Many pre-med schools offer success strategies at exorbitant prices. There are already web interface solutions available, such as Medibox, but they are not affordable for all students.

The mobile application, developed using React Native, encompasses various features such as authentication, a profile section for updating personal information, access to lessons, a forum for posing questions, subject-based search functionality, and MCQs. The mobile application is integrated with the backend through an API built with Node.js and Sequelize.

Additionally, the same API is utilized for the admin interface, which is developed using Next.js, a React.js framework that facilitates server-side rendering and information caching to minimize redundant requests.

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
- Développer des composants dans le langage d'une base de données
- Concevoir une base de données
- Mettre en place une base de données
- Développer des composants dans le langage d'une base de données
- Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement
- Concevoir une application
- Développer des composants métier
- Construire une application organisée en couches
- Développer une application mobile
- Préparer et exécuter les plans de tests d'une application

Organisation et cahier des charges

Analyse de l'existant

Comme expliqué précédemment, il existe plusieurs solutions pour réviser comme Medibox, un site web où les étudiants peuvent faire des entraînements de QCM et ont accès à des cours. Cependant cette solution n'est pas assez personnalisable pour les tuteurs qui rédigent leurs propres QCM et ne leur permet pas de communiquer avec les étudiants sur un sujet. De plus, l'abonnement n'est pas forcément accessible à tous les étudiants. Avec notre application, on se démarque de la concurrence sur plusieurs points.

Les utilisateurs du projet

Ce projet se décompose en deux parties : une application mobile et une interface web.

La partie application mobile sera utilisée par des étudiants. Ils devront se connecter pour pouvoir accéder à leur profil et effectuer différentes actions comme consulter un cours par exemple.

La partie site web est utilisée uniquement par les admin et les tuteurs grâce à des droits administrateur.

Cet espace leur permettra de générer des QCM de type examen, insérer les cours, les forums officiels et modérer les messages du forum.

Les fonctionnalités attendues

Application Mobile

Page de démarrage :

Lorsque l'utilisateur lance l'application mobile , il est redirigé vers une page d'accueil sur laquelle une navigation permet de voir et d'accepter les conditions d'utilisation puis vers la page connexion.

Page connexion :

La page de connexion permet aux utilisateurs d'accéder à leur compte et d'utiliser les fonctionnalités réservées aux utilisateurs connectés telles que la personnalisation de leur profil, l'envoi de messages sur le forum etc... Cette page inclut des mesures de sécurité, telles que la vérification du mot de passe et le contrôle des mails déjà existants.

Page d'accueil:

La page d'accueil permettra aux utilisateurs de voir les derniers cours mis en ligne et les derniers QCM effectués.

Page profil:

La page profil permet aux utilisateurs de gérer leur compte, en modifiant leurs informations personnelles tels que leur nom, prénom et leur photo de profil, etc.

Page thème:

La page thème permettra d'accéder aux différentes fonctionnalités de l'application selon le thème choisi.

Page cours :

La page cours pourra afficher les différents cours mis en ligne, par le biais des recherches ou directement selon le thème choisi depuis la page thème.

Page pré-QCM:

Sur cette page on peut :

- soit en appuyant sur le bouton recherche, nous diriger vers la page qcm
- soit appuyer sur le bouton génération grâce auquel on est redirigé sur une page de génération de qcm. L'utilisateur peut choisir le nombre de questions et le thème sur lequel il veut s'entraîner. Si le nombre de questions est supérieur à celui existant pour le thème choisi, une erreur sera renvoyée.

Page QCM:

Sur cette page on trouvera la liste de tous les qcm disponibles avec le nombre de questions associées.

Page QCM détail :

Sur cette page, on peut répondre aux questions une à une. En appuyant sur le bouton "terminer", l'utilisateur est redirigé vers la page des scores et il peut consulter également la correction.

Page pré-forum :

Sur cette page on peut :

- soit en appuyant sur le bouton recherche, nous diriger vers la page forum et ainsi consulter les différents sujets postés
- Soit arriver sur une page de création d'un nouveau sujet, en pressant le bouton "Nouveau sujet".

Page forum :

En arrivant sur cette page, on peut voir la liste des différents sujets avec leur nombre de messages associés. Il y a aussi une barre de recherche qui nous permet de choisir parmi les sujets existants. On peut également filtrer sur la recherche par thèmes / contenu de message / sujet.

Page forum détail :

En appuyant sur un sujet, on arrive sur la page détaillée de celle-ci. Dessus on a la possibilité de lire les précédents messages (la fonctionnalité de like n'est pas encore disponible). Dans le bloc qui contient le message, on retrouve le prénom de la personne qui a écrit le message et s'il est admin ou tuteur il aura un badge adapté. On peut également poster un nouveau message dans le fil du sujet.

Application Web (Panel admin)

Page connexion admin :

La page de connexion permet uniquement aux admins d'accéder au panel admin. Une fois connectés, ils pourront accéder à toutes les fonctionnalités du panel admin.

Page admin étudiants :

La page étudiants permet aux admins d'accéder au panel admin, de gérer les étudiants inscrits.

Ils pourront alors les inscrire ou les supprimer mais également modifier leurs informations telles que leur email de connexion en cas d'oubli.

Page admin cours :

Cette page affiche tous les cours mis en ligne, ils pourront être modifiés ou supprimés en cas de besoin.

Il sera aussi possible d'ajouter les nouveaux cours depuis cet espace.

Page admin forum :

Cette page affiche tous les forums créés, il sera possible d'en créer de nouveaux depuis cet espace, modifier les anciens ou supprimer un forum. Depuis cet espace, il sera aussi possible d'accéder aux messages des forums pour modérer les messages indésirables.

Page admin thème :

Cette page affiche tous les thèmes créés, il sera possible d'en créer de nouveaux, modifier les anciens ou d'en supprimer.

Page admin qcm:

Cette page affiche tous les qcm créés, il sera possible d'en créer de nouveaux, modifier les anciens ou supprimer un qcm.

Depuis cet espace, il sera aussi possible d'accéder aux questions d'un qcm choisies pour pouvoir en ajouter, supprimer ou en modifier une.

Contexte technique

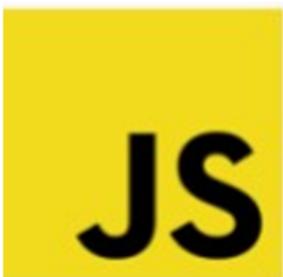
L'application mobile devra être accessible sur IOS, puis par la suite sur tous les systèmes d'exploitation.

L'application web devra être accessible sur tous les navigateurs.

Conception du projet

Choix de développement

Choix des langages



Pour réaliser ce projet, j'ai décidé d'utiliser uniquement du Javascript.



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 fréquemment 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.

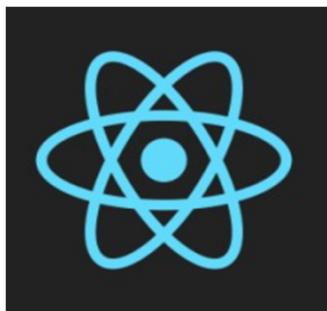
Choix des frameworks

Pour la création de mon API j'ai choisi d'utiliser Express Js qui est un framework Node Js.



ExpressJs est le framework le plus populaire pour Node Js. C'est un framework minimaliste permettant de garder un certain contrôle dans le développement du projet et apporte peu de surcouche, permettant ainsi de garder des performances optimales et une exécution rapide.

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 Android.



Pour la création de mon application web, j'ai utilisé Next.js qui est une surcouche React js pour sa facilité d'intégration, son routeur intégré. Il prend en charge aussi le rendu côté serveur, ce qui signifie que les pages sont pré-rendues sur le serveur avant d'être envoyées au navigateur.

Logiciels Et Autres Outils

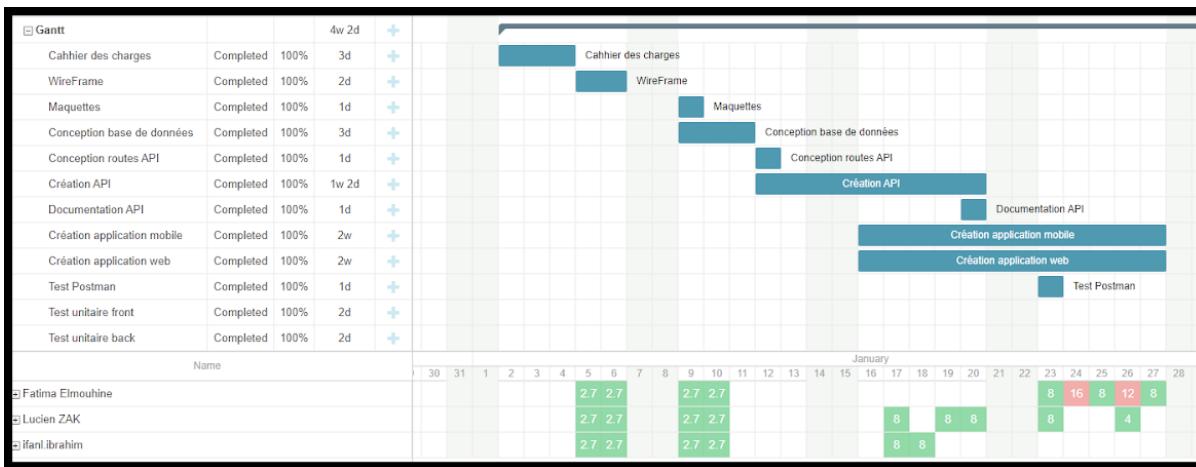
Dans le cadre de ce projet, j'ai dû utiliser d'autres outils:

- **Visual Studio Code pour écrire mon code;**
- **Docker pour containeriser notre application**
- **Jest pour les test unitaires**
- **Postman pour effectuer les requêtes API;**
- **Git pour le versionning de mon code;**
- **Trello pour organiser mon travail;**
- **NPM pour installer les paquets;**
- **Figma pour la création de mes maquettes, de la charte graphique, la création du logo**
- **draw.io pour le maquettage de la base de données;**
- **Expo pour émuler l' application mobile sur mon téléphone;**

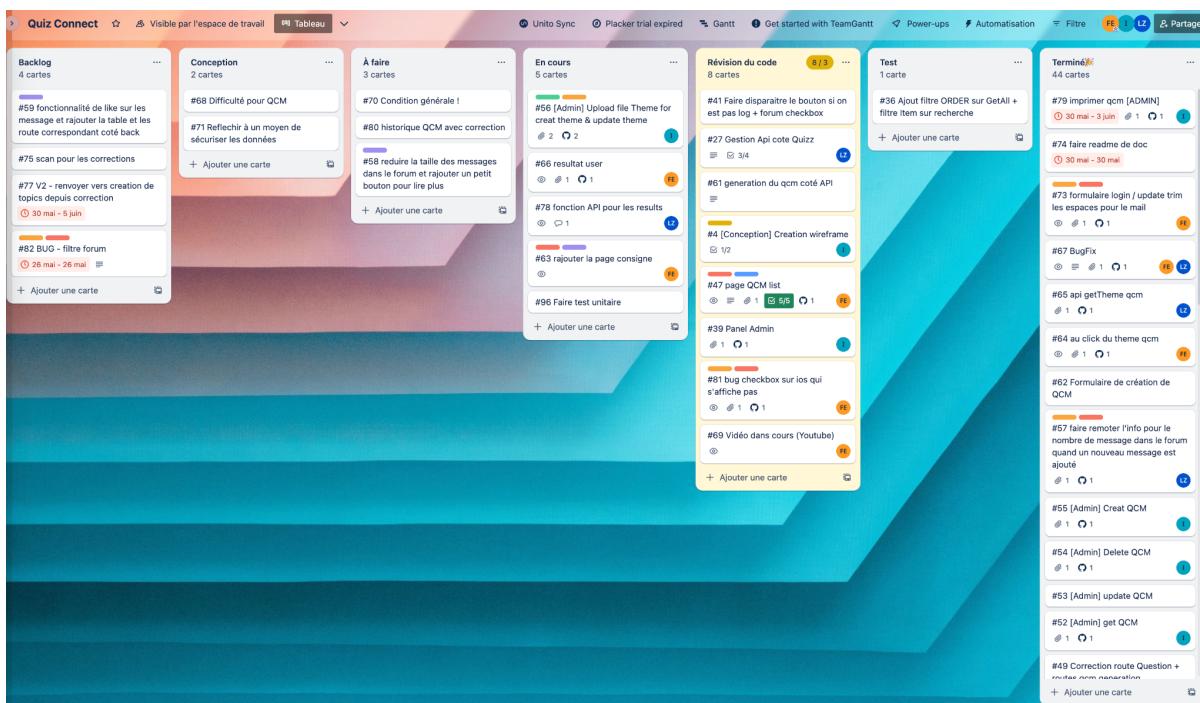
Organisation Du Projet

Ce projet a été réalisé durant mon année de formation, avec des temps en entreprise et en centre où j'avais des projets à rendre. L'organisation de mon travail a donc été essentielle.

Pour ce faire , j'ai tout d'abord créé un diagramme de Gantt. Bien que ce soit un outil dit "anti agile", cela 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.



J'ai aussi utilisé **Trello** pour lister les tâches à effectuer dans la colonne backlog.



Code couleur étiquette :

Admin
Conception
debug
Front
bonus
back

On a établi un code couleur pour les différentes tâches.

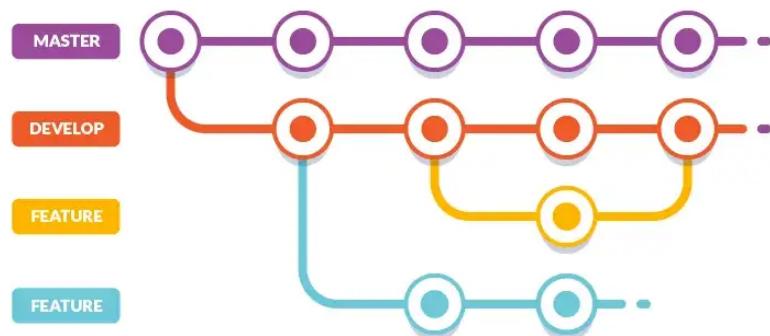
On a également utilisé une extension chrome "Trello Card Numbers" qui permet de numérotier nos tâches.

Ce système de numérotation nous a servi pour la nomenclature de nos branches sur git.

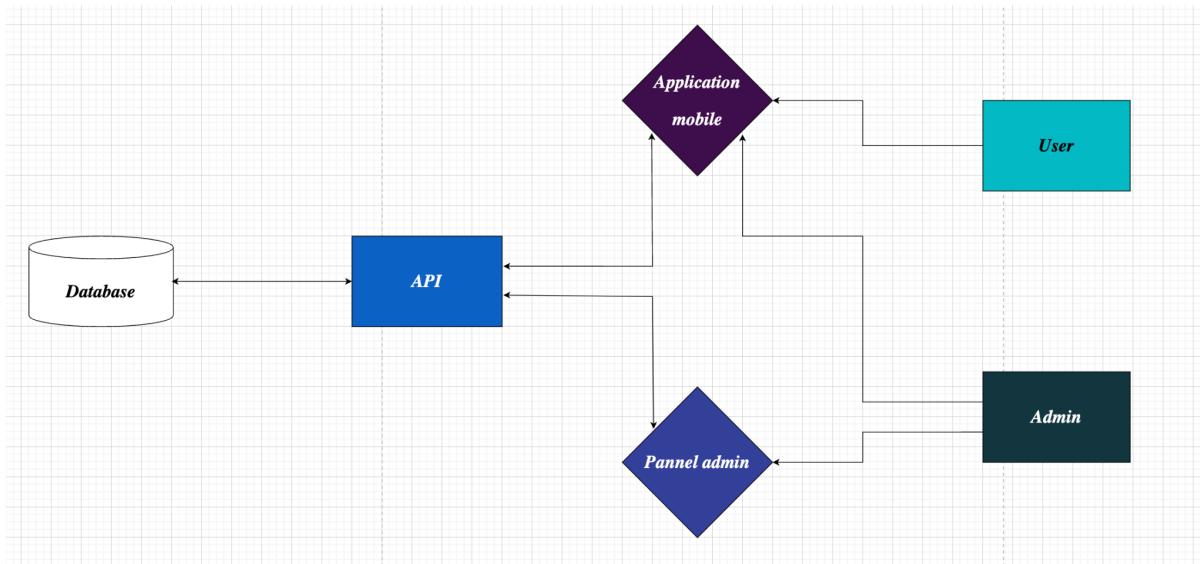
Nous avons également pu relier chaque ticket avec nos branches grâce à l'installation d'un plugin github de Trello.

Nous avons mis en place un git flow pour une organisation plus claire

- Master pour la version en production
- Develop pour la version en cours de développement permettant de tester le code
- Feature, Bugfix, Hotfix pour Merge du code supervisé (Pull Request)



Architecture logiciel



Nous avons opté pour ce type d'architecture pour des raisons pratiques. On a préféré isoler l'api de façon à ce qu'elle soit accessible depuis l'application mobile mais également depuis le panel admin.

Chaque partie (API, application mobile, admin) peut être développée et maintenue indépendamment des autres, ce qui facilite la gestion du code et permet des mises à jour spécifiques à chaque module.

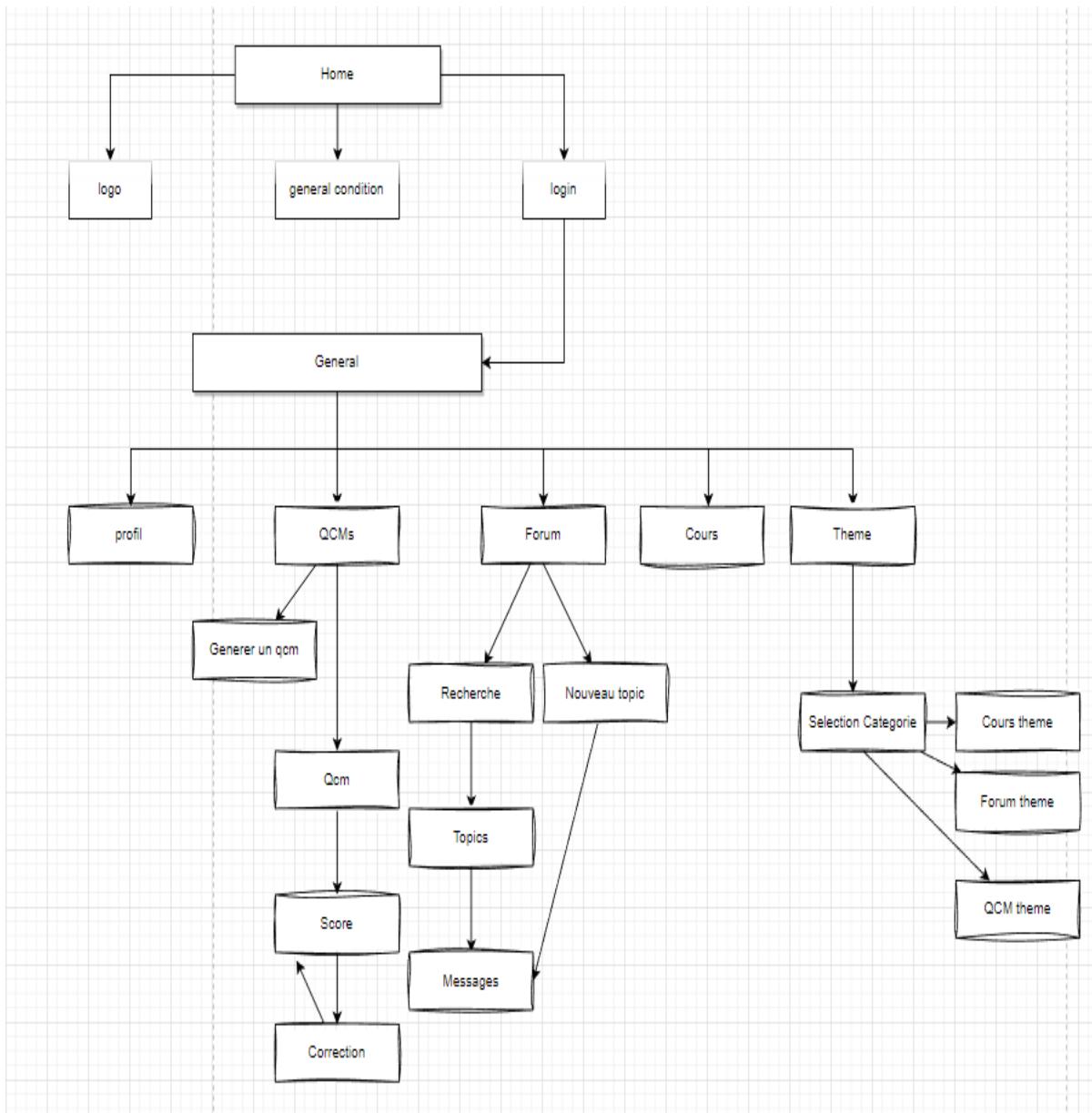
En séparant l'API, le frontend et l'admin, on a pu mettre en place des mesures de sécurité spécifiques à chaque partie de l'application. Par exemple, on peut appliquer des stratégies de sécurité différentes pour l'API, comme l'authentification et l'autorisation, afin de protéger les ressources et les données sensibles. La séparation des dossiers facilite la gestion des contrôles de sécurité spécifiques à chaque module.

Cela permet également une collaboration plus efficace entre les membres de l'équipe. On a pu travailler sur des parties spécifiques de l'application en parallèle, sans se marcher sur les pieds. Cela permet de gagner du temps et d'accélérer le processus de développement.

Conception du Front-end de l'application

Pour créer la maquette et la charte graphique , j'ai choisi d'utiliser **Figma** car il est gratuit, plutôt simple d'utilisation et permet la réalisation de maquettes réalistes.

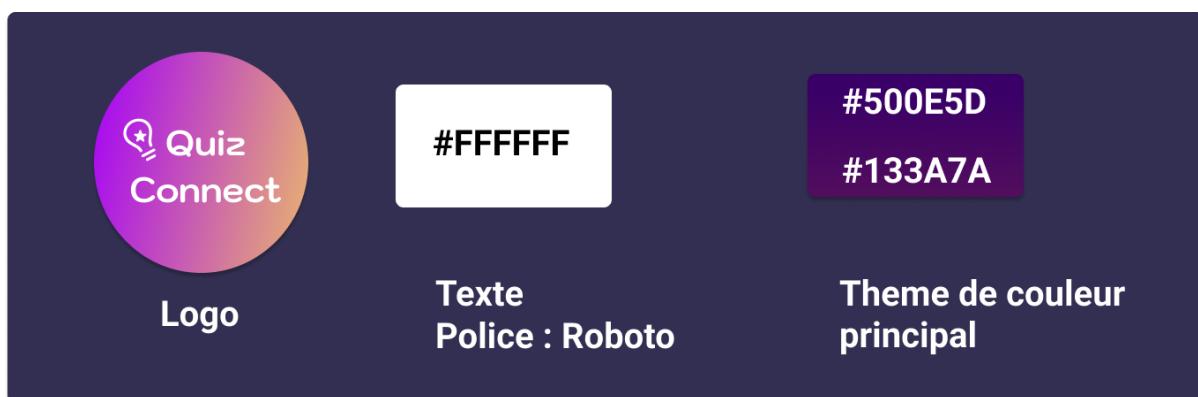
Arborescence du projet



Charte graphique

Après la création de l'arborescence du projet, j'ai commencé par la création du logo à l'aide de **Canva**. J'ai ensuite récupéré les couleurs de celui-ci pour créer la charte graphique en utilisant **Figma**.

En ce qui concerne la charte graphique, je définis, entre autres, les couleurs, le logo et la police utilisés pour la typographie afin que le site ait une identité visuelle, qu'il soit attrayant visuellement pour l'utilisateur. L'utilisateur ne voit et ne se fie qu'à l'apparence du site et ne peut pas voir le Back-office.

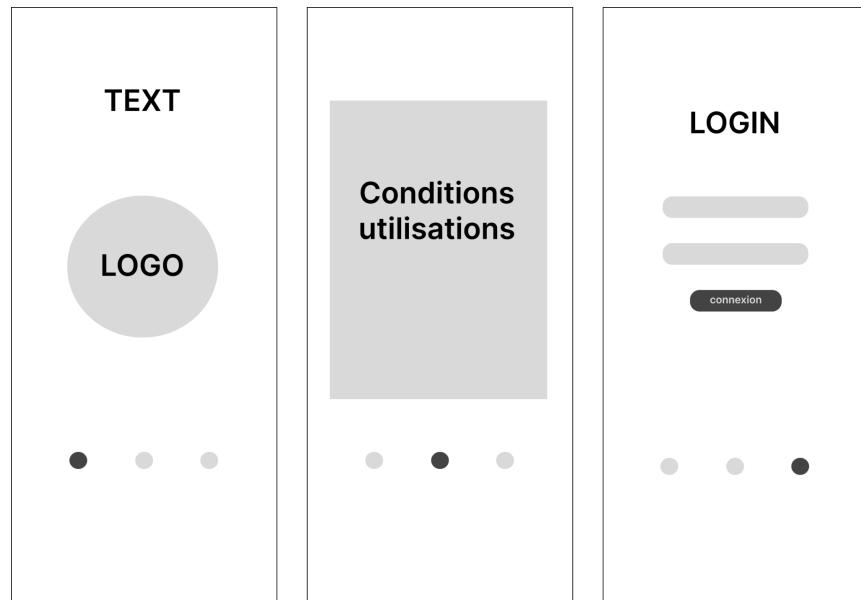


Maquettage (voir annexe 1)

Les maquettes ont été créées à l'aide de **Figma**.

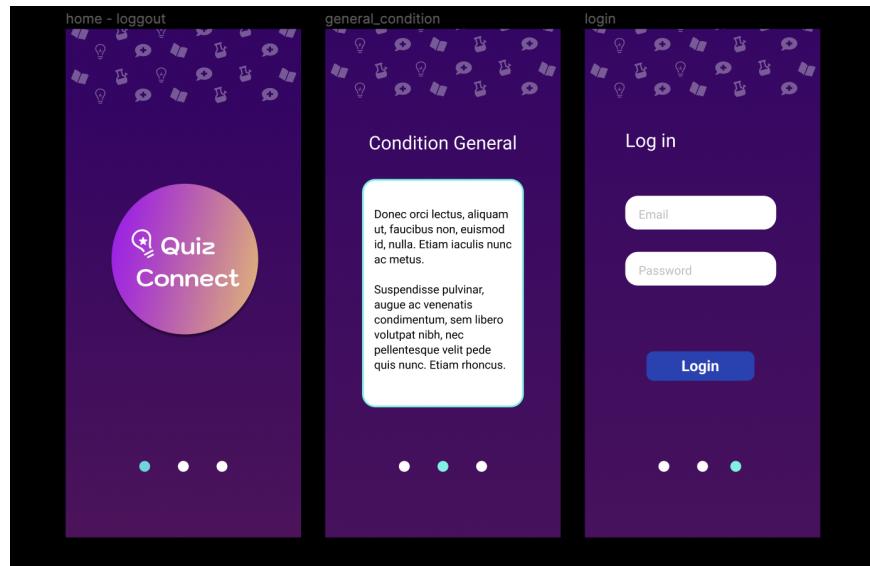
Wireframe

J'ai commencé par réaliser un wireframe afin de visualiser et valider comment seront agencés les éléments sur les différentes pages. J'ai choisi un design minimaliste, très en vogue actuellement dans l'univers du web.



Haute fidélité

J'ai ensuite réalisé la maquette haute fidélité permettant au client de voir clairement l'aspect graphique du site avec les couleurs et les images et ainsi valider la charte graphique du site.



Durant ce projet, plusieurs librairies compatibles avec React native pour le design, ont été utilisées, comme :

- React Native paper qui nous permet d'accéder à des composants déjà stylisés (card, list, slider...)
- React Native linear gradient pour nous permettre de faire des dégradés

Conception du Backend de l'application

Mise en place de la base de données

Pour ce projet , il est indispensable d'avoir une base de données, afin de garder les informations importantes des étudiants et des cours.

Modélisation de la base de données

Méthode MERISE

MERISE (Méthode d'Etude et de Réalisation Informatique pour les Systèmes d'Entreprise), c'est une méthode française qui a émergé dans les années 70 en France et qui permet la modélisation et la conception de S.I. Parmi les ressources informatiques de ces S.I., il y a en particulier les fichiers de données, bases de données et systèmes de gestion de bases de données (S.G.B.D.).

C'est sur ce dernier point que la méthode MERISE nous a été utile, car c'est en se basant sur ses principes de modélisation que nous avons conçu la base de données de TissApp.

Modèle conceptuel de données (MCD)(voir annexe 2)

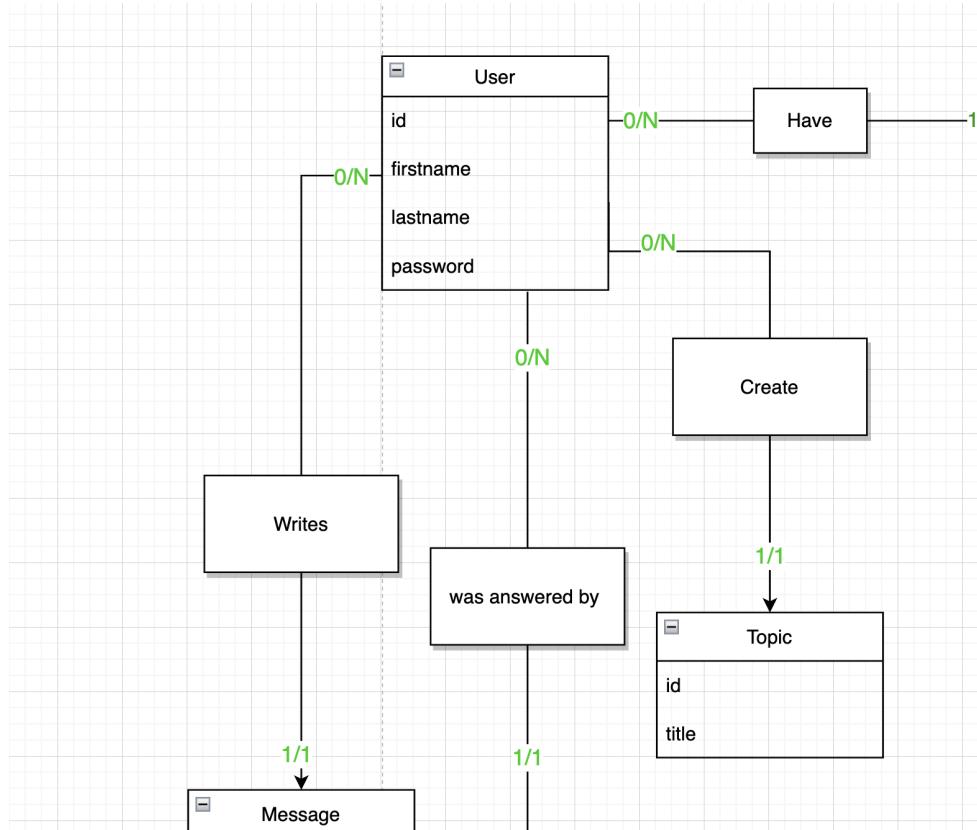
C'est un modèle simplifié de la base de données, où les tables existent seulement au stade d'entité. Entre les différentes entités il y a des liaisons que l'on appelle association: qui est le verbe d'action qui décrit l'opération qui se fait entre les deux entités. Donc nous distinguons principalement les entités et les associations, d'où son second nom de schéma Entité/Association.

Tout d'abord, il a fallu imaginer tous les besoins du client et notamment la partie back office. A partir de ces besoins, j'ai été en mesure d'établir les règles de gestion des données à conserver.

Ensuite, il faut définir le dictionnaire des données, c'est-à-dire toutes les données élémentaires qui vont être conservées en base de données et définir certaines caractéristiques qui figureront dans le MCD. Parmi ces caractéristiques, on retrouve par exemple la référence d'une donnée et notamment un identifiant unique, sa désignation, son type, etc.

Étape finale à sa conception, j'ai pu, à partir des informations précédemment recueillies, créer chaque entité, unique et décrite par un ensemble de propriétés ; Et leurs associations permettant de définir les liens et cardinalités entre les entités.

MCD



Par exemple, la table User est reliée à la table Topic par un verbe pour matérialiser leur relation comme on peut le voir illustré au

dessus :

- **1 utilisateur** à au minimum 0 topic et au maximum **N topics**
- **1 topic** ne peut être créé que par un **utilisateur**.

Modèle Logique de données (MLD) (voir annexe 3)

Le modèle logique de données (MLD) est une étape intermédiaire entre le modèle conceptuel de données et le modèle physique de données.

Le passage d'un MCD en MLD s'effectue selon quelques règles de conversion précise :

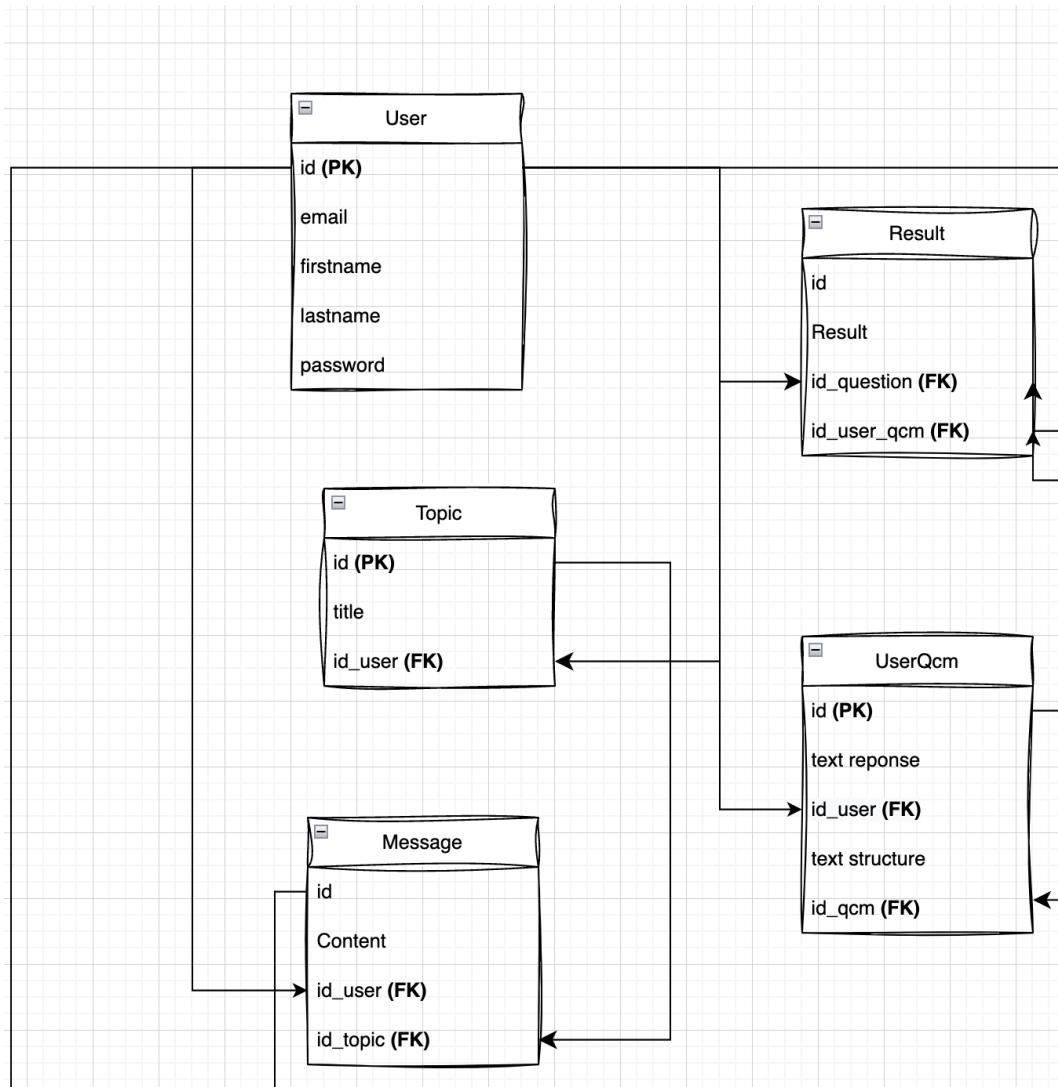
Une entité du MCD devient une table. Dans un SGBD de type relationnel, une table est une structure tabulaire dont chaque ligne correspond aux données d'un objet enregistré et où chaque colonne correspond à une propriété de cet objet.

Ces colonnes font notamment référence aux caractéristiques définies dans le dictionnaire de données du MCD.

Les identifiants respectifs de chaque entité deviennent des clefs primaires et toutes les autres propriétés définies dans le MCD deviennent des attributs. Les clefs primaires permettent d'identifier de façon unique chaque enregistrement dans une table et ne peuvent pas avoir de valeur nulle.

Les cardinalités de type "0:n" / "1:n" sont représentées à travers le référencement de la clef primaire de la table qui la possède, en clef étrangère au sein de la table à laquelle elle est liée. Si deux tables liées possèdent une cardinalité de type "0:n / 1:n", la relation sera traduite par la création d'une table de jonction, dont la clef primaire de chaque table deviendra une clef étrangère au sein de la table de jonction.

MLD



On retrouve les clés primaires annotées avec la mention (**PK = primary key**) et les clés étrangères (**FK = Foreign key**)

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 pour ne pas coder deux fois ma logique métier.

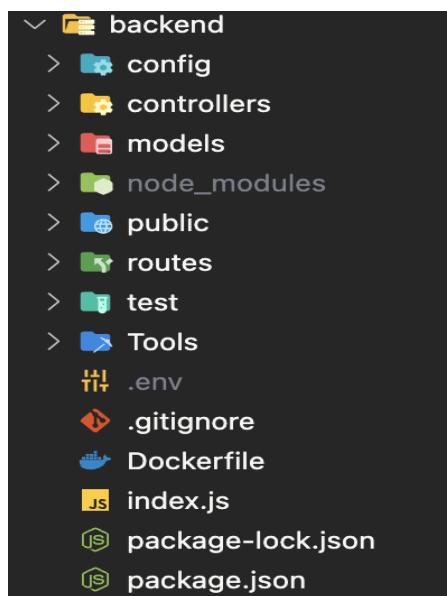
Dans le but de rendre mon backend plus efficace, je me suis concentrée sur la logique et l'optimisation de mon code.

J'ai donc effectué des recherches dans ce sens.

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

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 :

Mon back end est donc composé 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)
- **Tools** : contenant les fichiers qui regroupent les différentes opérations.
- **Models** : contenant les modèles de toutes mes tables (un par table et par fichier aussi), un fichier contient notre connexion à la base de données. Cette connexion est étendue ensuite dans tous les modèles (voir image suivante).
- **Public** : images uploadées depuis l'application mobile

```
const { Sequelize } = require("sequelize");
const { association } = require("./association");

const { MYSQL_LOCAL_PORT, MYSQL_ROOT, MYSQL_ROOT_PASSWORD, MYSQL_DATABASE } =
  process.env;
const sequelize = new Sequelize(
  `mysql://${MYSQL_ROOT}:${MYSQL_ROOT_PASSWORD}@localhost:${MYSQL_LOCAL_PORT}/${MYSQL_DATABASE}`
);
const modelDefiners = [
  require("./User"),
  require("./Qcm"),
  require("./UserQcm"),
  require("./Topic"),
  require("./Message"),
  require("./Question"),
  require("./Answer"),
  require("./Result"),
  require("./Theme"),
  require("./Type"),
  require("./Course"),
];
for (const modelDefiner of modelDefiners) {
  modelDefiner(sequelize);
}
```

Utilisation d'une API

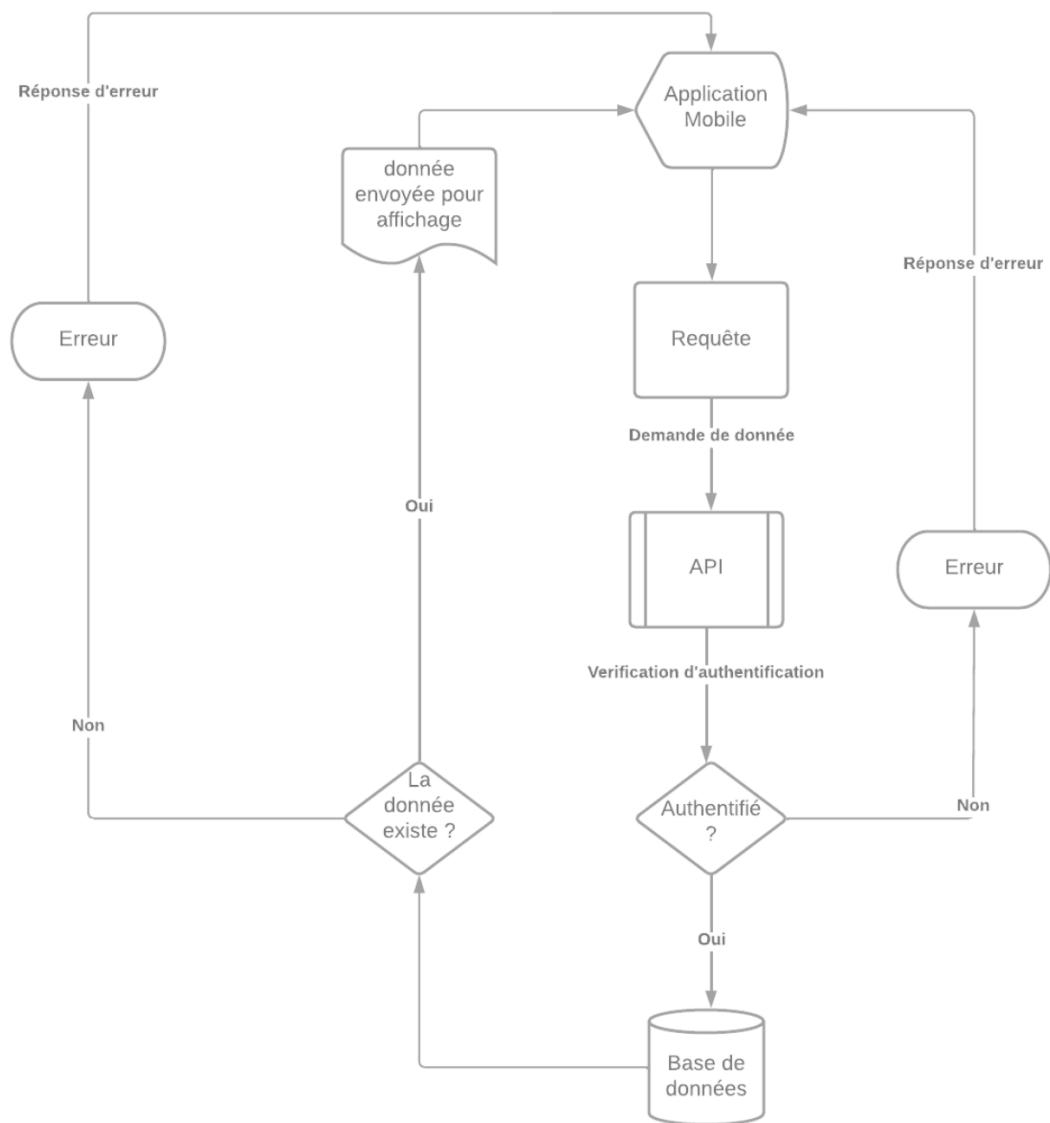
Une API, ou Interface de Programmation d'Application, offre la possibilité à différentes applications de communiquer entre elles et d'échanger des services ou des données de manière mutuelle.

Les interfaces de programmation d'application fournissent généralement un ensemble de fonctions qui simplifient l'accès aux services d'une application via un langage de programmation permettant d'envoyer des requêtes.

Lorsque le client envoie une requête sur mon API, le routeur analyse l'URL, et en

fonction de la route et de la méthode un Controller est appelé. Ce contrôleur/Controller 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.

Schéma de l'API



L'API en elle-même suit automatiquement une architecture de type REST, elle est stateless (c'est à dire qu'elle ne retient pas d'état à proprement parler,

elle ne stocke pas de données mais les fait simplement transiter via les requêtes). Pour les API destinées au web, on exploite le protocole HTTP et le système de statuts des réponses :

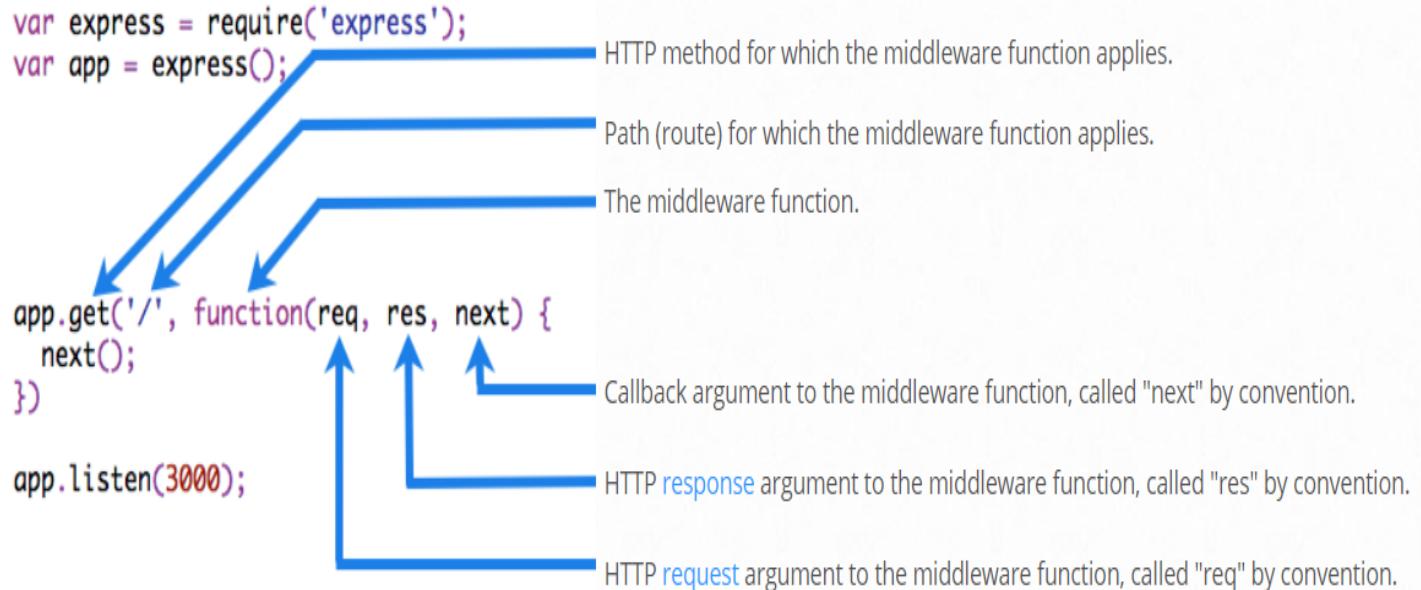
Code	Définition
200 : OK	<i>Indique que la requête a réussi. Cela signifie que la demande du client a été traitée avec succès par le serveur, et la réponse renvoyée contient les informations demandées.</i>
201 : CREATED	<i>Indique que la requête a réussi et qu'une ressource a été créée. Ce code est généralement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

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

Les différentes méthodes utilisées pour lire, écrire et modifier la donnée sont les suivantes :

- *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*

Fonctionnement des routes

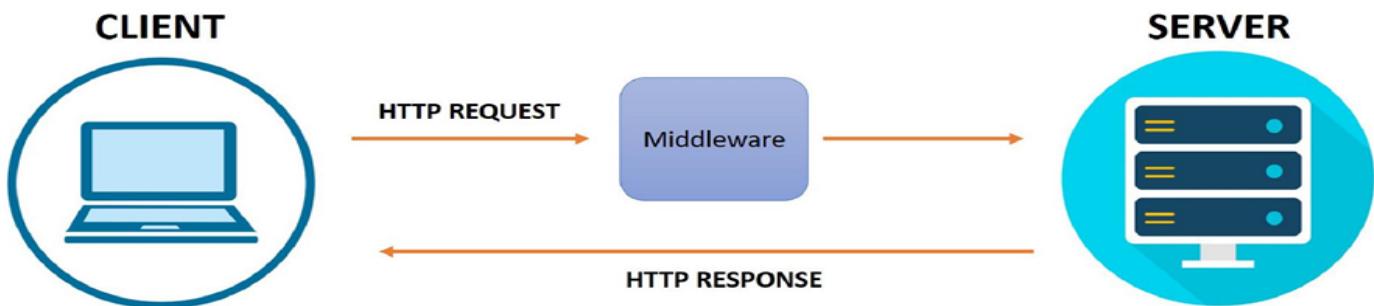


Lorsqu'une application Express reçoit une requête HTTP, elle crée deux objets : "req" contenant les informations de la requête, et "res" contenant les méthodes pour renvoyer une réponse. "req" est utilisé pour accéder aux données de la requête, telles que les paramètres d'URL ou les données du corps. "res" est utilisé pour envoyer une réponse, telle qu'une page HTML ou un objet JSON. Si un middleware est utilisé, il peut être appelé en utilisant la fonction "next", pour passer la requête au middleware suivant ou à la prochaine route correspondante.

Fonctionnement d'un middleware

Dans une application web construite avec Express, un middleware agit comme un composant logiciel intermédiaire exécuté entre la réception d'une requête et l'envoi d'une réponse. Son rôle consiste à effectuer différentes tâches telles que la validation des données, l'authentification des utilisateurs, la gestion des erreurs ou encore la compression des données.

Dans le contexte d'une application Express, les middlewares jouent un rôle clé en interceptant les requêtes et les réponses, leur permettant ainsi d'effectuer des opérations spécifiques avant de les transmettre à la route appropriée. Les middlewares sont généralement organisés en chaîne, ce qui signifie que chaque middleware peut agir sur la requête avant de la transmettre au middleware suivant, ainsi que sur la réponse avant de la renvoyer à l'utilisateur.



```
router.post('/', auth , postTopic);
```

Mon middleware ici correspond à la fonction auth()

```
const tokenAllStr = req.headers.cookies || req.headers.cookie ;
const tokenStr = tokenAllStr.split('=')[1];
try {
    const token = jwt.verify(tokenStr, SECRET_KEY);
    req.user = token;
    req.user.isAdmin = isAdmin(req.user);
    next();
} catch (error) {
    res.status(401).json({message: "Accès refusé"});
};
```

On vérifie le token de l'utilisateur dans le **middleware**, si tout est correct, on passe à la fonction suivante dans notre exemple ci-dessus il s'agit de la

méthode **`postTopic`** de mon controller Topics. Sinon on renvoie à l'utilisateur une **erreur 401**.

Si la fonction middleware en cours ne termine pas le cycle de demande-réponse, elle doit appeler la fonction **`next()`** pour transmettre le contrôle à la fonction middleware suivante. Sinon, la demande restera bloquée.

Les Controllers

Les controllers sont créés dans le dossier controllers de mon application, qui contenait toutes les logiques pour ce contrôleur spécifique. J'ai également exporté la fonction du contrôleur en utilisant **`module.exports`**.

Pour placer le contrôleur dans mon application Node.js, j'ai créé une route correspondante dans mon fichier de routes, qui faisait appel à la fonction du contrôleur.

Exemple: Si une erreur se produit lors de l'utilisation d'une des méthodes de Sequelize, on renvoie une erreur dans le bloc catch avec un statut 500.

En gérant les erreurs de cette manière, j'ai pu fournir des réponses claires et informatives aux utilisateurs de mon application, tout en assurant que les erreurs étaient correctement gérées et que mon application restait stable.



```
async function postTopic(req, res) {
  if (!req.body.data.title) {
    res.status(400).json("Les champs doivent être tous remplis");
  } else {
    const newTopic = {
      title: req.body.data.title,
      id_theme: req.body.data.id_theme,
      id_user: req.user.id,
    };

    await Topic.create(newTopic)
      .then((topic) => {
        res.status(201).json(topic);
      })
      .catch((err) => {
        res.status(500).json("Une erreur est survenue");
      });
  }
}
```

Conteneurisation du Backend

Nous avons décidé de conteneuriser notre backend à l'aide de docker pour plusieurs raisons.

Tout d'abord, Docker permet l'isolation et la portabilité de notre application, garantissant un déploiement cohérent et reproductible sur différentes machines. De plus, Docker simplifie le déploiement en encapsulant tous les composants nécessaires dans un conteneur unique, indépendamment de la configuration système. Dans notre cas, nous n'avions pas tous la même version de mysql et nous travaillions sur des OS différents.

Docker facilite également la scalabilité et la gestion des ressources en permettant la création de plusieurs instances de conteneurs et en fournissant des outils de surveillance et de gestion des performances.

Nous avons créé différentes commandes par le biais d'un script pour gérer notre container :

- **./start.sh run** : Lance le container MySQL et les fixtures (et création de base)
- **./start.sh makemigration** : Supprime les bases, et lance les fixtures
- **./start.sh stop** : Supprime les volumes et détruit les containers

En exécutant ces scripts, toute l'équipe était assurée de partir sur une même base avec des fixtures en base de données.

Connexion à la base de données

Pour se connecter à la base de données dockerisée, nous avons utilisé **DBeaver**. DBeaver est un outil de gestion de bases de données universel et gratuit. C'est un logiciel open-source qui prend en charge une large gamme de bases de données, y compris **MySQL**, PostgreSQL, Oracle, Microsoft SQL Server, SQLite, MongoDB, et bien d'autres.

DBeaver offre une interface graphique conviviale qui permet aux développeurs, administrateurs de bases de données et utilisateurs de travailler efficacement avec leurs bases de données. Il offre des fonctionnalités telles que l'exécution de requêtes SQL, la gestion des schémas et des tables, l'importation et l'exportation de données, la visualisation des résultats de requête, la création et la modification d'objets de base de données (tables, vues, procédures stockées, etc.).

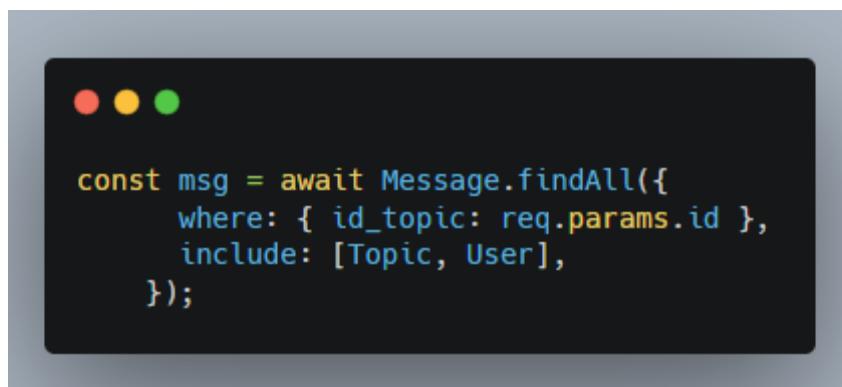
Sequelize

Sequelize est un **ORM** (Object-Relational Mapping) pour Node.js qui permet d'offrir une **couche d'abstraction** avec les bases de données relationnelles comme MySQL, PostgreSQL, SQLite... Un ORM est un outil qui facilite l'interaction entre le code d'un langage de programmation et les données stockées dans une base de données. Il permet de définir des modèles qui représentent les **entités** du domaine de l'application et qui sont mappés avec les tables de la base de données. Il permet aussi d'effectuer des requêtes sur les données en utilisant le langage du code plutôt que le langage SQL.

Définition des relations

Les relations avec **Sequelize** sont un moyen de définir les associations entre les entités. Il est donc important de définir correctement les relations entre vos modèles pour profiter de la génération automatique des clés étrangères dans les tables nécessaires, voire créer une table de liaison dans le cadre d'une relation de type **Many-To-Many**.

Sequelize fournira aussi les méthodes utiles pour accéder aux données associées, comme "**include**". Ce qui facilite grandement la manipulation des données.

A terminal window icon with three colored dots (red, yellow, green) at the top.

```
const msg = await Message.findAll({
  where: { id_topic: req.params.id },
  include: [Topic, User],
});
```

Les relations sont basées sur les clés étrangères qui sont déduites suite à la conception, notamment, au passage du MCD au MLD.

Sequelize propose quatre types de relations qui peuvent être combinés pour créer les relations standard : One-To-One, One-To-Many et Many-To-Many :

- La relation **HasOne** signifie qu'une relation One-To-One existe entre A et B, avec la clé étrangère définie dans le modèle cible (B).
- La relation **BelongsTo** signifie qu'une relation One-To-One existe entre A et B, avec la clé étrangère définie dans le modèle source (A).

- La relation **HasMany** signifie qu'une relation One-To-Many existe entre A et B, avec la clé étrangère définie dans le modèle cible (B).
- La relation **BelongsToMany** signifie qu'une relation Many-To-Many existe entre A et B, en utilisant une table de jonction C, qui aura les clés étrangères

Dans l'exemple suivant, il s'agit de définir la relation entre **User ⇔ Message**, ainsi que **Topic ⇔ Message**.



```
User.hasMany(Message, {foreignKey: 'id_user'});
Topic.hasMany(Message, {foreignKey: 'id_topic'});
```

Nous pouvons aussi personnaliser le nom des clés étrangères (avec *foreignKey*) ainsi générer. Ici, uniquement dans un souci d'harmonisation de nom de colonne dans la base de données.

Model

La méthode "init" est une méthode statique fournie par la classe "Model" de sequelize. Cette méthode est utilisée pour initialiser la définition de notre modèle de données.

Dans notre cas, j'ai utilisé cette méthode pour définir les différentes propriétés de notre entité "User". J'ai défini le type de données, le statut d'obligation ou non de la propriété, et j'ai également ajouté des fonctions de validation pour certaines propriétés.

J'ai également passé quelques options supplémentaires à cette méthode. Par exemple, j'ai fourni l'instance de "sequelize" que j'utilise pour la gestion de

la base de données et j'ai donné un nom à notre modèle pour faciliter sa référence ultérieurement.

Après avoir créé mon "model", j'ai dû effectuer une migration pour créer la table correspondante dans ma base de données. Pour cela, j'ai utilisé la commande "npx sequelize-cli migration:generate" pour générer un fichier de migration vide, que j'ai ensuite rempli avec le code nécessaire pour créer ma table.

Dans ce fichier de migration, j'ai spécifié le nom de ma table ainsi que les différentes colonnes que je voulais y ajouter en utilisant la syntaxe fournie par Sequelize.

J'ai également spécifié les types de données pour chaque colonne, ainsi que les contraintes de validation nécessaires.

Fixtures

Les fixtures sont des données de test que l'on peut charger dans une base de données à l'aide d'un ORM. Elles permettent de simuler des situations réelles et de vérifier le bon fonctionnement de l'application. Les fixtures sont utiles pour :

- Initialiser la base de données avec des données de base (par exemple, des rôles, des catégories, etc.)
- Remplir la base de données avec des données aléatoires ou réalistes pour tester les performances ou les fonctionnalités de l'application
- Créer des scénarios spécifiques pour tester des cas particuliers ou des erreurs
- Réinitialiser la base de données à un état connu avant ou après chaque test

Pour ce faire, nous avons eu une approche simple. Il s'agit d'un script exécutant les méthodes "create" de chaque entité avec un lien entre elles. Par exemple :

- On crée un utilisateur, puis on associe un message avec un texte 'placeholder' et avec un propriété '**id_user**' peuplée avec l'ID de l'utilisateur précédemment créé. L'ordre ici est important pour la

cohérence des données, on ne peut pas créer de message avec un utilisateur qui n'existe pas encore !

Un autre exemple illustré :

```
const QCM = await sequelize.models.Qcm.create({
  title: 'qcml',
  description:'Ceci est la description du qcm 1',
  createdAt: new Date(new Date() - Math.random()*(1e+12)),
  updatedAt: new Date(new Date() - Math.random()*(1e+12)),
  isGenerated: false,
  id_user: Admin.id,
  id_type: Type1.id})

for (let i = 1; i < 21; i++) {
  let question = await sequelize.models.Question.create(
    {
      text: 'Ceci est la question' + i + ' ?',
      createdAt: new Date(new Date() - Math.random()*(1e+12)),
      updatedAt: new Date(new Date() - Math.random()*(1e+12)),
      id_theme: Themel.id
    }
  )

  await sequelize.models.QcmQuestion.create({QcmId: QCM.id, QuestionId: question.id})
```

```
for (let j = 1; j < 6; j++) {
  if (j == 1) {
    let answer = await sequelize.models.Answer.create(
      {text: 'Ceci est la réponse' + j + ' juste a la question ' + i,
       createdAt: new Date(new Date() - Math.random()*(1e+12)),
       updatedAt: new Date(new Date() - Math.random()*(1e+12)),
       isCorrect_answer: true
      })
    sequelize.models.QuestionAnswered.create({
      QuestionId: question.id,
      AnswerId: answer.id})
  }

  else {
    let answer = await sequelize.models.Answer.create(
      {text: 'Ceci est la réponse' + j + ' fausse a la question ' + i,
       createdAt: new Date(new Date() - Math.random()*(1e+12)),
       updatedAt: new Date(new Date() - Math.random()*(1e+12)),
       isCorrect_answer: false
      })
    sequelize.models.QuestionAnswered.create(
      {QuestionId: question.id,
       AnswerId: answer.id
      })
  }
}
```

Plus compliqué, on crée un QCM, et on crée donc **20 questions associées à ce QCM** (une boucle avec une variable indentée), et pour chacune de ses questions, **5 réponses possibles dont une seule juste**.

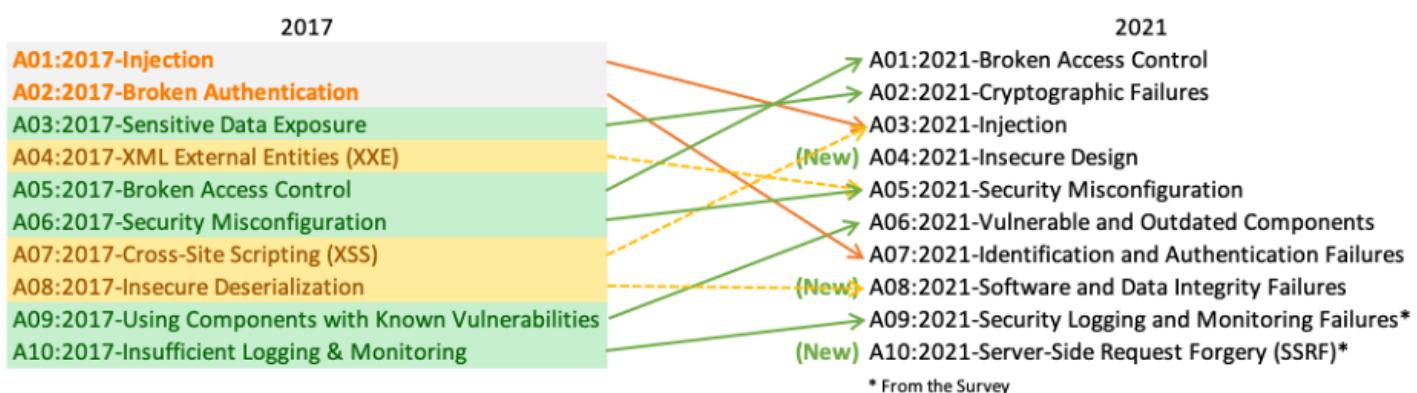
Sécurité

La sécurité a été un nerf central durant la conception et le développement de notre application.

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

C'est pour lutter contre les failles, sensibiliser et nous former que des structures comme OWASP existent.

OWASP (Open Web Application Security Project) est une communauté en ligne partageant des outils, documents et méthodes libres pour la sécurisation des applications web. Elle publie régulièrement des rapports permettant d'évaluer les plus fréquentes vulnérabilités rencontrées dans le web, propose des solutions pour garantir la sécurité des applications et données et assure son intégrité face à l'évolution des menaces.



Top 10 des vulnérabilités web publié par OWASP en 2021

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.
- Failles **XSS** ()
- **Credential stuffing** : vol du login et password
- **Attaque DDOS** : envoi de trafic en vue de surcharger le trafic d'une api
- **Man-in-the-middle** : consiste à pousser un utilisateur à se connecter à un service compromis, qui pourra alors s'emparer du jeton ou de la clé de l'utilisateur.

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

Injection SQL

Les attaques par injection de commandes SQL exploitent les failles de sécurité d'une application qui interagit avec des bases de données.

L'attaque SQL consiste à modifier une requête SQL en cours par l'injection d'un morceau de requête non prévu, souvent par le biais d'un formulaire. Le hacker peut ainsi accéder à la base de données, mais aussi modifier le contenu et donc compromettre la sécurité du système.

Pour s'en protéger, on a effectué plusieurs actions :

- La couche ORM transforme les données de la base de données en objets et vice-versa. L'utilisation d'une bibliothèque ORM réduit les requêtes SQL explicites et, par conséquent, est beaucoup moins vulnérable aux injections SQL.

```
const newTopic = {
  title: req.body.data.title,
  id_theme: req.body.data.id_theme,
  id_user: req.user.id,
};

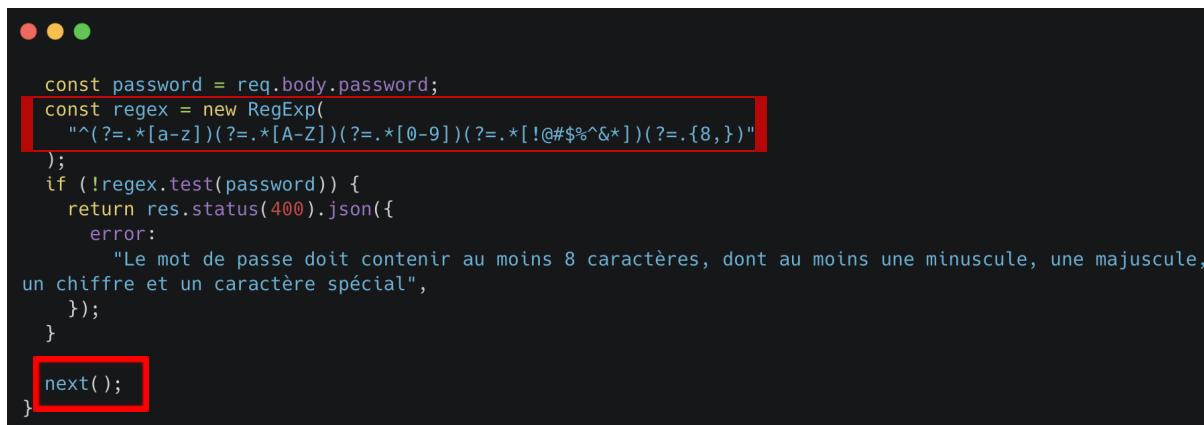
await Topic.create(newTopic)
  .then((topic) => {
    res.status(201).json(topic);
  })
```

Dans l'exemple ci-dessus, on a utilisé la fonction `create` de `sequelize` car elle utilise des paramètres liés pour les valeurs insérées dans la base de données, ce qui empêche les attaques d'injection SQL.

Credential stuffing : vol du login et password

Pour la sécurisation du mot de passe j'utilise deux fonctions :

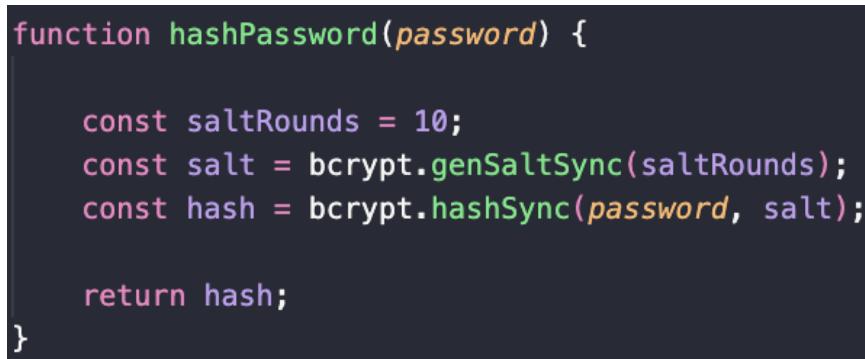
- La fonction **`ensurePasswordIsStrongEnough`** s'assure que le mot de passe respecte certains critères de complexité : il doit contenir au moins 8 caractères, dont au moins une lettre majuscule, une lettre minuscule, un chiffre et un caractère spécial. Si le mot de passe ne respecte pas ces critères, une erreur est levée.



```
const password = req.body.password;
const regex = new RegExp(
  "^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[!@#$%^&*])(?=.{8,})"
);
if (!regex.test(password)) {
  return res.status(400).json({
    error:
      "Le mot de passe doit contenir au moins 8 caractères, dont au moins une minuscule, une majuscule, un chiffre et un caractère spécial",
  });
}
next();
```

Une fois le mot de passe vérifié, on peut passer à la fonction suivante avec **`next()`**.

- 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.



```
function hashPassword(password) {

  const saltRounds = 10;
  const salt = bcrypt.genSaltSync(saltRounds);
  const hash = bcrypt.hashSync(password, salt);

  return hash;
}
```

Sécurisation avec TOKEN

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 hashée 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 lequel 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 rôles

Comme je l'ai expliqué précédemment dans la partie middleware, une gestion des rôles a été établie, ce qui rajoute une couche de sécurité supplémentaire pour ne pas laisser n'importe quel utilisateur accéder à des

données ou des actions sensibles. On vérifie le rôle de l'utilisateur en interceptant la requête et en analysant le rôle présent dans le jwt.

Utilisation de Helmet

ExpressJs est un framework robuste mais il n'est pas parfait en matière de sécurité, rien ne protège le serveur (node js) 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 reniffrage 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.

Autorisation des CORS Policy

```
// CORS (Cross Origin Resource Sharing) est un mécanisme permettant à des ressources

app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content, Accept, Content-Type, Authorization'
  );
  res.setHeader(
    'Access-Control-Allow-Methods',
    'GET, POST, PUT, DELETE, PATCH, OPTIONS'
  );
  next();
});
```

Les navigateurs modernes ont une politique de sécurité stricte appelée "Same-Origin Policy", qui limite les requêtes entre différents domaines. Cela signifie que si j'essaie d'effectuer une requête depuis mon serveur vers un autre domaine, cela pourrait être bloqué par le navigateur. Pour résoudre ce problème, j'utilise une technique appelée "CORS" (Cross-Origin Resource Sharing), qui me permet de spécifier les domaines qui sont autorisés à accéder à mes ressources. Dans ce code, j'utilise une fonction middleware Express pour définir les en-têtes CORS dans les réponses de mon serveur.

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.

On a utilisé Postman, un outil populaire pour tester et déboguer les APIs. Il y a aussi une fonctionnalité de documentation complète qui nous a permis de créer et de maintenir facilement la documentation de notre API.

Astrallys
COOP
New Collection
New Collection
Quiz connect
Users
Topics
Answer
Qcm
Course
Theme
Question
Search
Result

Quiz connect

Version CURRENT Language cURL ⚡

PUT update Open Request→

{{baseUrl}}users

Add request description...

Authorization Bearer Token

Token <token>

Body raw (json)

json

```
{  
  "id": 3,  
  "firstName": "Janee",  
  "lastName": "Doe",  
  "email": "janeeDoe@gmail.com",  
  "password": "toto"  
}
```

POST login Open Request→

{{fatimaUrl}}users/login

Add request description...

JUMP TO

Introduction
> Users
> Topics
> Answer
> Qcm
> Course
> Theme
> Question
> Search
> Result

Find and replace Console Runner Capture requests Cookies Trash ⓘ

Test unitaire

Un test unitaire est une technique de test logiciel qui consiste à vérifier le bon fonctionnement d'une unité de code isolée, généralement une fonction, une méthode ou une classe, de manière indépendante. L'objectif principal des tests unitaires est de s'assurer que chaque composant individuel du logiciel fonctionne correctement et produit les résultats attendus.

On a établi des tests unitaires avec le framework Jest. Nous l'avons utilisé parce qu'il est tout-en-un, il comprend une bibliothèque d'assertions, des fonctionnalités de mock intégrées, et un système de surveillance des modifications.

Il est facile à configurer et à utiliser, car il fournit une configuration par défaut pour les projets basés sur Node.js.

On a créé un dossier test dans lequel on va avoir un fichier par entité. On va tester chaque méthode présente dans les différents controllers

Dans l'exemple suivant, on teste la création d'un topic.

```

describe("postTopic", () => {
  let req, res;

  beforeEach(() => {
    req = {
      body: {
        data: {
          title: "Test Title",
          id_theme: 1,
        },
      },
      user: {
        id: 1,
      },
    };
    res = {
      status: jest.fn().mockReturnThis(),
      send: jest.fn(),
      json: jest.fn(),
    };
  });

  afterEach(() => {
    jest.restoreAllMocks(); // Restaure toutes les fonctions mock après chaque test
  });

  test("should create a new topic and return it as JSON", async () => {
    jest.spyOn(Topic, "create").mockResolvedValueOnce({
      id: 1,
      title: "Test Title",
      id_theme: 1,
      id_user: 123,
    });

    await postTopic(req, res);

    expect(res.status).toHaveBeenCalledWith(201);
    expect(res.json).toHaveBeenCalledWith({
      id: 1,
      title: "Test Title",
      id_theme: 1,
      id_user: 123,
    });
  });
});

```

La fonction **describe** de jest est utilisée pour regrouper les tests associés à **postTopic**. Cela permet d'organiser les tests et de les exécuter en groupe. On crée un **mock**, un objet ou une fonction simulée utilisée lors des tests unitaires pour remplacer une dépendance réelle. Ce mock dans notre cas reproduit le **body** d'une requête et sa réponse.

La fonction **afterEach** est exécutée après chaque test. Elle est utilisée pour réinitialiser toutes les fonctions mock utilisées par Jest en utilisant

`jest.restoreAllMocks()`. Cela garantit que chaque test est isolé et n'affecte pas les autres tests.

```
PASS  test/topics.test.js
postTopic
  ✓ should create a new topic and return it as JSON (1 ms)
  ✓ should send a 400 status and error message when title is missing
  ✓ should send a 500 status and error message when an error occurs during creation (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.387 s, estimated 1 s
Ran all test suites matching /test\topics.test.js/i.
```

Lorsque je développe une API Node.js Express, il est important de tester chaque route et chaque fonctionnalité pour m'assurer qu'elles fonctionnent correctement. Une façon de le faire est de tester l'API en utilisant une application tierce telle que Postman.

En utilisant Postman, je peux envoyer des requêtes HTTP aux différentes routes de mon API et visualiser les réponses renvoyées par l'API. Je peux également inclure des données de requête et de réponse, telles que des paramètres de requête, des corps de requête JSON et des codes d'état HTTP, pour m'assurer que l'API renvoie les résultats attendus.

En outre, Postman peut être utilisé pour tester des routes qui nécessitent une authentification. Je peux inclure les informations d'identification requises dans la requête et m'assurer que l'API les vérifie correctement avant d'autoriser l'accès à la ressource demandée.

The screenshot shows the Postman interface with the following details:

- HTTP Method:** POST
- URL:** {{fatimaUrl}}users/login
- Body:** Raw JSON (selected)
- JSON Content:**

```
1 {  
2   "email": "admin@admin.com",  
3   "password": "kktuyfg868tcf*778&"  
4 }
```

Dans l'exemple ci-dessous, on test la connexion si tout se passe bien, il renvoie un code 200 et selon le problème on renvoie un code d'erreur, ici on s'est trompé dans le mot de passe, c'est une 403.

The screenshot displays two panels of a test results interface, likely from a tool like Postman or similar. Both panels have tabs at the top: Body, Cookies (1), Headers (9), and Test Results (1/1). The Test Results tab is active in both.

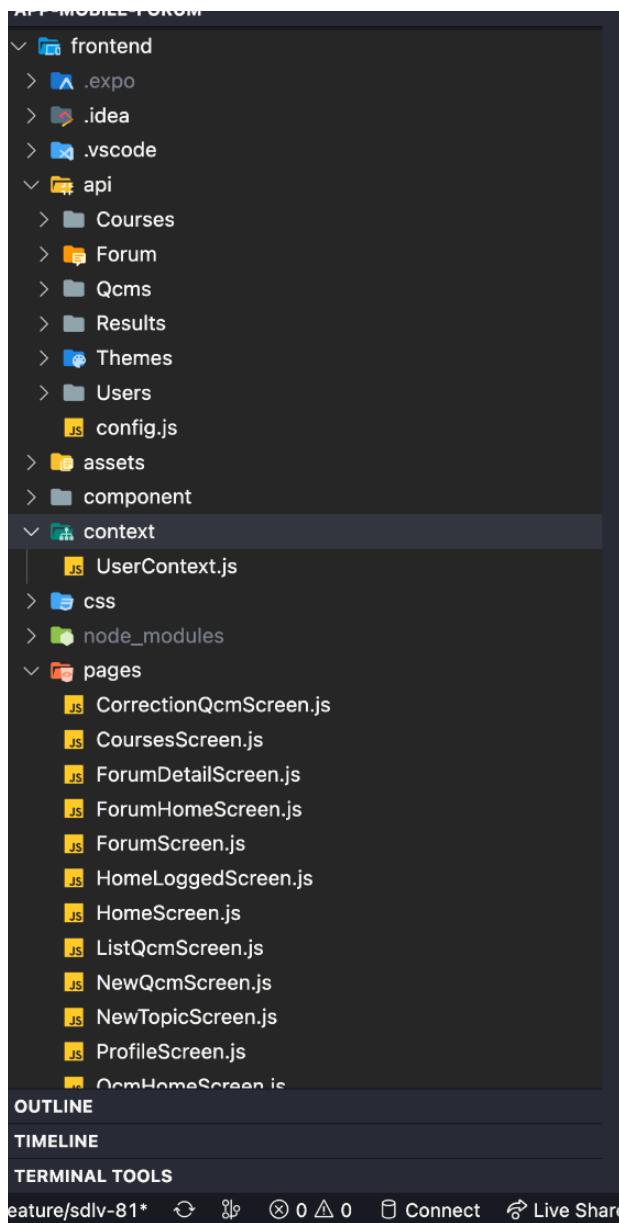
Top Panel: Shows a green "PASS" button and the text "Status code is 200".

Bottom Panel: Shows a red "FAIL" button and the text "Status code is 200 | AssertionError: expected response to have status code 200 but got 403".

Développement Du Front-end de l'application

Le projet étant réalisé sur React Native, le choix nous est laissé quant à l'organisation des dossiers et nous avons choisi un système dit atomique avec des organismes, molécules et atomes.

Arborescence



```
APP-MOBILE-FORUM
└── frontend
    ├── .expo
    ├── .idea
    ├── .vscode
    └── api
        ├── Courses
        ├── Forum
        ├── Qcms
        ├── Results
        ├── Themes
        ├── Users
        └── config.js
    ├── assets
    ├── component
    └── context
        ├── UserContext.js
        ├── css
        └── node_modules
    └── pages
        ├── CorrectionQcmScreen.js
        ├── CoursesScreen.js
        ├── ForumDetailScreen.js
        ├── ForumHomeScreen.js
        ├── ForumScreen.js
        ├── HomeLoggedScreen.js
        ├── HomeScreen.js
        ├── ListQcmScreen.js
        ├── NewQcmScreen.js
        ├── NewTopicScreen.js
        ├── ProfileScreen.js
        └── QcmHomeScreen.js

```

OUTLINE

TIMELINE

TERMINAL TOOLS

feature/sdlv-81* ⌂ ⌂ ⌂ 0 △ 0 ⌂ Connect ⌂ Live Share

- le dossier `src` contient tous les dossiers essentiels au projet.
- le dossier `components` contient tous les composants que l'on pourra utiliser/réutiliser.
- un dossier `api` contient les fichiers de classes qui nous permettront d'effectuer des actions vers l'API, qui est également fragmenté en plusieurs sous-dossiers en utilisant une instance `axios` que l'on verra plus tard.



```
import axiosInstance from '../config'

export const postQcm = async ( title, id_type ) => {
    const { data } = await axiosInstance.post(`qcms`, {
        title: title,
        isGenerated: false,
        id_type: id_type
    })

    try {

        if (data.status == 201) {
            console.log(data);
            console.log(data.data);
            logIn(data.status);
            return data
        }
    } catch (e) {
        switch (e) {
        case e.request:
            console.log(e.request)
            console.log(e.message)
            break
        case e.response:
            console.log(e.response)
            console.log(e.message)
            break
        default:
            console.log(e.config)
        }
    }

    return data
}
```

Page et composant

Un composant permet donc de pouvoir découper une page en éléments indépendants et réutilisables. Il y a plusieurs méthodes permettant de créer des composants. Nous avons choisi d'utiliser la plus communément utilisée aujourd'hui c'est celle des composants dits de fonction. Le composant est créé comme une fonction que l'on peut exporter et sa valeur de retour est l'ensemble des éléments qu'il devra faire apparaître à l'écran. Un composant accepte des props qui peuvent être réutilisés dans le code.

```
import { Card, Divider, Button, Surface } from "react-native-paper";

export default function CourseCardComponent(props) {
  const [themeTitle, setThemeTitle] = React.useState(
    props.course.Theme?.title || props.themeTitle
  );
  const [themeId, setThemeId] = React.useState(
    props.course.Theme?.id || props.course.id_theme
  );

  return (
    <Surface
      elevation={3}
      style={{
        ...styles().cardContainer,
        backgroundColor: themeColor[themeId],
      }}
    >
      <View>
        <Text style={styles().themeTitle}>{themeTitle}</Text>
        <Text style={styles().title}>{props.course.title}</Text>
      </View>
      <Surface style={styles().btn} elevation={3}>
        <Button
          icon={props.course.type === "pdf" ? "download" : "eye"}
          onPress={() => Linking.openURL(props.course.link)}
        >
          {props.course.type === "pdf" ? "Télécharger" : "Voir"}
        </Button>
      </Surface>
    </Surface>
  );
}
```

Dans l'exemple ci-dessus, j'ai créé un composant **CourseCardComponent**, qui me sert à afficher dans plusieurs **pages** comme sur la page **HomePageLogged** qu'on voit ci-dessous.

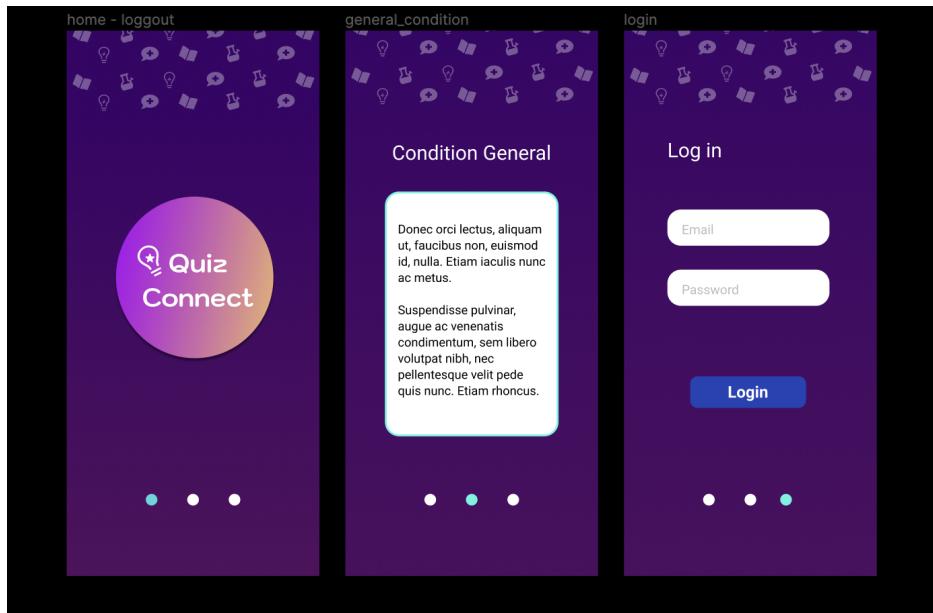
```
<LinearGradient
  colors={[ "purple", "#02254F", "#2D84EA" ]}
  style={styles.containerGradient}>
  <Image source={require("../assets/logo_fond.png")} style={styles.bgTop} />
  <View style={[ styles.profileImgContainer]}>...
  </View>
  <Text style={styles.sectionTitle}>Mes derniers cours</Text>
  <ScrollView style={{ display: "flex", flexDirection: "column" }}>
    <View style={styles.sectionLatestCourse}>
      {courses.length !== 0 &&
        courses.map((course, i) => {
          return <CourseCardComponent key={i} course={course} />;
        })
      }
    </View>
  </ScrollView>
</LinearGradient>
```

Certains composants ne sont pas réutilisés mais le fait de les concevoir comme un composant permet l'éventualité d'une modification simple et unique dans le fichier qui y est lié si jamais des modifications étaient à effectuer dans le code à l'avenir.

Navigation

Que l'on soit connecté ou non, la navigation à l'intérieur de l'application n'est pas la même .Dans un premier temps et comme déjà indiqué, nous avons souhaité faire en sorte que l'application ne soit pas accessible si nous ne sommes pas connectés. La page d'accueil de l'application nous laisse alors uniquement trois choix :

- L'écran d'accueil avec le logo et slogan
- La page des conditions général
- se rendre sur la page de connexion



Une fois connecté, nous arrivons donc sur la page d'accueil de l'utilisateur présentant plusieurs actions. On a accès à un bouton de navigation qui nous permet de se diriger vers différentes pages.

Menu

```

<UserContextProvider>
  <NavigationContainer>
    <Stack.Navigator>
      <Stack.Screen options={{ headerShown: false}} name="HomeScreen" component={HomeScreen} />
      <Stack.Screen options={{ headerShown: false, gestureEnabled: false}} name="HomeLoggedScreen" component={HomeLoggedScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ProfileScreen" component={ProfileScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ForumDetailScreen" component={ForumDetailScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ForumScreen" component={ForumScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ThemeScreen" component={ThemeScreen} />
      <Stack.Screen options={{ headerShown: false}} name="SectionChoiceScreen" component={SectionChoiceScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ForumHomeScreen" component={ForumHomeScreen} />
      <Stack.Screen options={{ headerShown: false}} name="NewTopicScreen" component={NewTopicScreen} />
      <Stack.Screen options={{ headerShown: false}} name="SearchByThemeScreen" component={SearchByThemeScreen} />
      <Stack.Screen options={{ headerShown: false}} name="CoursesScreen" component={CoursesScreen} />
      <Stack.Screen options={{ headerShown: false}} name="QcmHomeScreen" component={QcmHomeScreen} />
      <Stack.Screen options={{ headerShown: false}} name="ListQcmScreen" component={ListQcmScreen} />
      <Stack.Screen options={{ headerShown: false}} name="QuestionQcmScreen" component={QuestionQcmScreen} />
      <Stack.Screen options={{ headerShown: false, gestureEnabled: false}} name="ScoreScreen" component={ScoreScreen} />
      <Stack.Screen options={{ headerShown: false}} name="NewQcmScreen" component={NewQcmScreen} />
      <Stack.Screen options={{ headerShown: false}} name="CorrectionQcmScreen" component={CorrectionQcmScreen} />
      <Stack.Screen options={{ headerShown: false }} name="StatsScreen" component={StatsScreen} />
    </Stack.Navigator>
  <MenuButton/>
</NavigationContainer>
</UserContextProvider>

```

Nous avons utilisé **StackNavigator** qui fonctionne exactement comme une pile d'appels. Chaque écran vers lequel on navigue, est poussé vers le haut de la pile. Chaque fois qu'on appuie sur le bouton Retour, les écrans se détachent du haut de la pile.

Développement du Panel d'administration

J'ai créé un panel d'administration avec "Next.js" pour gérer l'application Quiz Connect. L'objectif était de permettre aux administrateurs et aux tuteurs de se connecter et de gérer de manière générale l'application mobile.

En utilisant les composants de "Next.js", j'ai créé une interface utilisateur simple et efficace qui permet aux administrateurs de naviguer facilement dans le panel d'administration. J'ai également mis en place des fonctionnalités de sécurité pour protéger les données sensibles, telles que l'authentification des administrateurs et la gestion des autorisations.

Pour sécuriser l'application, j'ai utilisé des technologies telles que JSON Web Tokens (JWT) et des middleware Express. Cela m'a permis de garantir que seuls les utilisateurs autorisés ont accès aux fonctionnalités de gestion des utilisateurs et des messages par exemple.

Les administrateurs de Quiz Connect peuvent désormais gérer efficacement les QCM, forum etc...

The screenshot shows the Quiz Connect administration panel. On the left, there's a sidebar with icons for Etudiants, Cours, Forums, Thèmes, and QCM, along with a Déconnexion button. The main area has a title 'Liste des étudiants' and a table with columns for ID, Prénom, Nom, E-mail, Mot de passe, and Rôle. There are three entries: 1. admin2, 2. user, and 3. user2. A modal window titled 'Ajouter un utilisateur' is open, containing fields for Prénom, Nom, E-mail, Mot de passe, Confirmer le mot de passe, and Rôle (with an ADMIN option selected). At the bottom of the modal are 'ENREGISTRER' and 'ANNULER' buttons. To the right of the modal, there's a table with columns for Photo de profil, Date de création, and Date de modification, showing three user profiles with placeholder data.

Utilisation de Next.js



J'ai choisi d'utiliser Next.js pour plusieurs raisons. Tout d'abord, Next.js est un framework JavaScript très populaire pour le développement d'applications web côté serveur et côté client. Next.js offre une approche basée sur les composants, ce qui facilite la création d'interfaces utilisateur réactives et modulaires.

Ensuite, Next.js fournit une architecture solide pour le développement d'applications universelles. Il prend en charge le rendu côté serveur (SSR) ce qui signifie que vos pages web sont pré-rendues sur le serveur et envoyées au navigateur, ce qui permet d'améliorer considérablement les performances de l'application car les utilisateurs peuvent voir rapidement le contenu de base sans attendre que le JavaScript s'exécute côté client.

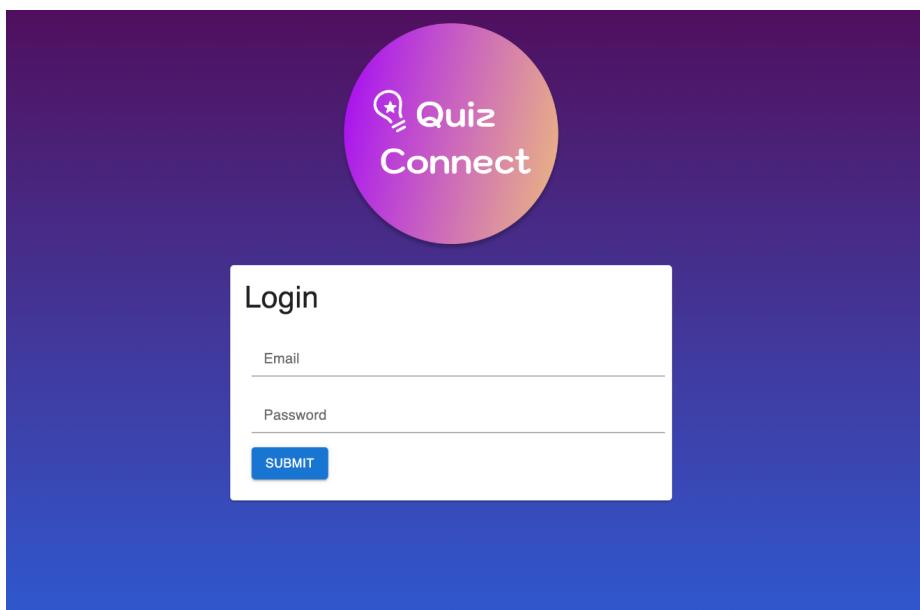
De plus, Next.js offre des fonctionnalités avancées telles que la prise en charge du routing : Next.js offre un système de routage facile à utiliser, ce qui facilite la création de routes pour vos pages web. On peut définir des routes dynamiques, des routes imbriquées et des routes basées sur des fichiers.

Un autre avantage de Next.js est l'intégration de React : Next.js est construit sur la base de React, ce qui signifie que l'on peut profiter de toutes les fonctionnalités puissantes de React pour la construction d'interfaces utilisateur réactives. On peut utiliser des composants React, le modèle de programmation déclaratif et la gestion de l'état de React dans votre application Next.js.

Également un écosystème riche : Next.js bénéficie d'un écosystème riche et dynamique. On peut utiliser de nombreuses bibliothèques et packages

existants de la communauté React pour ajouter des fonctionnalités supplémentaires à votre application.

Page de connexion



Pour permettre aux administrateurs de se connecter, j'ai créé une page de connexion qui prend en compte l'email et le mot de passe des utilisateurs. Les administrateurs sont les seuls utilisateurs ayant un accès complet au panel d'administration. Ainsi, j'ai implémenté une vérification dans la base de données pour s'assurer que seul un utilisateur ayant un rôle "admin" peut se connecter en tant qu'administrateur.

Une fois qu'un administrateur se connecte avec succès, il peut accéder à toutes les fonctionnalités du panel d'administration, notamment la gestion des utilisateurs et des QCM. J'ai utilisé les composants de Next.js pour créer une interface utilisateur claire et facile à utiliser qui permet aux

administrateurs de visualiser, modifier ou supprimer des données ciblées de l'application mobile Quiz Connect selon les besoins.

En plus de la page de connexion, j'ai mis en place des fonctionnalités de sécurité pour protéger les données sensibles. J'ai utilisé des technologies telles que JSON Web Tokens (JWT) pour générer des jetons d'authentification qui garantissent que les administrateurs connectés ont accès uniquement aux fonctionnalités de gestion de l'application Quiz Connect . De plus, j'ai utilisé des middleware Express pour contrôler l'accès aux routes du panel d'administration et garantir que seuls les administrateurs connectés peuvent accéder aux fonctionnalités de gestion.

Page de gestion des utilisateurs

Liste des étudiants										
	ID	Prénom	Nom	E-mail	Rôle	Photo de profil	Date de création	Date de modification	Modifier	Supprimer
<input type="checkbox"/>	1	admin	admin	admin@admin.com	(ROLE_ADMIN) (ROLE_TUTOR)		25/01/2023	13/02/2023 10:21:49	<button>MODIFIER</button>	<button>SUPPRIMER</button>
<input type="checkbox"/>	2	user	user	user@user.com	(ROLE_TUTOR)		25/01/2023	25/01/2023 15:52:02	<button>MODIFIER</button>	<button>SUPPRIMER</button>
<input type="checkbox"/>	3	user2	user2	user2@user2.com	(ROLE_STUDENT)		25/01/2023	26/01/2023 11:39:38	<button>MODIFIER</button>	<button>SUPPRIMER</button>
<input type="checkbox"/>	4	lucien	zak	test@test.com	(ROLE_STUDENT)		17/02/2023	17/02/2023 16:19:51	<button>MODIFIER</button>	<button>SUPPRIMER</button>

J'ai créé une page d'administration des utilisateurs qui permet aux administrateurs de visualiser et de gérer les profils des utilisateurs de Quiz Connect . En utilisant les composants de Next.js, j'ai créé une interface utilisateur efficace qui récupère la liste de tous les utilisateurs enregistrés dans la base de données. Les administrateurs peuvent facilement visualiser les informations des utilisateurs, notamment leurs emails, leurs noms et prénoms et leurs rôles.

Les administrateurs peuvent également modifier les informations des utilisateurs. En cliquant sur le bouton de modification, les administrateurs peuvent mettre à jour les champs d'email, de prénom et de mot de passe des utilisateurs. Pour garantir la sécurité et l'intégrité des données, seuls les administrateurs ont accès à cette fonctionnalité dû à leurs rôles.

En plus de la modification des profils des utilisateurs, les administrateurs peuvent également supprimer les utilisateurs. Pour supprimer un utilisateur,

les administrateurs cliquent simplement sur le bouton de suppression correspondant à l'utilisateur concerné.

également, les administrateurs ont la possibilité de créer un utilisateur en cliquant sur le bouton d'ajout, Cela aura pour effet d'inscrire un étudiant qui pourra se connecter par la suite via les identifiants qu'il aura communiquées aux équipes RH.

Grâce à cette page d'administration des utilisateurs, les administrateurs de Quiz Connect peuvent facilement visualiser et gérer les profils des étudiants avec des fonctionnalités de sécurité en place, garantissant ainsi la sécurité et l'intégrité des données.

Explication de la logique postUser



```
import axiosInstance from '../config'

export const postUser = async (firstName, lastName, email, password, role) => {
    const { data } = await axiosInstance.post('users', {
        firstName,
        lastName,
        email,
        password,
        role
    })
    try {
        if (data.status == 201) {
            console.log(data);
            console.log(data.data);
            logIn(data.status);
            return data
        }
    } catch (e) {
        switch (e) {
            case e.request:
                console.log(e.request)
                console.log(e.message)
                break
            case e.response:
                console.log(e.response)
                console.log(e.message)
                break
            default:
                console.log(e.config)
        }
    }
    return data
}
```

J'ai créé une fonction appelée postUser qui me permet de créer un utilisateur en envoyant une requête POST.

Lorsque j'appelle cette fonction, j'envoie une demande à l'URL qui est représentée par mon **axiosInstance** qui est la route fournie par mon api. à la suite de ça je lui ajoute la méthode "post" qui permet de créer une entrée dans ma base de données, puis je précise à ma route que je souhaite interagir avec la ta table 'users' et je lui passe les paramètres dont la route à besoin pour créer l'entité, en l'occurrence les informations de l'utilisateur récupérées dans le formulaire de création.

Ensuite, j'attends la réponse de la requête en utilisant await devant ma requête, ce qui me permet d'obtenir les données renvoyées par le serveur au format JSON selon le statut de la réponse. Si elle est 201, l'entité se crée et je récupère ses données, sinon la fonction me retournera une erreur avec la cause de pourquoi la requête a échoué.

```
● ● ●

const handleSubmit = async (event) => {
    event.preventDefault();
    if (password !== confirmPassword) {
        setMessage('Les mots de passe ne correspondent pas');
        setOpen(true);
        return;
    }
    try {
        const response = await postUser(firstName, lastName, email, password, role);
        console.log(response);
        setMessage('Utilisateur créé avec succès');
        setTimeout(() => {
            props.onClose()
        }, 2000);
    } catch (error) {
        console.log(error);
        if (error.response.status === 404) {
            setMessage('Cette adresse email est déjà utilisée');
        } else if (error.response.status === 406) {
            setMessage('Cette adresse email n\'est pas valide');
        } else if (error.response.status === 400) {
            setMessage('Tous les champs doivent être remplis');
        } else {
            setMessage('Erreur lors de la création de l\'utilisateur');
        }
    }
    setOpen(true);
};
```

Une fois cela fait, je peux réutiliser cette fonction à chaque fois que je crée un utilisateur dans ma fonction handleSubmit à laquelle j'envoie les informations de l'utilisateur pour que la fonction **postUser** les récupère et cette fois-ci, si le résultat est bon, j'envoie un message de succès qui dit que l'utilisateur a bien été créé, sinon un message d'erreur apparaît selon l'erreur relevée. ça peut être de remplir les champs manquants ou que l'email est déjà utilisé par exemple.

Explication de la logique deleteUser



```
import axiosInstance from '../config'

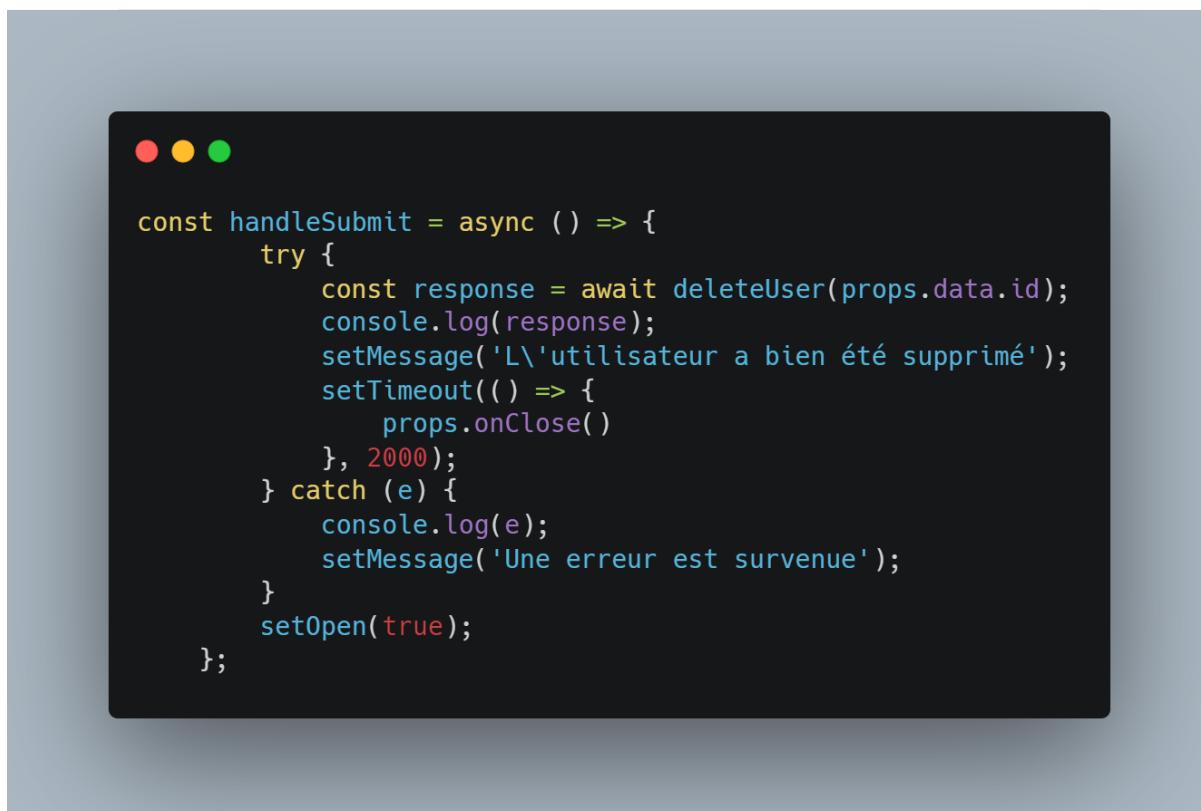
export const deleteUser = async (userID) => {

    const { data } = await axiosInstance.delete(`users`, {
        data: {
            id: userID
        }
    })
    try {
        if (data.status == 200) {
            console.log(data);
            console.log(data.data);
            return data
        }
    } catch (e) {
        switch (e) {
            case e.request:
                console.log(e.request)
                console.log(e.message)
                break
            case e.response:
                console.log(e.response)
                console.log(e.message)
                break
            default:
                console.log(e.config)
        }
    }
}
```

J'ai créé une fonction appelée deleteUser qui me permet de supprimer un utilisateur en envoyant une requête DELETE.

Lorsque j'appelle cette fonction, j'envoie une demande à l'URL qui est représentée par mon **axiosInstance** qui est la route fournie par mon api. à la suite de ça, je lui ajoute la méthode "delete" qui permet de créer une entrée dans ma base de données, puis je précise à ma route que je souhaite interagir avec la table 'users' et je lui passe les paramètres dont la route a besoin pour supprimer l'entité, en l'occurrence l'id qui représente l'utilisateur qui a été récupéré.

Ensuite, j'attends la réponse de la requête en utilisant await devant ma requête, ce qui me permet d'obtenir les données renvoyées par le serveur au format JSON selon le statut de la réponse; Si elle est 201, l'entité se supprime sinon la fonction me retournera une erreur avec la cause pour laquelle la requête a échoué.



```
const handleSubmit = async () => {
    try {
        const response = await deleteUser(props.data.id);
        console.log(response);
        setMessage('L\'utilisateur a bien été supprimé');
        setTimeout(() => {
            props.onClose()
        }, 2000);
    } catch (e) {
        console.log(e);
        setMessage('Une erreur est survenue');
    }
    setOpen(true);
};
```

Une fois fait, je peux réutiliser cette fonction à chaque fois que je supprime un utilisateur dans ma fonction handleSubmit à laquelle j'envoie l'id de l'utilisateur

pour que la fonction **deleteUser** les récupère et cette fois-ci, si le résultat est bon, j'envoie un message de succès qui dit que l'utilisateur a bien été supprimé. Sinon un message d'erreur apparaît pour signaler qu'une erreur est survenue lors de la suppression.

Explication de la logique putUser

Pour modifier les informations des utilisateurs, j'utilise la même logique que celle de la création à la différence que j'utilise la méthode "PUT". Et je lui passe les paramètres dont la route a besoin pour modifier l'entité, en l'occurrence les informations de l'utilisateur qui ont été récupérées dans le formulaire.

J'utilise cette même logique pour gérer tout mon panel admin, que ce soit les QCM, les forums, les cours etc...

Recherche anglophone

Context

On a voulu établir des relations N-N avec l'ORM sequelize, de ce fait j'ai dû faire des recherches en anglais pour affiner les résultats les plus pertinents. Grâce à la documentation de sequelize j'ai pu trouvé une solution.

Creating associations in Sequelize is done by calling one of the `belongsTo` / `hasOne` / `hasMany` / `belongsToMany` functions on a model (the source), and providing another model as the first argument to the function (the target).

- `hasOne` - adds a foreign key to the target and singular association mixins to the source.
- `belongsTo` - add a foreign key and singular association mixins to the source.
- `hasMany` - adds a foreign key to target and plural association mixins to the source.
- `belongsToMany` - creates an N:M association with a join table and adds plural association mixins to the source. The junction table is created with `sourceld` and `targetId`.

Creating an association will add a foreign key constraint to the attributes. All associations use `CASCADE` on update and `SET NULL` on delete, except for n:m, which also uses `CASCADE` on delete.

When creating associations, you can provide an alias, via the `as` option. This is useful if the same model is associated twice, or you want your association to be called something other than the name of the target model.

As an example, consider the case where users have many pictures, one of which is their profile picture. All pictures have a `userId`, but in addition the user model also has a `profilePictureId`, to be able to easily load the user's profile picture.

```
User.hasMany(Picture)
User.belongsTo(Picture, { as: 'ProfilePicture', constraints: false })

user.getPictures() // gets you all pictures
user.getProfilePicture() // gets you only the profile picture

User.findAll({
  where: ...,
  include: [
    { model: Picture }, // load all pictures
    { model: Picture, as: 'ProfilePicture' }, // load the profile picture.
    // Notice that the spelling must be the exact same as the one in the association
  ]
})
```

To get full control over the foreign key column added by Sequelize, you can use the `foreignKey` option. It can either be a string, that specifies the name, or an object type definition, equivalent to those passed to `Sequelize.define`.

```
User.hasMany(Picture, { foreignKey: 'uid' })
```

The foreign key column in Picture will now be called `uid` instead of the default `userId`.

Traduction

Créer des associations dans Sequelize se fait en appelant l'une des fonctions `belongsTo` / `hasOne` / `hasMany` / `belongsToMany` sur un modèle (la source), et en fournissant un autre modèle en premier argument de la fonction (la cible).

hasOne - ajoute une clé étrangère à la cible et des mixins d'association singulières à la source.

belongsTo - ajoute une clé étrangère et des mixins d'association singulières à la source.

hasMany - ajoute une clé étrangère à la cible et des mixins d'association plurielles à la source.

belongsToMany – crée une association N:M avec une table de jointure et ajoute des mixins d'association plurielles à la source. La table de jonction est créée avec sourceId et targetId.

Pour avoir un contrôle total sur la colonne de clé étrangère ajoutée par Sequelize, vous pouvez utiliser l'option `foreignKey`. Cela peut être soit une chaîne de caractères qui spécifie le nom, soit une définition de type objet, équivalente à celles passées à `sequelize.define`.

La création d'une association ajoutera une contrainte de clé étrangère aux attributs. Toutes les associations utilisent **CASCADE** lors de la mise à jour et **SET NULL** lors de la suppression, sauf pour **n:m**, qui utilise également CASCADE lors de la suppression.

Lors de la création d'associations, vous pouvez fournir un alias, via l'option `as`. Cela est utile si le même modèle est associé deux fois, ou si vous souhaitez que votre association soit appelée autrement que le nom du modèle cible.

À titre d'exemple, considérons le cas où les utilisateurs ont de nombreuses images, dont l'une est leur image de profil. Toutes les images ont un `userId`, mais en plus le modèle utilisateur a également un `profilePictureId`, pour pouvoir charger facilement l'image de profil de l'utilisateur.

Conclusion

En conclusion, le projet Quiz Connect était une expérience passionnante et enrichissante pour moi. En collaboration avec mes collègues étudiants, nous avons créé une application mobile, Quiz Connect, pour permettre aux étudiants en médecine de s'entraîner pour leurs examens et réviser leurs cours.

Cela m'a beaucoup apporté, tant dans la découverte de nouveaux frameworks et architectures que dans le travail en équipe. On avait différentes méthodes de travail, mais on a su s'accorder pour réaliser ce projet et trouver le bon rythme.

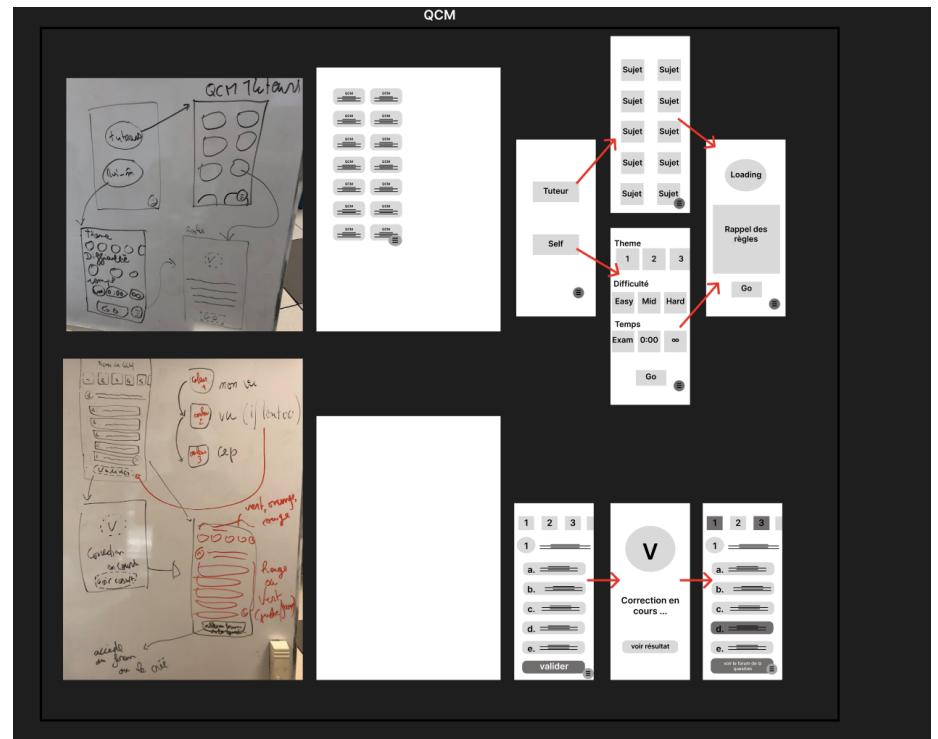
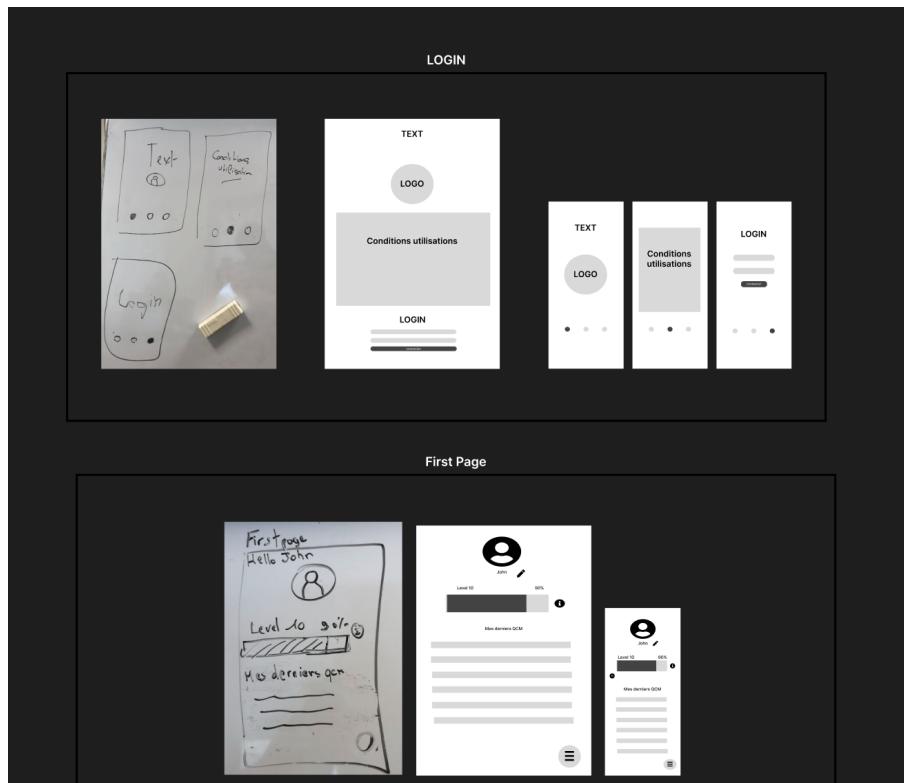
J'ai pu également dans le cadre de mon alternance développer d'autres API notamment avec symfony (API Platform). Cela m'a permis de voir une architecture de projet et une mise en œuvre de la sécurité différente.

Je suis reconnaissant d'avoir eu cette opportunité et j'ai hâte de continuer à développer mes compétences dans ce domaine passionnant.

Annexes

Maquettage

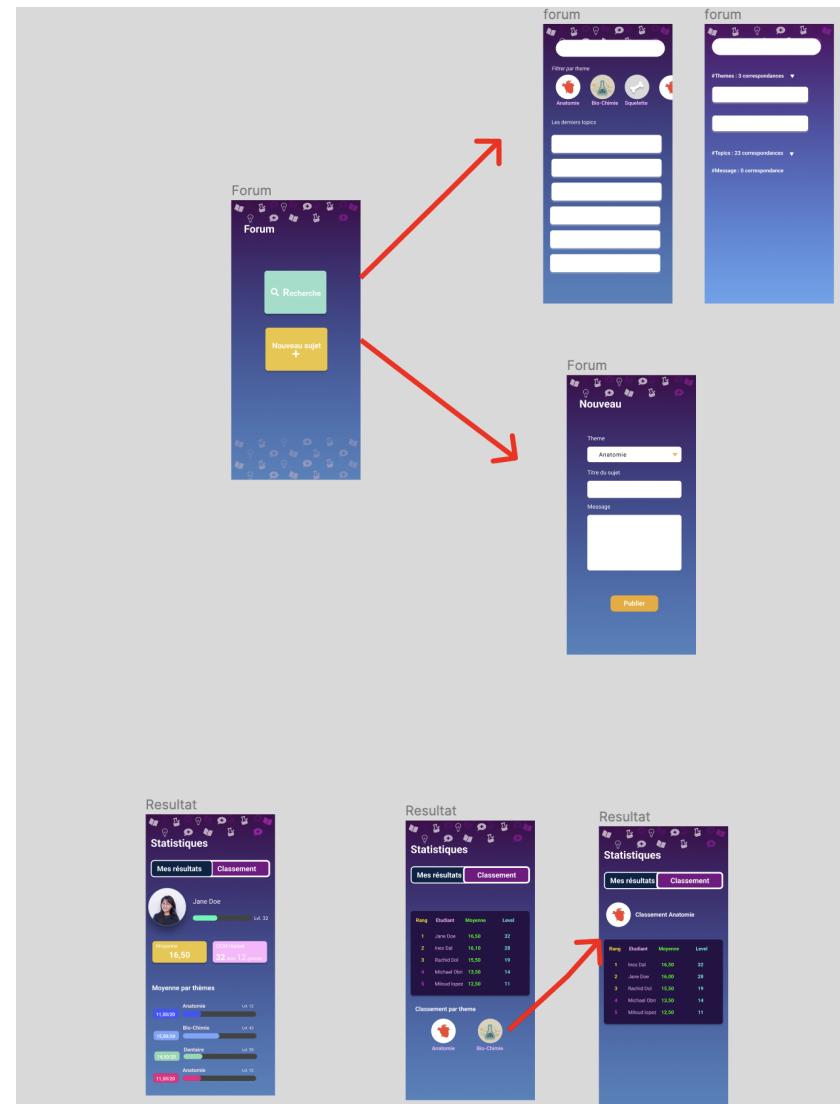
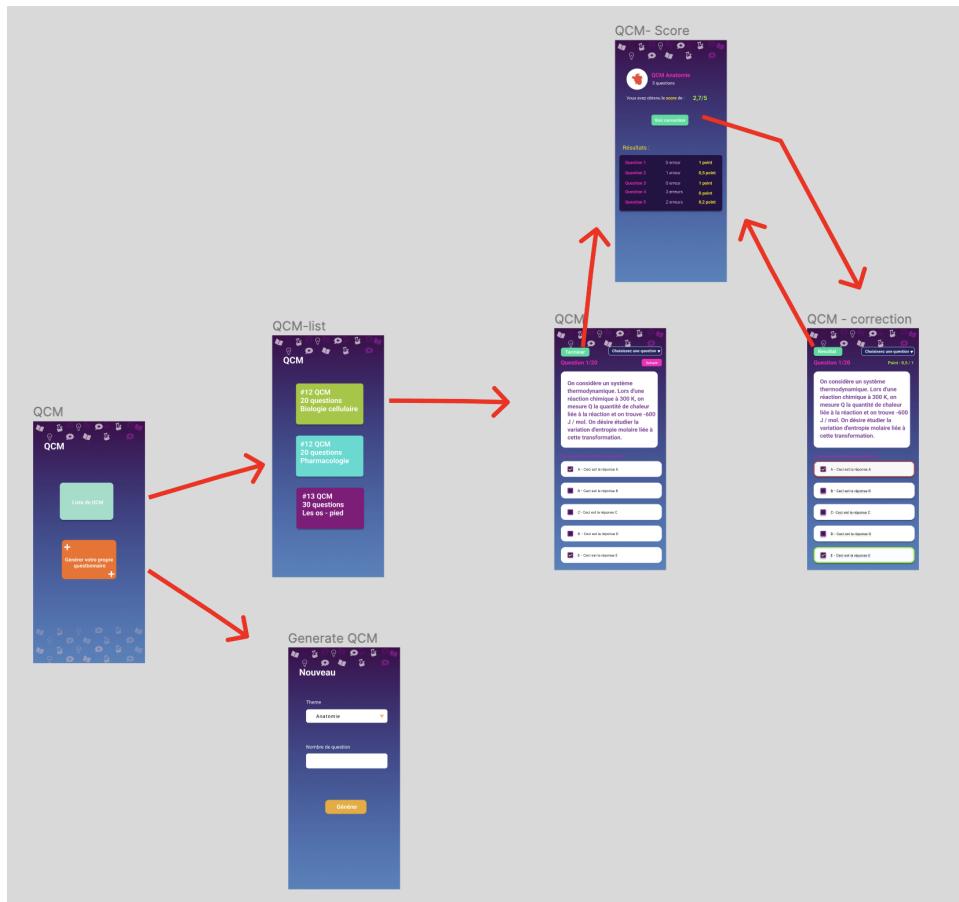
Wireframe



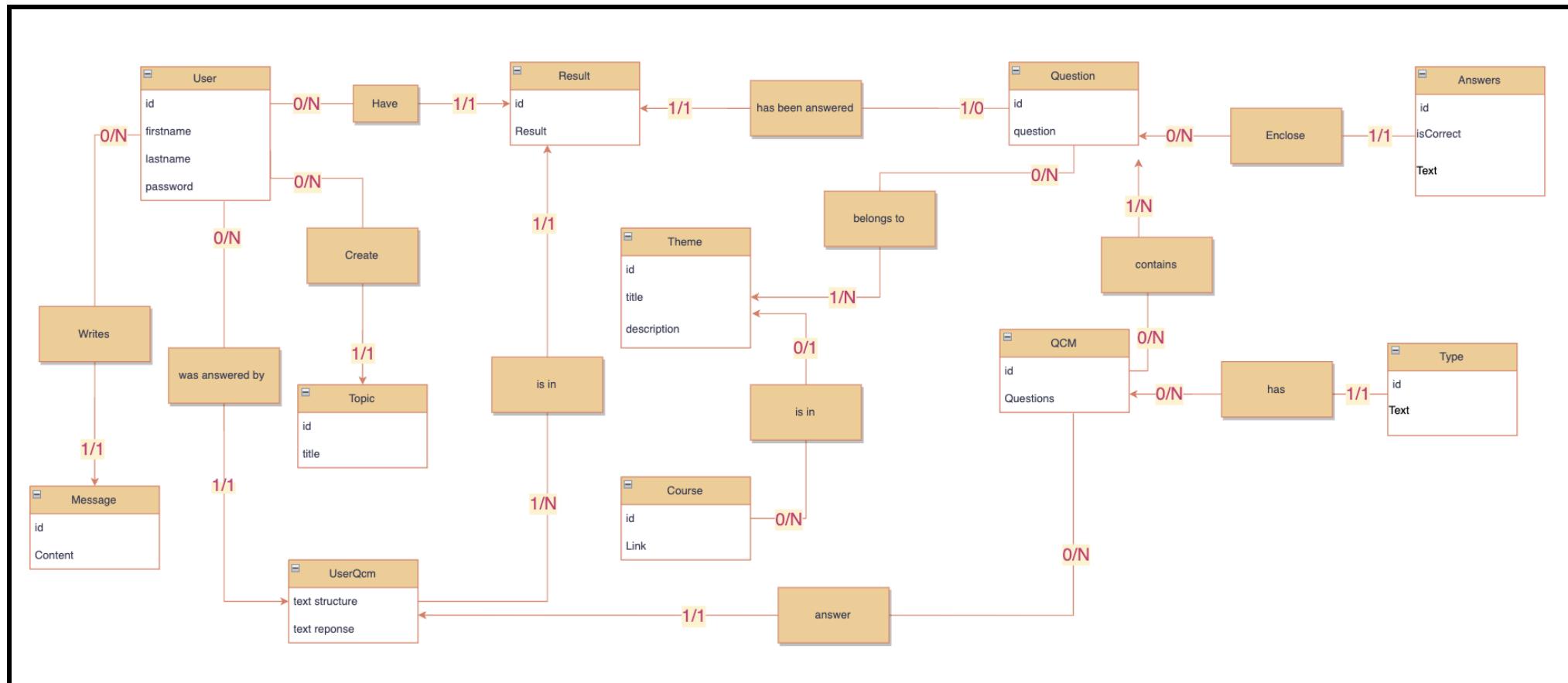
Maquette haute fidélité

The wireframe illustrates the user flow through the Quiz Connect application:

- Initial Screens:**
 - home - logout**: Shows the Quiz Connect logo.
 - general_condition**: Shows a "Condition General" card with placeholder text: "Donec erat lectus, aliquam ut, faucibus non euismod id nulla. Etiam lacinia nunc ac metus. Suspendisse pulvinar, augue ac venenatis condimentum, sem libero volutpat nibh, nec pellentesque velit pede quis nunc. Etiam rhoncus."
 - login**: A login form with fields for Email and Password, and a Login button.
- Logged-in User Screens:**
 - theme**: Shows a sidebar with "Themes" categories: Anatomie, Bio-Chimie, Squelette, and another Anatomie category.
 - home - logged**: Shows a profile picture of "Jolie Doe" (Level 12) and a "Mes derniers cours" section with three course cards.
 - profile user**: A profile edit screen with fields for Nom, Prenom, Email, and Password, and a Modifier button.
- Content Generation and Selection:**
 - QCMs**: A list of QCMs under the "Anatomie" theme:
 - #12 QCM 30 questions Les os - main
 - #13 QCM 30 questions Les os - pied
 - Choice Page**: A central page for selecting content:
 - Cours (green button)
 - Forum (pink button)
 - OQM (yellow button)
 - Forum filtre theme**: A results page for the "Anatomie" theme, listing three course cards.
- Navigation Flow:**
 - An arrow points from the "Courses" screen down to the "Choice Page".
 - Two arrows point from the "Choice Page" to the "Forum filtre theme" screen: one from the top and one from the bottom.
 - A red arrow points from the "Courses" screen to the "Forum filtre theme" screen.



Modèle conceptuel de données



Modèle logique de donnée

