AI13 : RAPPORT DE PROJET

Développement d'une application web d'évaluation de stagiaires

LIEN DU GIT

Semestre A24

BENYAGOUB Imène - DELANNOY Jules - HASSAN Yousra - RAFEI Jana

CONTEXTE

Dans le cadre de l'UV AI13, nous avons travaillé tout au long du semestre sur la conception et la réalisation d'une application web dédiée à l'évaluation des compétences de stagiaires. Cette application permet l'évaluation à travers la mise à disposition de questionnaires à choix multiples.

Ce projet s'inscrit dans une démarche pédagogique visant à maîtriser les outils de développement web modernes tout en répondant à un besoin réel en gestion et évaluation de compétences.

Utilisateurs

Trois types d'utilisateurs ont accès à l'application :

- Visiteur : accès limité, consultation des informations générales sur les questionnaires et les statistiques simples associées
- Stagiaire : utilisateur pouvant répondre aux questionnaires, visualiser ses résultats et gérer son profil
- Administrateur : utilisateur responsable de la gestion de l'application (utilisateurs, questionnaires et ressources associées)

Cas d'utilisation

Le diagramme des cas d'utilisation de l'application (simplifié aux fonctionnalités principales) est donné par la figure suivante :

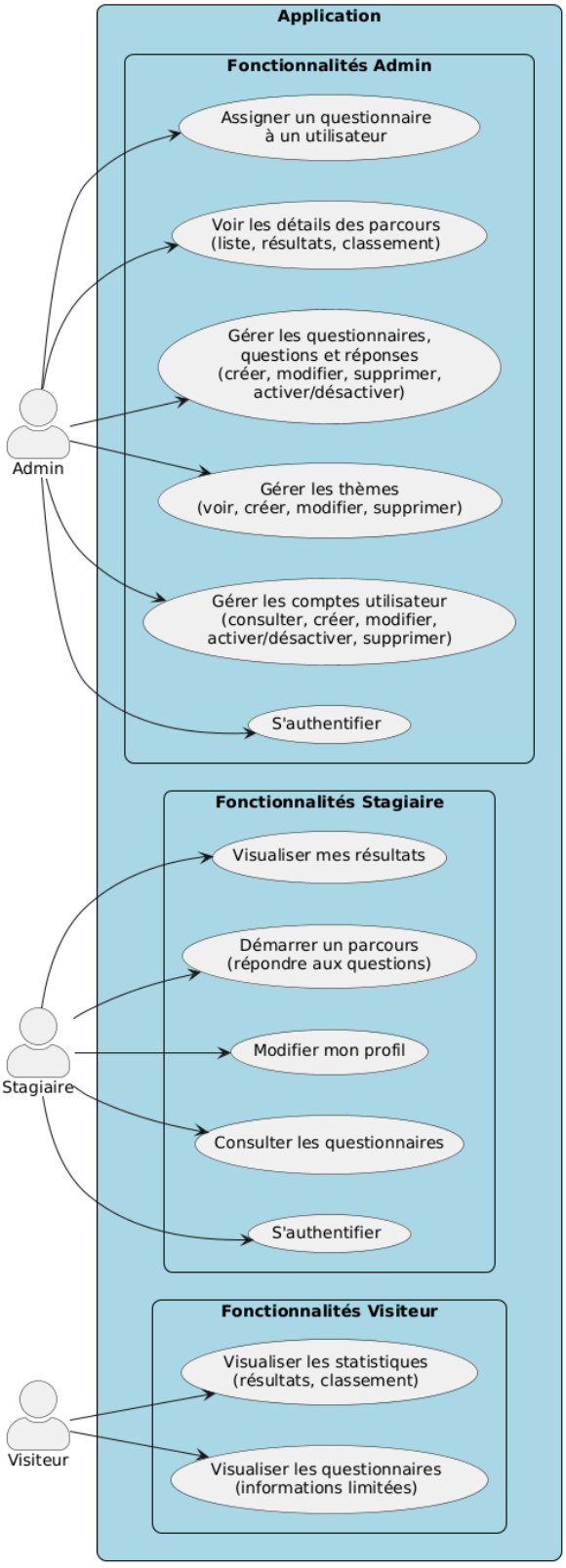

Figure 1 : Diagramme simplifié des cas d'utilisation

Technologies utilisées

L'application repose sur l'utilisation des solutions technologiques suivantes :

- Back-end : Java Spring (REST API) avec Hibernate pour la gestion des données
- Front-end : ReactJS pour l'interface utilisateur des stagiaires
- Base de données : MySQL avec phpMyAdmin

1. ARCHITECTURE ET CONCEPTION

a) Architecture de l'application

Architecture :

- *Composants (modules)*
  - *MVC*
  - *Application universelle (REACT)*
- *Infrastructure*
  - *Serveur web, navigateur, serveur base de données*
- *Etc.*

Conception :

- *Diagramme de classes*
- *Design patterns*
- *Des astuces pour gérer des particularités de votre modèle*
  - *Exemple : Récupérer des parcours qui concernent des questionnaires modifiés*

L'architecture de l'application repose sur une structure modulaire, où chaque composant joue un rôle spécifique. Le front-end est chargé de l'interface et de la gestion des interactions avec l'utilisateur. Le back-end est lui utilisé pour gérer le traitement des requêtes utilisateurs via une logique métier définie, ainsi que les communications avec la base de données pour la persistance des données.

Le diagramme ci-dessous illustre les principaux composants de l'application et leurs interactions :

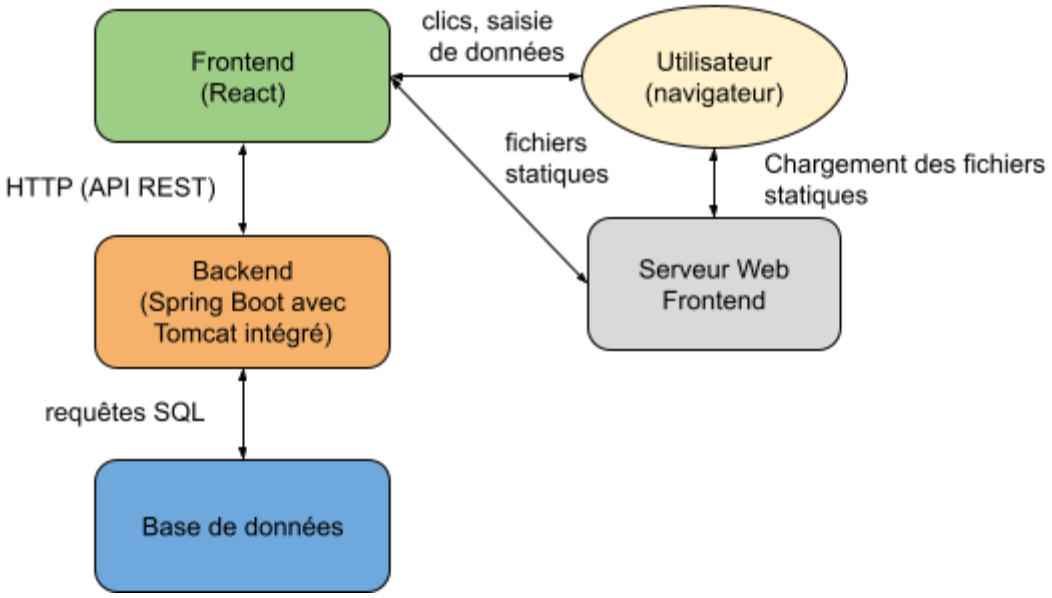

Figure 2 : Architecture globale de l'application

Le détail de l'architecture de chaque composant (backend et frontend) est donné en deuxième partie de ce rapport.

b) Diagramme de classe

Le diagramme de classe que nous avons utilisé pour le développement de notre application est donné par la figure suivante :

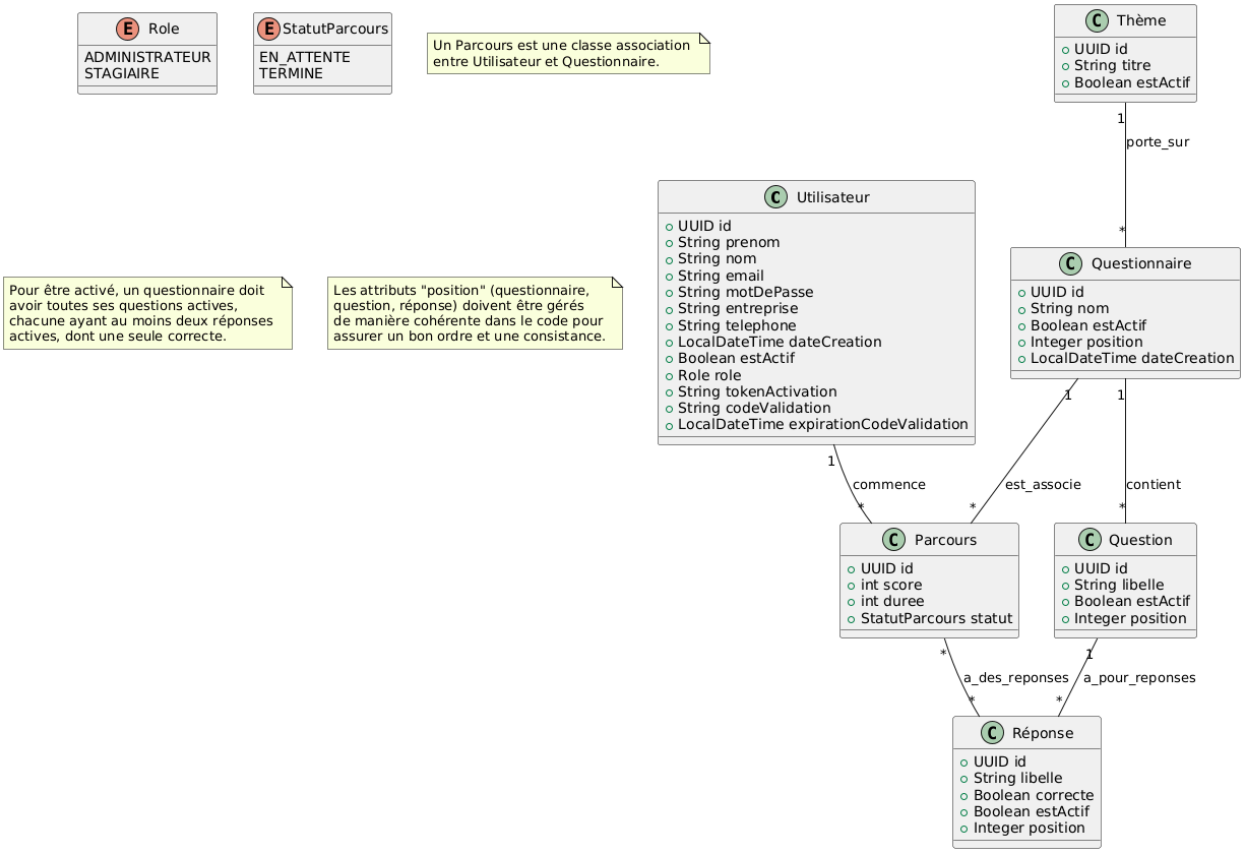

Figure 3 : Diagramme de classes

Nous avons choisi de ne pas utiliser d'iterator pour notre conception et plutôt d'implémenter la gestion des positions des différentes entités (questionnaires, questions, réponses) directement dans la couche service de notre application.

c) Modèle logique de données

La transformation du diagramme de classe en modèle logique de données est géré automatiquement par Hibernate. Le modèle résultant est le suivant :

- **Utilisateur**(#id : UUID, prenom : String, nom : String, email : String, motDePasse : String, entreprise : String, telephone : String, dateCreation : LocalDateTime, estActif : Boolean, role : {ADMIN, STAGIAIRE}, tokenActivation : String,

codeValidation : String, expirationCodeValidation : LocalDateTime)

- **Thème**(#id : UUID, titre : String, estActif : Boolean)
- **Questionnaire**(#id : UUID, nom : String, estActif : Boolean, position : Integer, dateCreation : LocalDateTime, theme_id => Thème(id))
- **Question**(#id : UUID, libelle : String, estActif : Boolean, position : Integer, questionnaire_id => Questionnaire(id))
- **Réponse**(#id : UUID, libelle : String, correcte : Boolean, estActif : Boolean, position : Integer, question_id => Question(id))
- **Parcours**(#id : UUID, score : int, duree : int, statut : {ASSIGNÉ, RÉALISÉ}, #utilisateur_id => Utilisateur(id), #questionnaire_id => Questionnaire(id))

d) Particularités de notre modèle

Gestion des utilisateurs inactifs et suppression automatique

Lorsqu'un utilisateur est désactivé, une date de désactivation (deactivation_date) est renseignée dans la base de données. Si la date de désactivation dépasse un mois sans qu'une réactivation ait été demandée, ces utilisateurs deviennent éligibles pour une suppression.

Un processus permettant d'identifier les utilisateurs inactifs depuis plus d'un mois et de les supprimer est exécuté à chaque démarrage de l'application. En effet, notre application n'étant pas en production, il ne nous était pas possible de créer une tâche programmée afin de gérer automatiquement cette suppression chaque jour par exemple.

Versionnage

Lorsqu'un admin essaie de modifier un quiz (que ce soit le quiz lui même, ses questions ou ses réponses), alors qu'un stagiaire est en train de le compléter, l'application empêche cette modification, avertit l'admin de l'existence de ce stagiaire, et elle lui invite à créer une nouvelle version du quiz afin qu'il puisse y apporter les modifications nécessaires sans interférer avec l'expérience du stagiaire.

Pour détecter l'existence d'un stagiaire en cours d'utilisation d'un quiz, l'application

enregistre chaque démarrage et terminaison d'une session de quiz à l'aide du composant `QuizSessionManager.`

2. DÉVELOPPEMENT DU PROJET

- Backend
  - Spring, Spring boot
  - Persistance des données
  - APIs
- Front-end
  - React
  - Tailwindcss
- Sécurisation
- Tests

a) Backend

Structure du projet

Afin de faciliter la collaboration au sein de notre groupe, nous avons opté pour un dépôt Git unique pour héberger à la fois le frontend et le backend. Cette décision s'est avérée pertinente dans notre contexte, où les membres de l'équipe n'ont pas eu à jongler simultanément entre les deux parties du projet (chacun s'est vu assigné soit des fonctionnalités du backend, soit du frontend). Ainsi, les développeurs backend, principalement sous IntelliJ, et les développeurs frontend, utilisant principalement VSCode, avons pu travailler de manière fluide au sein d'un même espace de travail versionné.

Pour l'implémentation de notre API, nous avons structuré notre projet Spring Boot en suivant la structure présentée par l'UV AI13 pour la séparation des responsabilités :

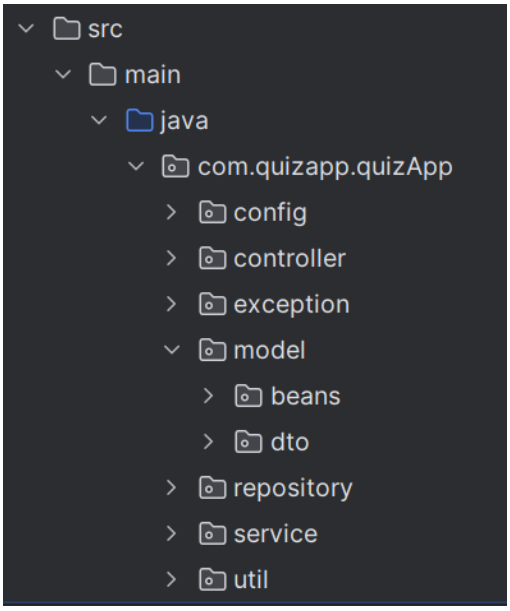

Figure 4 : Arborescence de l'API Spring Boot développée

Cette structure, largement adoptée dans le développement d'applications Spring, repose sur une séparation nette des responsabilités entre les différents composants de l'application :

- config : Ce dossier centralise tous les paramètres de configuration de l'application.
- **controller :** Les contrôleurs sont responsables de la gestion des requêtes HTTP. Ils reçoivent les requêtes, exécutent la logique métier appropriée et renvoient les réponses au client.
- exception : Ce dossier regroupe les exceptions personnalisées de l'application, permettant de gérer les erreurs de manière spécifique et de fournir des messages d'erreur clairs à l'utilisateur.
- **model :**
    - beans : Ce dossier contient les entités JPA qui représentent les objets métier persistés en base de données. Ces entités sont annotées avec les annotations JPA pour définir leur mapping avec les tables de la base de données.
    - **dto :** Ce dossier contient les objets de transfert de données (Data Transfer Objects). Les DTO sont utilisés pour transporter des données entre les

différentes couches de l'application. Ils permettent de découpler les structures de données des entités JPA et d'optimiser et sécuriser les échanges de données.

- **repository :** Les interfaces de repository définissent les opérations CRUD (Créer, Lire, Mettre à jour, Supprimer) sur les entités. Spring Data JPA permet de générer automatiquement l'implémentation de ces interfaces.
- **service :** Les services encapsulent la logique métier de l'application. Ils utilisent les repositories pour interagir avec la base de données et peuvent appeler d'autres services.
- util : Ce dossier contient des classes utilitaires utilisées dans plusieurs parties de l'application.

Implémentation

L'implémentation du backend repose sur le framework Spring Boot pour créer une API REST. La gestion des données est assurée par JPA/Hibernate qui permet de manipuler les entités de notre application et les relations entre eux (OneToMany, ManyToOne, etc).

Quiz: Le service quiz permet de créer un quiz avec ses questions et réponses en utilisant une seule requête. Il gère également la modification, l'activation/désactivation, la suppression d'un quiz et l'affichage des quiz. La modification d'une question ou d'une réponse d'un quiz est gérée par les services dédiés aux entités Question et Answer respectivement, donc le service quiz ne gère pas une requête update qui contient une nouvelle liste de questions ou de réponses. L'admin doit envoyer une requête au service concerné pour faire cette modification. La suppression d'un quiz engendre la suppression de ses questions et les réponses de ces derniers.

Question: Le service question permet de créer/modifier/supprimer une question pour un quiz précis. La requête de suppression d'une question engendre la suppression de ses propres réponses.

Answer : Le service Answer permet de créer/modifier/supprimer une réponse pour une question précise.

L'ordre des questions dans un quiz et des réponses dans une question est respecté grâce

à la persistance Jakarta. En effet, Jakarta ordonne les entités en fonction de l'attribut 'position'. Ainsi, pour permuter deux entités, il suffit de modifier la valeur de leur attribut 'position' pour que Jakarta mette automatiquement à jour l'ordre dans la liste.

Record : Le service Record de l'application gère la création, la mise à jour, et la suppression des enregistrements liés aux quiz, comme les résultats ou les statistiques des utilisateurs. Il assure la persistance et la récupération efficace des données grâce à JPA, tout en validant les entrées via Jakarta Validation. Ce service facilite également le suivi des performances des utilisateurs en centralisant les opérations sur les enregistrements associés.

User : Le service User gère les opérations liées aux utilisateurs de l'application, comme l'inscription, la mise à jour des profils, et la gestion des rôles ou permissions. Il utilise JPA pour la persistance des données et Jakarta Validation pour garantir l'intégrité des entrées. Ce service permet également l'authentification et l'autorisation des utilisateurs, assurant ainsi la sécurité et le contrôle d'accès.

Sécurité

Pour assurer la sécurité de notre application, nous avons implémenté plusieurs mécanismes en utilisant Spring Security, avec une configuration adaptée à nos besoins spécifiques. Notre objectif principal était de gérer efficacement l'authentification, l'autorisation et la protection des ressources sensibles de l'application.

Les dépendances essentielles que nous avons ajouté au fichier pom.xml incluent :

- spring-boot-starter-security : pour activer le module Spring Security.
- JWT (JSON Web Token) : pour gérer les processus d'authentification et de gestion des sessions sans état (stateless).

Ces dépendances nous ont permis d'accéder aux classes et outils nécessaires pour la gestion des utilisateurs, des rôles et des tokens d'authentification.

Les principaux mécanismes de sécurité que nous avons implémenté pour notre application sont les suivants :

1) Configuration principale : SecurityConfig

La classe SecurityConfig centralise les règles de sécurité et configure les autorisations pour chaque endpoint de l'application.

Nous avons mis en place des règles précises d'autorisation pour chaque ressource :

- Les routes publiques (comme l'authentification et la création de compte) sont accessibles sans authentification.
- Les routes protégées (comme la gestion des utilisateurs, des thèmes ou des questionnaires) nécessitent des rôles spécifiques (ADMIN ou TRAINEE).

2) Gestion des mots de passe : PasswordEncoderConfig

Nous utilisons un encodeur de mot de passe basé sur BCrypt, configuré dans la classe PasswordEncoderConfig. Cette approche garantit que les mots de passe sont stockés sous forme hachée dans la base de données, permettant un accès sécurisé aux données sensibles.

3) Filtre d'authentification JWT : JwtAuthenticationFilter

L'authentification repose sur un JwtAuthenticationFilter, qui intercepte les requêtes entrantes pour vérifier le JWT. Ce filtre extrait les informations de l'utilisateur, comme son rôle, depuis le token et les transmet au contexte de sécurité de Spring (SecurityContextHolder). Cela permet de valider les permissions de chaque utilisateur en fonction de son rôle.

4) Gestion des utilisateurs : CustomUserDetailsService

La classe CustomUserDetailsService implémente l'interface UserDetailsService pour fournir les détails de l'utilisateur lors du processus d'authentification. Les rôles des utilisateurs sont extraits de la base de données et préfixés avec ROLE_ pour respecter les conventions de Spring Security. L'utilisateur est ensuite encapsulé dans un objet de type UserDetails, incluant ses informations de connexion et son rôle.

L'authentification repose sur le mécanisme suivant :

- Lorsqu'un utilisateur se connecte, un JWT est généré en utilisant la classe JwtUtil.
- Ce JWT contient l'identifiant de l'utilisateur, son rôle et une date d'expiration.
- Le token est retourné au client, qui l'utilise pour accéder aux routes protégées.
- Lors de chaque requête, le filtre JwtAuthenticationFilter vérifie la validité du token et charge les informations utilisateur dans le contexte de sécurité.

Tests

Le développement de l'API a été accompagné de tests systématiques réalisés à l'aide de Postman. Cet outil nous a permis de simplifier la vérification du bon fonctionnement des endpoints et de garantir leur conformité avec les spécifications fonctionnelles et techniques.

Deux axes principaux ont guidé notre stratégie de test :

- Tests fonctionnels :
    - Vérification du comportement des endpoints en simulant les différentes actions possibles (création, lecture, mise à jour, suppression).
    - Validation des réponses attendues (format JSON, codes HTTP) et des cas d'erreur (ex. données manquantes, permissions insuffisantes).
- Tests de sécurité :
    - Validation des restrictions d’accès basées sur les rôles (ADMIN, TRAINEE), avec une attention particulière portée aux endpoints protégés.
    - Vérification de l’intégrité et de la validité des tokens JWT dans le cadre de l’authentification.

Pour organiser les tests et simplifier leur exécution, nous avons structuré notre configuration en collections Postman par entité métier. Chaque collection regroupe les requêtes associées à une ressource de l'application, comme le montre la figure suivante :

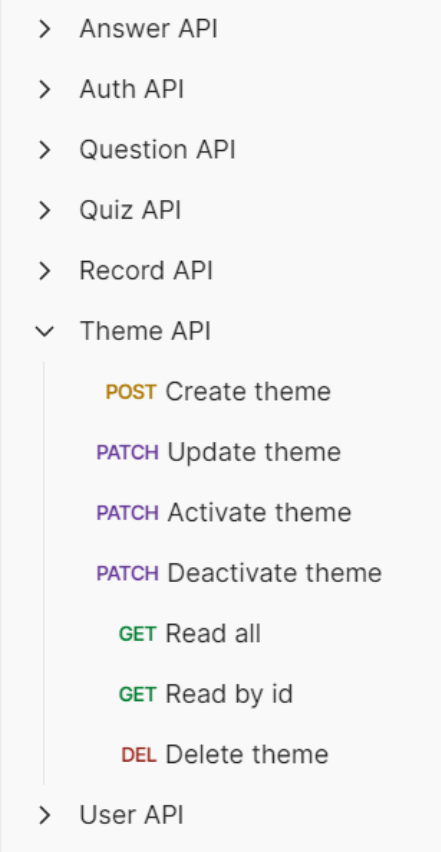

Figure 5 : Collections Postman créées pour la réalisation de tests

Cet outil nous a permis de rendre les tests plus organisés et accessibles, en réduisant le temps nécessaire pour configurer manuellement les requêtes à chaque test. De plus, le partage des collections entre les membres de l'équipe a facilité la collaboration, en particulier avec les développeurs front-end, qui ont pu directement importer les requêtes préconfigurées. Cette approche a simplifié l'intégration front-end/back-end et assuré une cohérence entre les deux couches de l'application, tout en minimisant les risques d'erreurs dues à une mauvaise interprétation des spécifications de l'API.

b) Frontend

React.js

Le frontend a été développé grâce à la bibliothèque open source React.js. Très appréciée par les entreprises, cette librairie nous a permis de développer l'interface utilisateur de l'application. En effet, React est un outil puissant qui présente de multiples avantages que nous avons su exploiter et que nous détaillerons dans la suite.

1) Approche SPA

Premièrement, React opère selon une approche SPA (Single Page Application) : les pages sont dynamiques et la mise à jour des données est gérée côté client. Cela diminue, voire élimine, les rafraîchissement fréquents des pages, et améliore donc grandement l'expérience utilisateur. Cependant, étant donné que le site nécessite de naviguer entre les pages, nous avons fait appel à une bibliothèque React chargée du routage : ***react-router-dom***. Celle-ci a permis de créer des routes dynamiques et de passer d'une page à une autre sans recharger l'intégralité de l'application.

2) Approche modulaire

De plus, React fonctionne selon une logique de composants, chacun codant une partie de l'interface. Cette conception a été conçue pour optimiser le développement de code et alléger le projet en réduisant les répétitions (en faveur de leur réutilisation). En effet, une application React est constituée de plusieurs composants isolés, qui sont assemblés pour former des "pages". Ces maxi-composants seront par la suite injectés dans notre seule page html : ***index.html***, située à la racine de notre dossier /front-end, par le biais du maxi-composant racine: la fonction App().

La fonction App() assume donc un rôle important puisque c'est le point de départ dans lequel tous les autres composants sont organisés et rendus. Ainsi, c'est à cet endroit que nous avons décidé de configurer la navigation et l'agencement de l'interface utilisateur de l'application (pour centraliser).

La figure ci-dessous présente la structure des routes de notre application côté front-end.

```jsx
function App() {

  return (
    <Router>
      <Routes>
        {/* Visitor */}
        <Route path="/" element={<Navigate to="/home" />} />
        <Route path="/home" element={<VisitorHome />} />

        <Route path="/home/bythemes" element={<VisitorByTheme />} />
        <Route path="/home/byclients" element={<VisitorByTrainees />} />

        {/* Authentication */}
        <Route path="/signin" element={<Signin />} />
        <Route path="/signup" element={<Signup />} />
        <Route path="/forgotpassword" element={<ForgotPassword />} />

        {/* Trainee */}
        <Route path="/traineespace" element={<TraineeHome />} />
        <Route path="/traineespace/records" element={<TraineeRecords />} />
        <Route path="/traineespace/quiz" element={<TraineeQuiz />} />
        <Route path="/traineespace/endquiz" element={<TraineeQuizEnd />} />
        <Route path="/profile" element={<Profile />} />
        <Route path="/updateprofile" element={<ModifyProfile />} />
        <Route path="/activate" element={<WelcomeBack />} />
        <Route path="/confirmdeactivation" element={<WantToDeactivate />} />
        <Route path="/confirmlogout" element={<WantToLogOut />} />
      </Routes>
    </Router>
  );
}
export default App
```

Figure 6 : Configurations des routes dans notre application (fichier app.jsx)

3) Structure du front-end

La structure du front-end est détaillée dans la figure ci-dessous. Les codes associés aux composants sont situés dans le dossier /src.

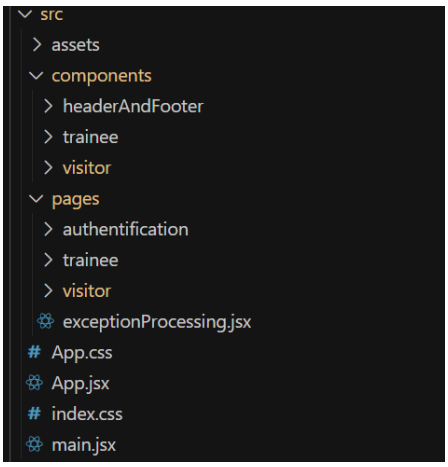

Figure 7 : Structure des fichiers gérant les composants

Les composants isolés sont donc dans /components, et vont être utilisés dans les pages qui seront injectées dans index.html. Ainsi par exemple, le composant gérant le ***footer***, est commun à toutes les pages et sera donc injecté à la fin de chaque page sous forme de balise HTML : <Footer />.

Enfin, pour décrire l'interface utilisateur dans nos composants, nous avons utilisé l'approche JSX qui utilise une syntaxe HTML dans le Javascript.

4) Aspect technique

Pour accélérer la "compilation" côté front-end et ne pas avoir à recharger constamment le serveur après chaque modification, nous avons utilisé ***Vite***, un outil permettant d'améliorer significativement les performances de développement.

L'outil permet d'installer toutes les dépendances et configurations nécessaires pour compiler rapidement un projet React. Une fois lancé, il met à jour très rapidement l'interface sur le navigateur en fonction du code entré, et permet donc de voir en temps réel le résultat du code. Il est évident que cela élimine la nécessité de relancer l'application après chaque modification, ce qui en fait un outil très prisé des développeurs. Il nous a également été très utile pour identifier et gérer les erreurs.

Enfin, pour lancer notre application, il faut se déplacer sur le dossier front-end et taper la ligne de commande : ***npm run dev***. L'application se lancera sur le port 5173 et vous pourrez naviguer aisément sur l'interface.

TailwindCSS

Pour la partie CSS, nous avons opté pour un framework open source, mis à disposition des développeurs : TailwindCSS. Ce choix a été motivé par la volonté de nous concentrer sur l'aspect fonctionnel de notre application, tout en soignant l'interface. En effet, TailwindCSS propose des classes prêtes à l'emploi et très diverses. Il ne nous reste donc plus qu'à les appeler pour choisir le design que l'on veut appliquer à notre page.

De plus, l'avantage de TailwindCSS réside dans la simplicité de son utilisation. Le framework ne nécessite que de l'installer à la racine de l'application et de configurer 3 fichiers en copiant-collant le code spécifié dans la documentation. Nous pouvons alors intégrer les classes proposées directement dans les balises, ce qui permet de développer un code propre et concis. En outre, grâce à PurgeCSS intégré, le framework élimine les classes non utilisées dans la version finale, ce qui réduit la taille des fichiers CSS.

Les fichiers associés au CSS sont :

- **tailwind.config.js** : définit la configuration de TailwindCSS dans le dossier, et sur quels fichiers s'applique-t-il. Travaillant avec React JPX, nous avons spécifié tous les fichiers js/ts et jsx/tsx.

Figure 8 : Configurations du fichier tailwind.config.js

- **postcss.config.cjs** : Ce fichier configure PostCSS pour utiliser deux plugins : *tailwindcss* et *autoprefixer*. 'tailwindcss' génère les styles basés sur les classes définies du projet tandis que autoprefixer ajoute automatiquement des préfixes pour assurer la compatibilité avec tous les navigateurs.
- **index.css** : Ce fichier sera importé par index.html, qui est le ***point d'entrée*** dans l'application React. Nous configurons donc ce fichier pour qu'il répercute les configurations (importation des couches principales de styles générés par Tailwind CSS) dans toute l'application selon:

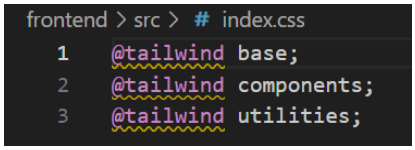

Figure 9 : Configurations du fichier index.css

Finalement, ces étapes nous ont permis d'obtenir un design soigné tout en optimisant le temps consacré à sa réalisation.

3. GESTION DE PROJET

Afin de réaliser au mieux ce projet, après avoir réalisé une analyse des besoins et avoir décrit toutes les fonctionnalités nécessaires, nous avons divisé notre groupe de travail en 2. Imene Benyagoub et Jana Rafei se sont occupées du backend en se répartissant les différentes parties de l'API à développer alors que Yousra Hassan et Jules Delannoy se sont occupés du frontend en se focalisant respectivement sur la partie visiteurs/authentification et stagiaires. Cette division nous paraît être optimale car cela permet à chacun de pouvoir travailler sur des sujets qui restent la plupart du temps disjoints. Ainsi, même si le travail des uns avait une incidence sur le travail des autres, la plupart du temps il nous était possible de travailler chacun de notre côté de façon autonome.

Pour assurer le suivi de l'avancement du projet et notre coordination, nous avons effectué des points réguliers (hebdomadaires voire bi-hebdomadaires). Cela nous a également permis de définir des priorités de développement pour maximiser notre efficacité et d'ajouter certaines méthodes ou parties de l'application qui n'avaient pas été pensées pendant l'élaboration de l'application. Au-delà de nos points, nous communiquions également à l'aide d'un logiciel conversationnel.

Pour travailler ensemble, nous avons utilisé le logiciel gitlab avec une branche principale (main) à laquelle nous ajoutons nos modifications faites sur des branches séparées ou directement via des commits.

CONCLUSION

Nous avons pu réaliser **toutes** les fonctionnalités de ce projet. Ainsi les accueils visiteurs et stagiaires, l'authentification, l'API et la gestion de la base de données sont terminés.

Il reste cependant des améliorations possibles à effectuer. La mise en commun de certains éléments React peut certainement encore être faite. Également, la mise en place de procédures pour le versionnage de l'application sur Gitlab aurait permis un meilleur suivi des modifications, cela aurait été indispensable pour un projet à long terme. Malgré cela, nous sommes fiers du travail que nous livrons.