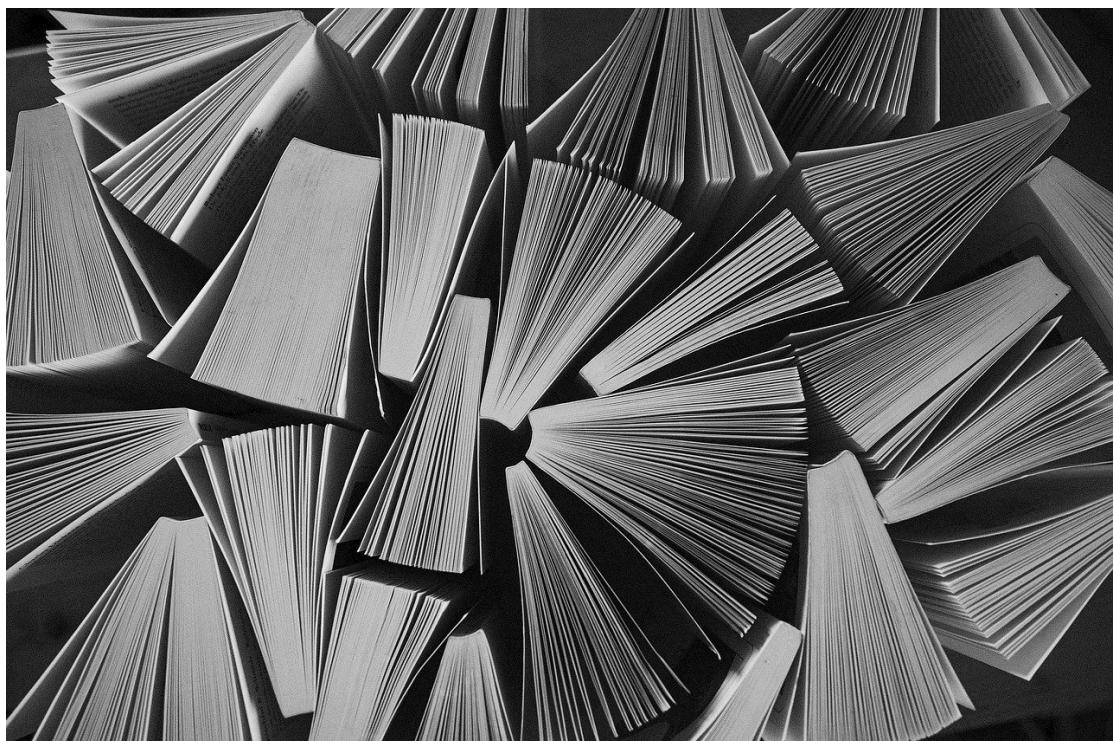


READ-IT

Dossier projet



31678 - Concepteur développeur d'applications

Mars 2024

Délhia GBELIDJI

Présentation du projet	6
The project	6
Le projet	7
Gestion du projet.....	8
L'équipe	8
Les outils.....	9
Bonnes pratiques.....	11
Spécifications fonctionnelles	14
Expression des besoins et persona.....	14
Construction de l'identité graphique.....	14
Les couleurs	15
Le logo	15
Maquettes	17
Spécifications techniques	22
Frontend	22
Pourquoi Next.js ?	22
Structure du projet	23
Librairie UI : Material UI	24
Gestion des formulaires	26
Sécurité.....	29
Tests.....	29
Backend	30
Pourquoi NestJs ?	30
Structure du projet	30
L'ORM Prisma	31
Base de données	34
Conception de la BDD et diagramme UML.....	34
Pourquoi SQLite ?	36
Développement de l'application.....	37
L'authentification côté back.....	37
Présentation d'une fonctionnalité front : la page Account.....	43
Présentation d'un test	47
Préparation du déploiement	49
Déploiement du front.....	49
Déploiement du back	50
Gérer la connexion front/back	52
Et la suite ?	53

Conclusion	54
Bibliographie	55

Présentation du projet

The project

READ-IT is the ideal fusion of technology and the publishing world. It introduces a novel approach to how we think about publishing and producing books.

This concept originated when I embarked on a career shift. While earning my bachelor's degree in modern literature, I was keen not to let my acquired knowledge go to waste. I also found several challenges within the book publishing process: it's lengthy, exclusive, and overly complex without the right connections. Whether one is an editor, proofreader, or author, breaking into the book industry is notably arduous, demanding considerable time and energy. Motivated by these challenges, I was driven to create a platform that streamlines the book publishing process.

READ-IT aims to provide an application where all participants in the publishing process can communicate and collaborate effortlessly, empowering authors to shape their books as they envision. Moreover, it democratizes book publishing, allowing anyone to publish a book, even without industry contacts. The platform eases every stage, from the first manuscript to the final publication as an e-book.

However, the project presented in this document is a Minimum Viable Product (MVP) of that platform. In collaboration with Karim FRABOULET, we developed an application that allows users to register and start a publishing project. Within this project, they can upload their manuscript, which becomes accessible to any logged-in user. Readers can offer suggestions or comments on the manuscript, providing valuable feedback for the author to refine their work.

We used Next.js for the frontend part and Nest.js, Prisma and SQLite for the backend part and the database management.

Le projet

READ-IT représente la fusion idéale entre la technologie et le monde de l'édition. Il introduit une nouvelle approche de la façon dont nous pensons l'édition et la production de livres.

Ce concept est né lorsque j'ai entrepris un changement de carrière. Je tenais à mettre à profit les connaissances acquises lors de mes études en littérature moderne. Ce premier domaine d'étude supérieur m'a permis d'identifier plusieurs défis dans le processus de publication d'un livre : il est long, exclusif et trop complexe sans un bon réseau professionnel voir mondain préétablit. Que l'on soit éditeur, correcteur ou auteur, se lancer dans l'industrie du livre est particulièrement ardu, exigeant et demande beaucoup de temps et d'énergie. Motivé par ces défis, j'ai été amené à créer une plateforme qui optimise le processus de publication de livres.

READ-IT vise à fournir une application où tous les participants au processus de publication peuvent communiquer et collaborer sans effort, permettant ainsi aux auteurs de façonnez leurs livres comme ils l'imaginent. De plus, il démocratise l'édition de livres, permettant à quiconque de publier un livre, même sans contacts avec l'industrie. La plateforme facilite chaque étape, du manuscrit initial à la publication finale sous forme de livre électronique.

Cependant, le projet présenté dans ce document est un produit minimum viable (MVP) de cette plateforme. En collaboration avec Karim FRABOULET, nous avons développé une application permettant aux utilisateurs de s'inscrire et de lancer un projet d'édition. Au sein de ce projet, ils peuvent télécharger leur manuscrit, qui devient accessible à tout utilisateur connecté. Les lecteurs peuvent proposer des suggestions ou des commentaires sur le manuscrit, fournissant ainsi de précieux commentaires à l'auteur pour affiner son travail.

Nous avons utilisé Next.js pour la partie frontend et Nest.js, Prisma et SQLite pour la partie backend et la gestion de la base de données.

Gestion du projet

L'équipe



Karim FRABOULET

Apprenti chez SNCF Connect

<https://github.com/Karimbappe>

“Tech et lecture, c'est mon truc. Je suis passionné par les deux et j'explore le monde du numérique avec curiosité. Actuellement en apprentissage chez SNCF Connect, je suis toujours à la recherche de nouveaux challenges pour développer mes compétences.”



Délhia GBELIDJI

Apprenti chez Teamoty

<https://github.com/DelhiaGbelidji>

“Tech et lecture, bah c'est aussi mon truc. J'ai d'abord fait des études dans les Lettres modernes avant de me reconvertis dans la tech. Et pour garder le meilleur des deux mondes j'ai décidé de créer une plateforme qui lit les deux. READ-IT c'est mon idée mais ce projet, on le porte à deux Karim et moi. Il nous ressemble et allie nos deux passions communes.

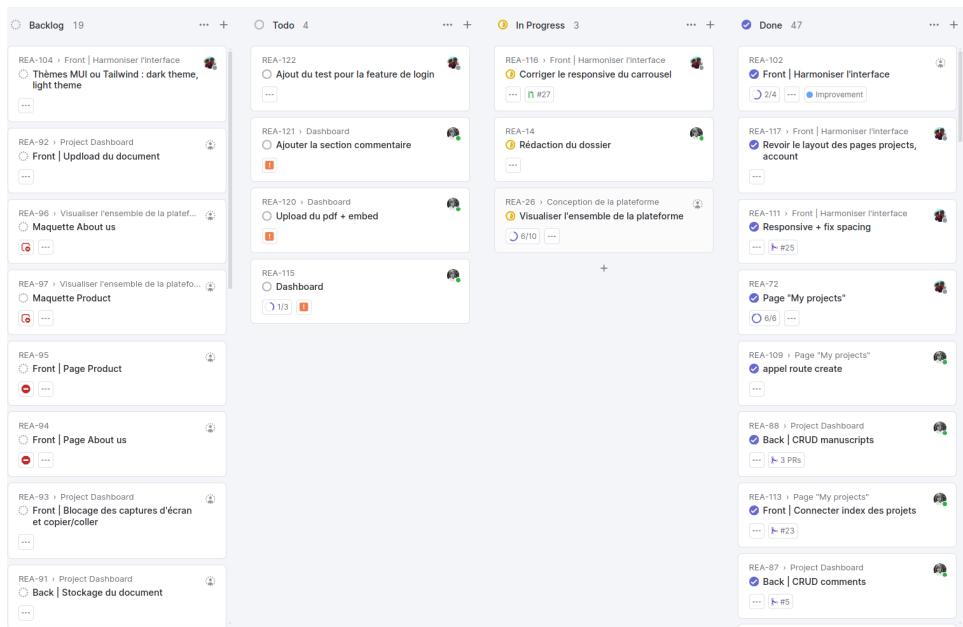
Les outils

- Github

Pour gérer efficacement la collaboration, la documentation et le versionning, nous avons choisi la plateforme Github. Cela nous a permis d'échanger sur les modifications via des pull requests. Nous avons utilisé deux repositories séparés, un pour le frontend et un pour le backend. Pour chaque nouvelle fonctionnalité, nous avons également créé une branche dédiée. Son nom devait suivre la syntaxe suivante : *feature/REA-001/nom_de_la_fonctionnalité*; ce qui permet à Linear de suivre automatiquement l'avancée de nos branches.

- Linear

Il s'agit d'un gestionnaire de tickets gratuit. Il propose des fonctionnalités collaboratives comme le tableau Kanban que nous avons mis en place en début de projet. Nous avons aussi lié nos repositories à Linear afin que la mise à jour des tickets se fasse automatiquement lorsque la pull request est publiée et lorsqu'elle est mergée.



- Notion

Nous avons utilisé Notion pour centraliser nos recherches, nos idées et les points que nous avons développé au cours du projet avec mon collaborateur et les encadrants qui nous ont accompagnés au cours de l'année. On peut y retrouver, par exemple, nos recherches préalables concernant les besoins dans le milieu de l'édition, ou des ébauches d'idées pour l'identité graphique de READ-IT. Cet outil nous a également permis de centraliser les

différentes ressources comme les liens vers la documentation technique des technologies utilisées dans le projet.



Graphic Identity Dump

Colors Ideas :

Pour une maison d'édition inclusive, la couleur la plus représentative pourrait être le violet. Le violet est souvent associé à la créativité, à l'égalité, à la diversité et à la justice. C'est une couleur qui évoque l'esprit d'inclusion, en mélangeant les qualités du rouge (passion) et du bleu (sérénité). Le violet transmet un sentiment d'ouverture, d'acceptation et de tolérance, ce qui en fait un choix approprié pour une maison d'édition axée sur l'inclusivité.

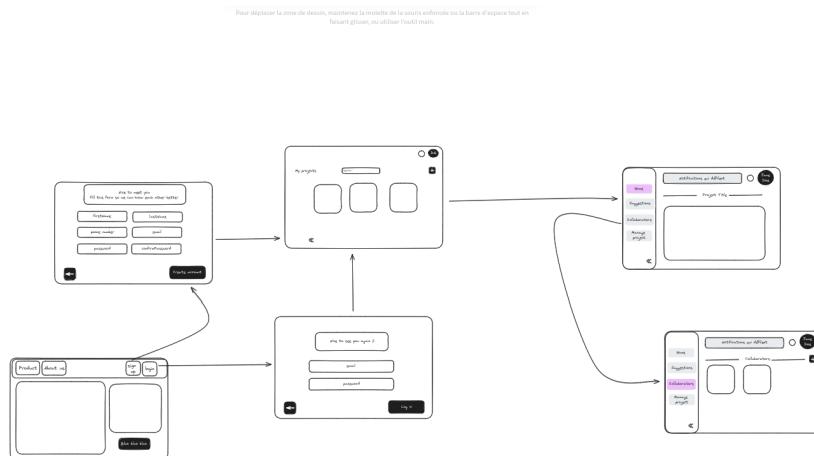


- Figma

Pour le design, notre choix s'est porté sur Figma comme outil de design collaboratif. Cet outil nous a permis de créer quelques vues de la plateforme et de maquetter des composants réutilisables comme les boutons, les logos ou les avatars.

- Excalidraw

Excalidraw est un tableau blanc collaboratif que nous avons employé pour concevoir des schémas de la plateforme, en supplément de Figma ou pour esquisser la modélisation de la base de données.



- Slack et Google Meet

Slack est l'outil de messagerie que nous utilisons afin de nous tenir au courant des avancées de chacun. Nous réalisons aussi des séances en visio sur Google Meet, pour faire du pair programming, par exemple.

Bonnes pratiques

Pour assurer le bon fonctionnement de l'équipe et de nos développements nous avons mis en place des bonnes pratiques à adopter.

Mon apprentissage chez Teamoty m'a apporté des connaissances et des compétences que j'ai pu mettre à profit lors du développement de mon projet READ IT. Comme les échanges se déroulaient principalement en ligne, par le biais de Slack ou Github, il était crucial de suivre ces principes pour rendre plus claire la compréhension des actions réalisées par les autres membres.

Tout d'abord, le bon suivi des tickets Linear en s'assurant que le développement correspond bien au scope du ticket. Comme je gérais le tableau, j'ai essayé de créer un maximum de tickets en avance et de les compléter en fonction de nos décisions ou des potentiels changements techniques. Chaque ticket est associé à un ticket "Maquette" pour s'assurer qu'il y ait soit une maquette Figma, soit un schéma Excalidraw. Il est aussi renseigné un chemin utilisateur. De plus, chaque ticket est ensuite divisé en sous-tickets : la plupart du temps, la partie backend correspondante et le découpage de la tâche côté frontend en plusieurs sous-tâches. Cela permet de réduire la taille des PR et à l'autre membre de l'équipe de pouvoir travailler en parallèle sur une autre partie de ce ticket. J'ai essayé de prendre en compte la notion de "Definition of Ready" qui indique qu'une user story doit regrouper tous les éléments nécessaires pour commencer un développement : maquette, description et estimation (même si nous n'avons pas mis en place l'estimation de nos tickets).

The screenshot shows a Jira board with two cards side-by-side. Both cards feature a 'Good to see you!' message at the top, followed by a login form with fields for 'Email' and 'Password'. Below the form is a large red button labeled 'Connexion'. The left card is titled 'Front | Login Page' and the right card is titled 'Front | SignUp'. Below the cards, a list of issues is visible under the heading 'Sub-issues (5/5)'. The issues are: REA-36 Front | Login Page, REA-37 Front | SignUp, REA-38 Back | Créer un utilisateur, REA-39 Back | Connexion utilisateur, and REA-67 Back | Sessions utilisateur : log out. To the right of the issues are several small circular icons representing different features or components.

Par ailleurs, une bonne visibilité du ticket dans le tableau permet une plus grande transparence pour tous les membres de l'équipe : s'assurer que le ticket est dans la bonne colonne et bien rempli. Cela permet aussi d'avoir une vision globale sur l'avancée du projet et des fonctionnalités qu'il reste à développer, en fonction de leur priorité, que j'ai renseigné au maximum sur les features les plus importantes du projet.

Ensuite, le bon traitement des pulls requests permet aussi une meilleure efficacité. Chaque développement est suivi par la publication de la pull request qui doit être relue et validée par l'autre membre de l'équipe. J'ai mis en place une règle Github qui empêche le merge d'une branche dans develop s'il n'y a pas (au moins) une approbation.

The screenshot shows a GitHub pull request interface for a file named 'Fix carousel #27'. At the top, it says 'Karimbappe wants to merge 1 commit into main from feature/REA-116/corriger_responsive_carousel'. Below this, there are tabs for 'Conversation' (1), 'Commits' (1), 'Checks' (0), and 'Files changed' (3). The main area shows a preview of the code changes, which include modifications to a 'book荐书' component. A comment from 'Karimbappe' is visible, adding the 'Ready to review' label 6 hours ago. Another comment from 'linear' is also present. On the right side, there are sections for 'Reviewers' (DelhaGbediji), 'Assignees' (No one—assign yourself), 'Labels' (Ready to review), 'Projects' (None yet), 'Milestone' (No milestone), and 'Development' (Successfully merging this pull request may close these issues). The 'Notifications' section indicates that the user is receiving notifications because they're watching this repository. At the bottom, there's a summary of the review status: 'Changes approved' (1 approving review), '1 approval', and 'This branch has no conflicts with the base branch'.

Nous avons aussi décidé de trois labels, fonctionnalité présente sur Github, qui nous permettent de savoir si la pull request peut être relue, si le développement est encore en cours ou si la pull request peut être relue mais pas mergée. Il s'agit des labels suivants :

Do not merge		Edit	Delete
Ready to review	⌚ 1	Edit	Delete
WIP	Work in progress	Edit	Delete

Do not merge permet d'indiquer que le développement est terminé mais que la branche ne peut pas encore être mergée.

Ready to review permet de notifier que le développement est terminé et que la pull request peut être relue.

WIP indique que le développement encore en cours ou que les retours sont en cours de traitement. La pull request n'est alors pas à relire.

Ces règles ont été synthétisée dans un document Wiki, outil proposé par Github:

Best Practice

Karim edited this page yesterday · 1 revision

[Edit](#) [New page](#)

Pages 1

- ▼ Best Practice
 - I / Rigorous Tracking of Linear Tickets:
 - II/ Association of Each Ticket with a “Design Ticket”:
 - III/ Division of Tickets into Sub-Tickets:
 - IV/ Application of the “Definition of Ready” Concept:
 - V/ Effective Management of Pull Requests (PRs):
 - VI/ Use of Specific Labels for PRs:

I / Rigorous Tracking of Linear Tickets:

- Ensure each development aligns with the ticket's scope.
- Anticipate ticket creation, completing them based on team decisions or technical changes.

II/ Association of Each Ticket with a “Design Ticket”:

- Include a Figma mockup or an Excalidraw diagram.
- Indicate a user journey for each ticket.

III/ Division of Tickets into Sub-Tickets:

- Distribute between backend part and several frontend sub-tasks.
- Reduce the size of PRs to facilitate parallel work on other aspects of the ticket.

IV/ Application of the “Definition of Ready” Concept:

- Ensure that each user story contains all necessary elements to begin development: mockup, description, and estimation (although ticket estimation was not systematically applied).
- Visibility and optimal management of tickets on the board.
- Keep tickets in the appropriate column and ensure they are well filled.
- Provide a global view of project progress and the functionalities remaining to be developed, with particular attention to priority features.

V/ Effective Management of Pull Requests (PRs):

- Follow up each development with a PR, which must be reviewed and approved by another team member before merging.
- Implement a GitHub rule preventing merging without at least one approval.

VI/ Use of Specific Labels for PRs:

- Do not merge: Indicates that development is complete but the branch cannot yet be merged.
- Ready to review: Notifies that development is finished and the PR is ready for review.
- WIP (Work In Progress): Signifies that development is still ongoing or feedback is being processed. The PR is not ready for review.

Spécifications fonctionnelles

Expression des besoins et persona

L'utilisateur de READ-IT exerce un métier de l'édition. Il peut ainsi correspondre à plusieurs personas ayant des besoins différents : l'auteur, l'éditeur et tous les autres métiers de l'édition, traducteurs, dessinateurs, correcteurs ainsi que les imprimeurs, équipes commerciales et marketing.

La plateforme utilisée par ces différents profils utilisateurs doit être collaborative, stable et sécurisée tout en répondant aux besoins spécifiques de chaque métier. Elle doit aussi être simple d'utilisation et sobre pour fluidifier l'expérience utilisateur. Le but étant d'accélérer le processus d'édition d'un livre et de laisser plus de liberté à l'auteur pour construire son livre comme il le souhaite, un système de commentaires avec validation de l'auteur à chaque suggestion ou correction d'un autre métier doit être mis en place.

Pour notre produit minimal viable (MVP), nous avons choisi les fonctionnalités suivantes :

- Page d'accueil,
- Enregistrement de l'utilisateur,
- Authentification,
- Création, édition et suppression d'un projet d'édition par l'utilisateur,
- Affichage des projets de tous les utilisateurs,
- Téléchargement d'un manuscrit au format PDF,
- Système de commentaires ouvert à tous les utilisateurs pour discuter du manuscrit,
- Page de gestion du compte permettant de modifier les informations de l'utilisateur, de changer le mot de passe et de supprimer le compte.

Construction de l'identité graphique

Les valeurs fondamentales de READ-IT sont l'inclusivité, le partage et l'ouverture au monde. La structure même de la plateforme permet à tout auteur de démarrer son projet d'édition de manière autonome tout en bénéficiant de la collaboration d'autres utilisateurs. Cette dynamique favorise l'accessibilité au secteur souvent exclusif de l'édition et contribue à enrichir la diversité sur le marché du livre.

Pour une maison d'édition qui prône l'inclusivité, le rose apparaît comme la couleur la plus représentative. Le rose, souvent associé à la compassion, à la bienveillance et à l'amour, incarne parfaitement ces valeurs. Il symbolise la tendresse et l'acceptation, mêlant

les qualités d'attention et d'ouverture qui sont au cœur de READ-IT. En véhiculant des sentiments d'empathie et de compréhension, le rose est un choix judicieux pour une maison d'édition dédiée à l'inclusivité, reflétant son engagement envers l'accueil et le soutien de la diversité des voix et des récits.

Les couleurs

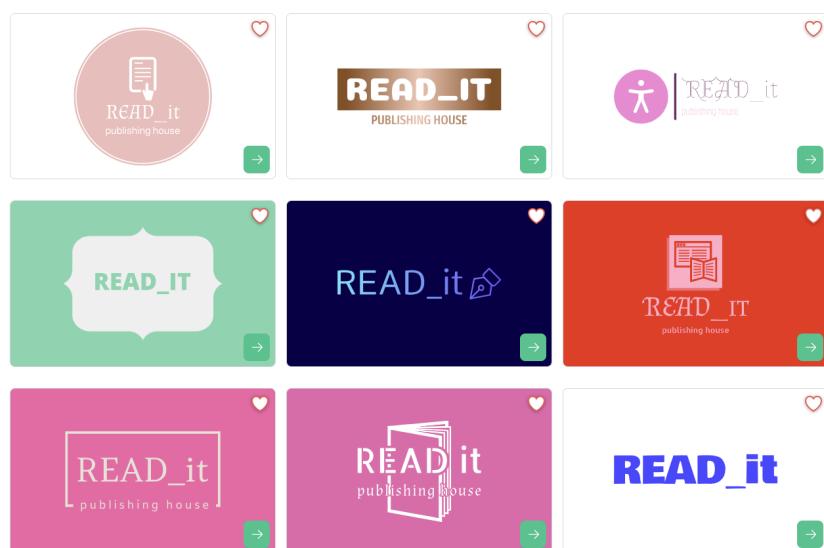
L'esthétique générale du design est épurée et minimaliste, mettant l'accent sur la facilité d'utilisation grâce à un contraste marqué qui favorise la lisibilité. L'utilisation d'une palette de couleurs principalement monochromes, rehaussée par des détails colorés, confère une allure à la fois contemporaine et élégante.



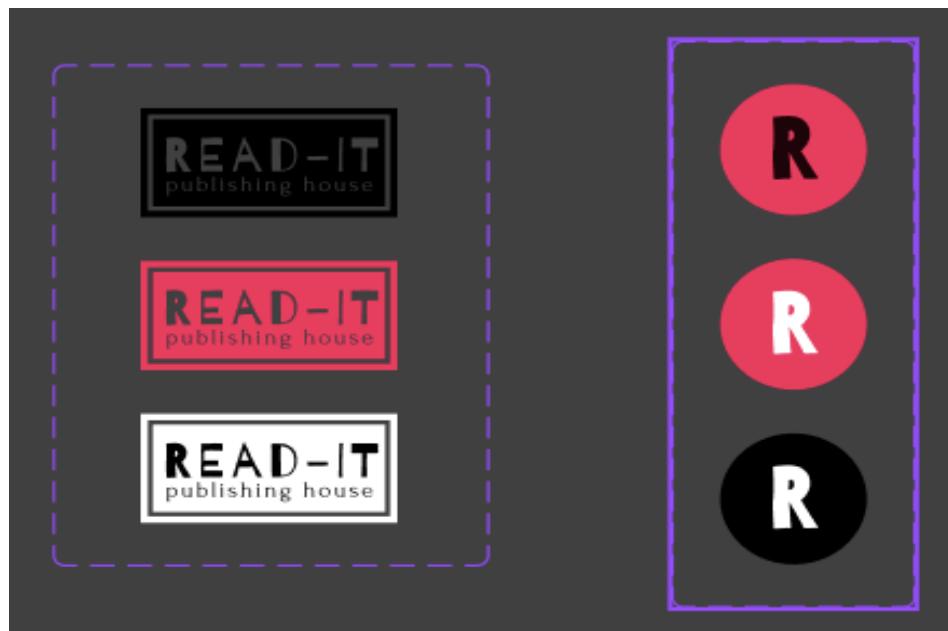
Ci-dessus, la palette que nous avons générée lors de nos recherches, nous avons finalement choisi comme couleurs principales le rose **F7195C** et le noir.

Le logo

Voici quelques inspirations pour un logo, générées sur le site logo.com :

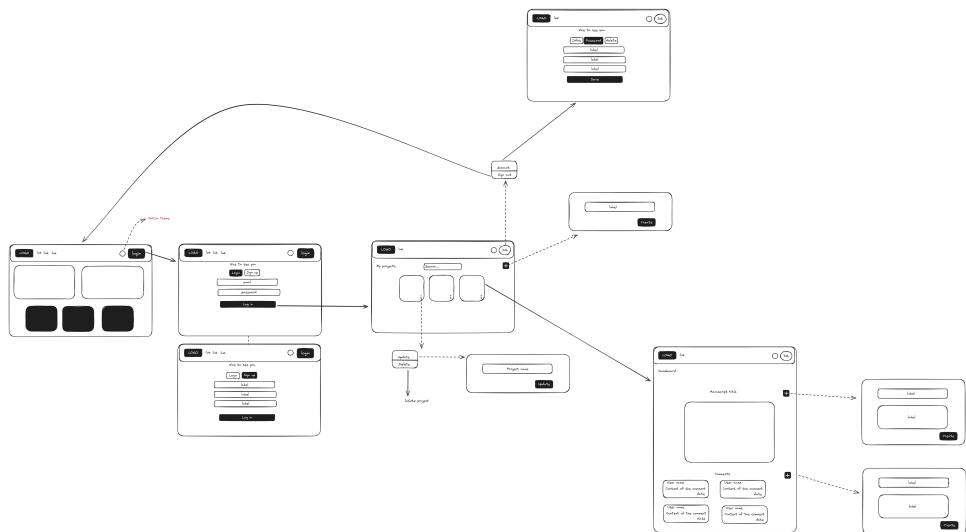


Finalement, le design du logo a été réalisé sur Figma. Ces conceptions de logos forment une identité visuelle unifiée, convenant à une variété de supports tout en conservant l'essence de READ-IT. Ils évoquent un style contemporain qui fait écho au domaine de l'écriture et de l'édition, une connexion soulignée par le nom de la maison d'édition elle-même.



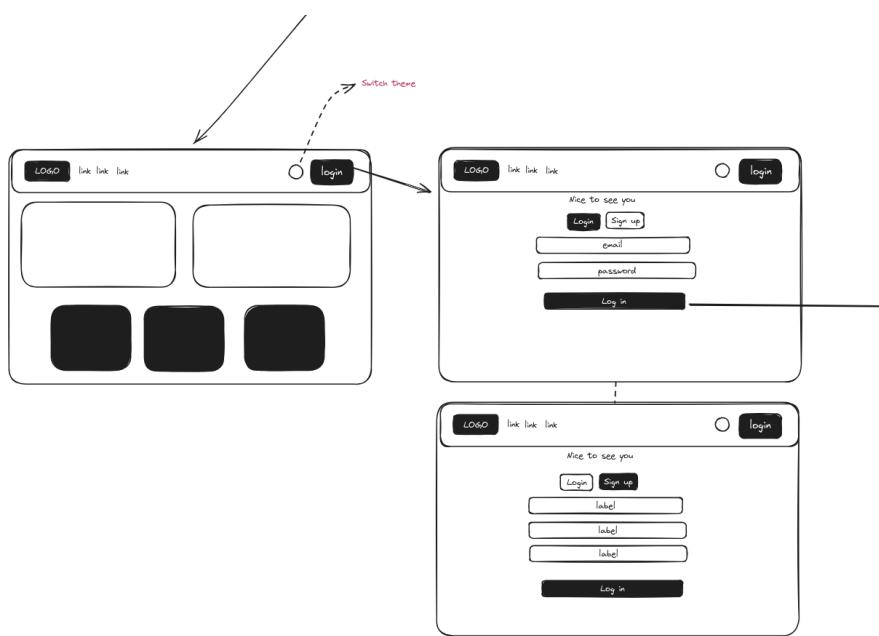
Maquettes

Comme mentionné précédemment, nous n'avons pas construit les maquettes entièrement sur Figma. En revanche, j'ai réalisé un schéma entier de la plateforme sur la plateforme Excalidraw.

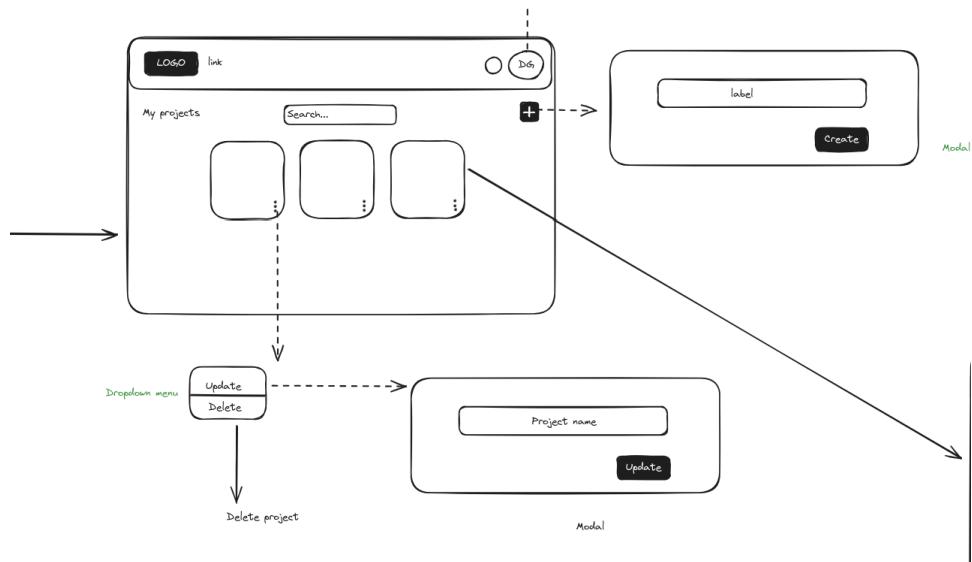


Ci-dessus, le schéma entier de la plateforme.

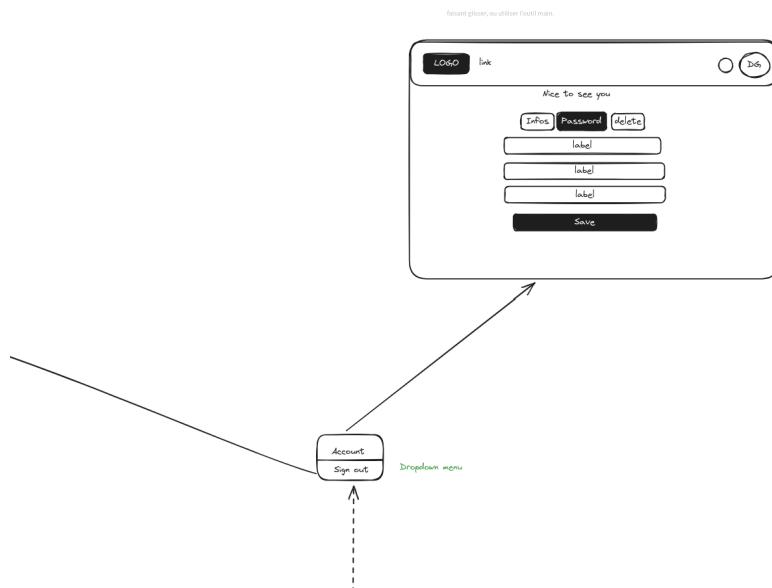
Page d'accueil, création de compte et connexion :



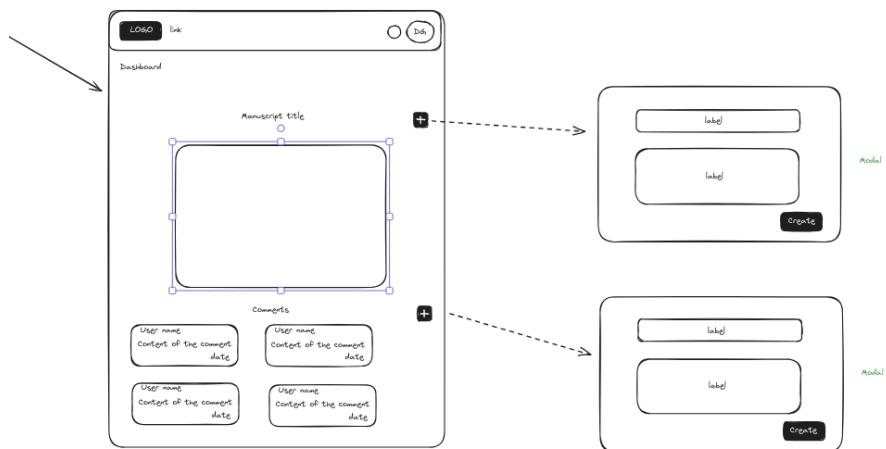
Liste des projets et modal permettant les actions du CRUD :



Page de gestion du compte :



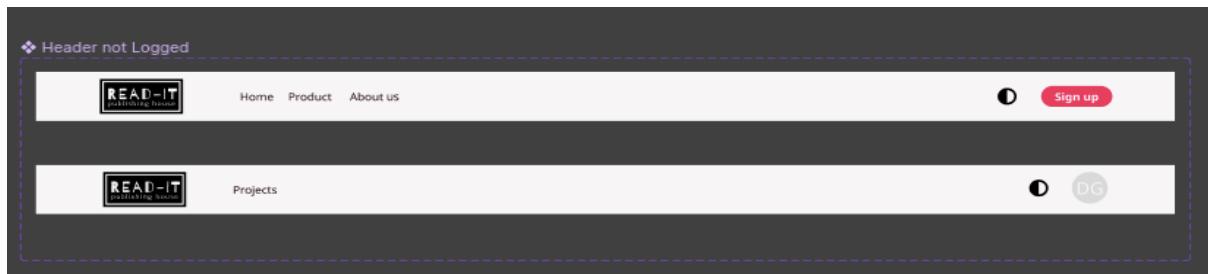
Dashboard du projet avec l'affichage du manuscrit et les commentaires :



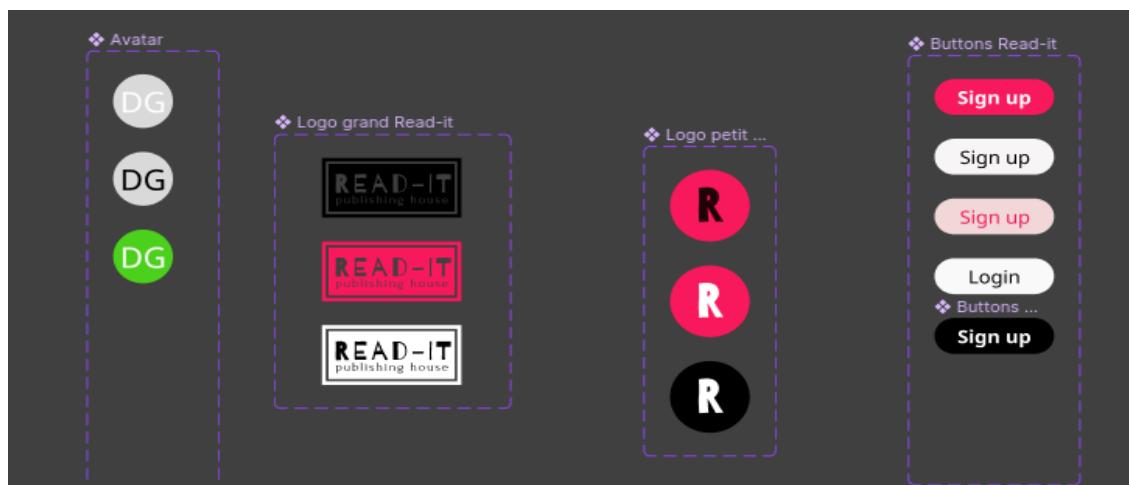
Les traits en pointillés représentent l'ouverture d'un composant de style modal, menu déroulant. Les traits pleins montrent les redirections vers des pages.

Parallèlement sur Figma, nous avons travaillé sur certaines pages comme la page de login ou sur le design des composants réutilisables à plusieurs endroits sur la plateforme.

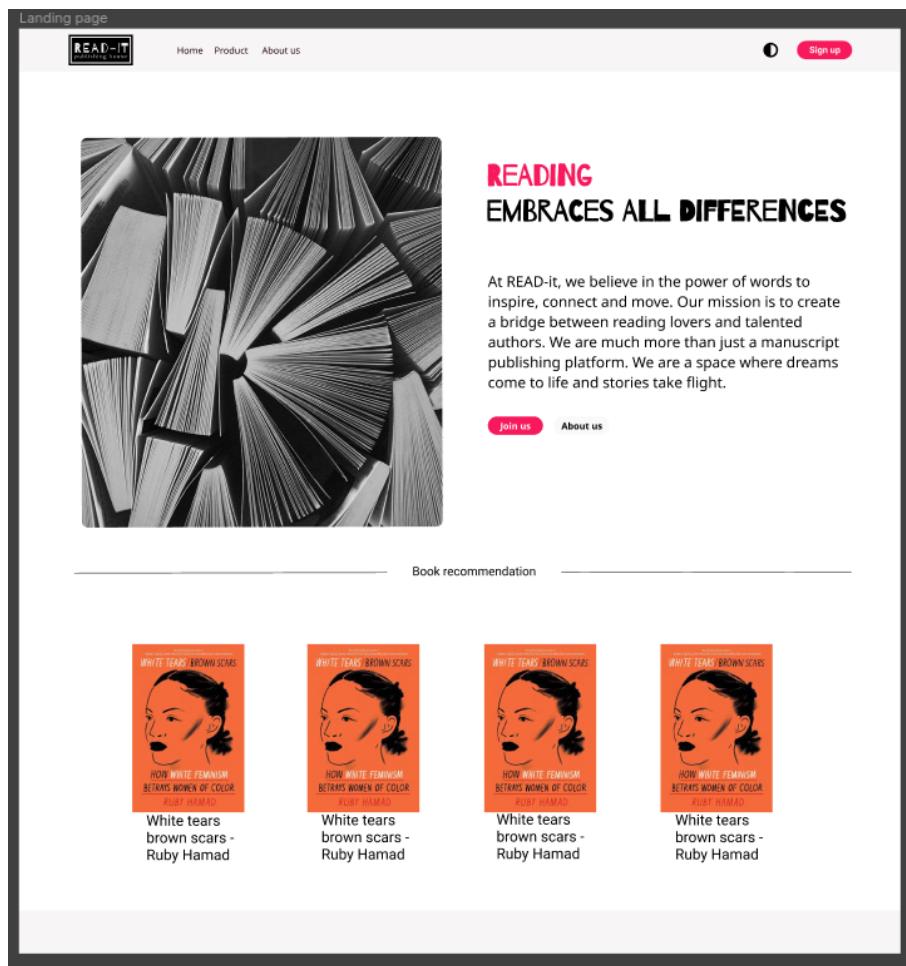
Headers :



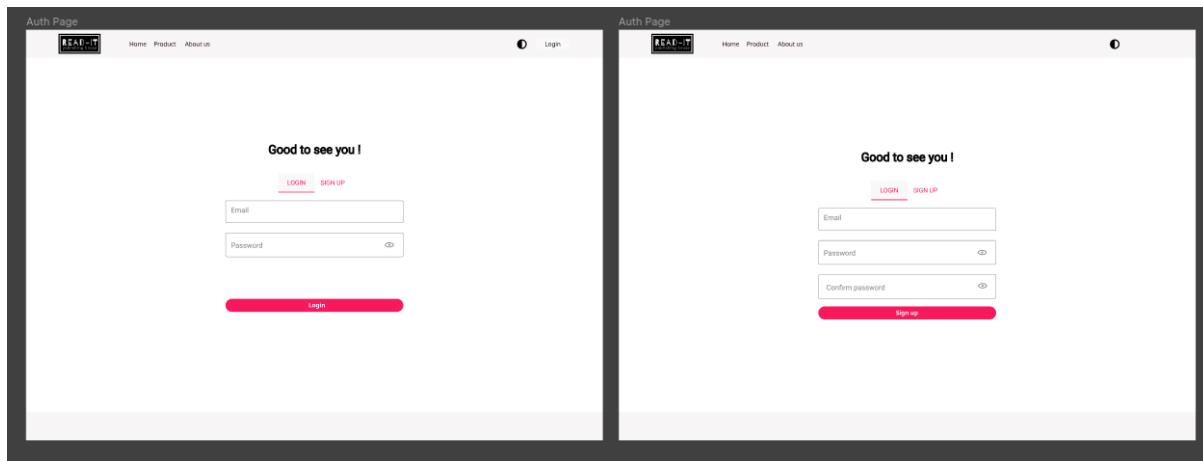
Boutons, avatar et déclinaisons du logo :



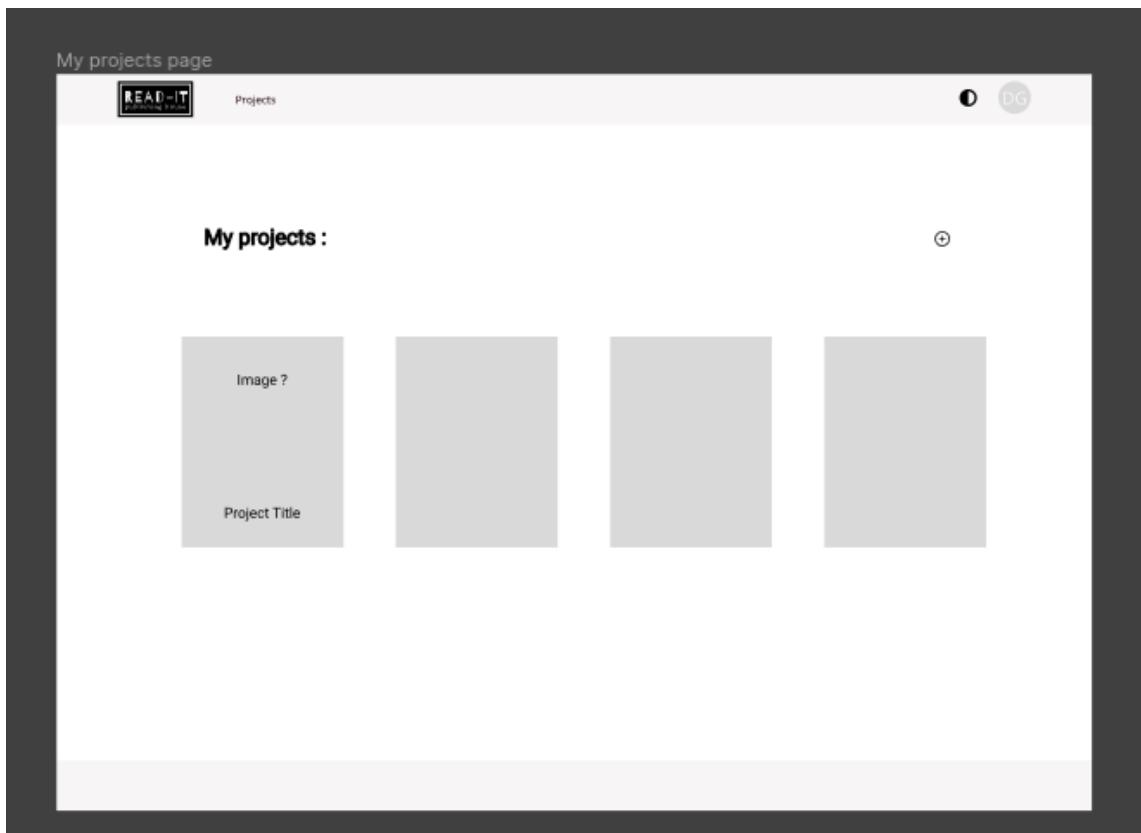
Maquette de la page d'accueil :



Authentification :



Mockup de la page projets :



Spécifications techniques

READ-IT est une plateforme développée avec le framework Next.js côté front et le framework Nestjs côté back.

Frontend

Pourquoi Next.js ?

Extrait de la documentation Next.js (<https://nextjs.org/docs>):

"Next.js is a React framework for building full-stack web applications. You use React Components to build user interfaces, and Next.js for additional features and optimizations.

Under the hood, Next.js also abstracts and automatically configures tooling needed for React, like bundling, compiling, and more. This allows you to focus on building your application instead of spending time with configuration."

Next.js a été conçu pour être flexible, offrant la possibilité d'intégrer aisément des API, des services externes et des outils de développement, rendant ainsi le processus de création d'applications web complet et souple. Grâce à l'écosystème riche de Next.js et à sa communauté active, il est également possible de trouver rapidement des solutions aux défis de développement.

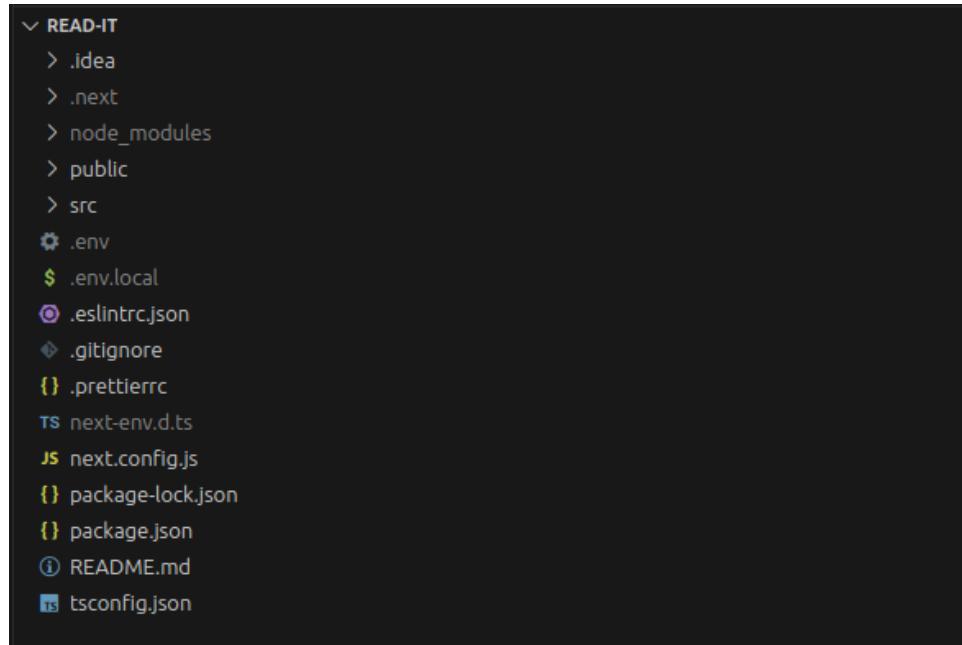
L'intégration de l'App Router dans la version 14 de Next.js présente de nombreux bénéfices pour la création de projets web modernes. La création d'applications React est simplifiée grâce à Next.js, qui propose des fonctionnalités comme le Rendering Server-Side (SSR) et la génération de sites statiques (SSG), ce qui améliore significativement les performances de l'application et l'expérience utilisateur en accélérant le chargement des pages et en optimisant le référencement (SEO). La mise en place de l'App Router représente une avancée majeure dans la gestion des routes, en proposant une méthode plus intuitive et déclarative pour définir les trajets des utilisateurs dans l'application. Cette nouvelle méthode encourage une organisation plus efficace du code, une diminution des tâches de maintenance et une expérience de développement plus fluide.

READ-IT est une plateforme vouée à grandir et à se complexifier, il me paraissait important de garantir une expérience utilisateur agréable, en préservant des temps de chargements courts, une application robuste avec une UI plaisante.

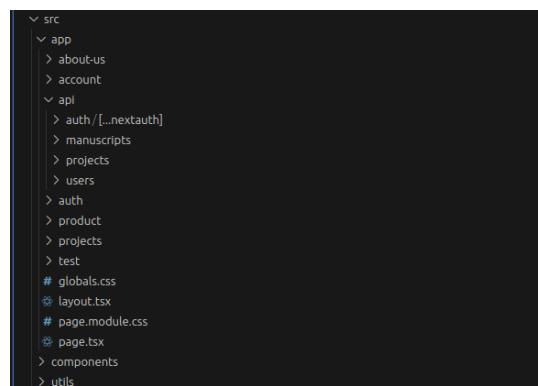
De plus, la curiosité de découvrir cette technologie qui m'était inconnue, n'ayant développé qu'avec React ou du Javascript vanilla lors de mon apprentissage, m'a poussée à choisir Next.js. Sans oublier que c'est un framework très utilisé et de plus en plus demandé par les entreprises.

Structure du projet

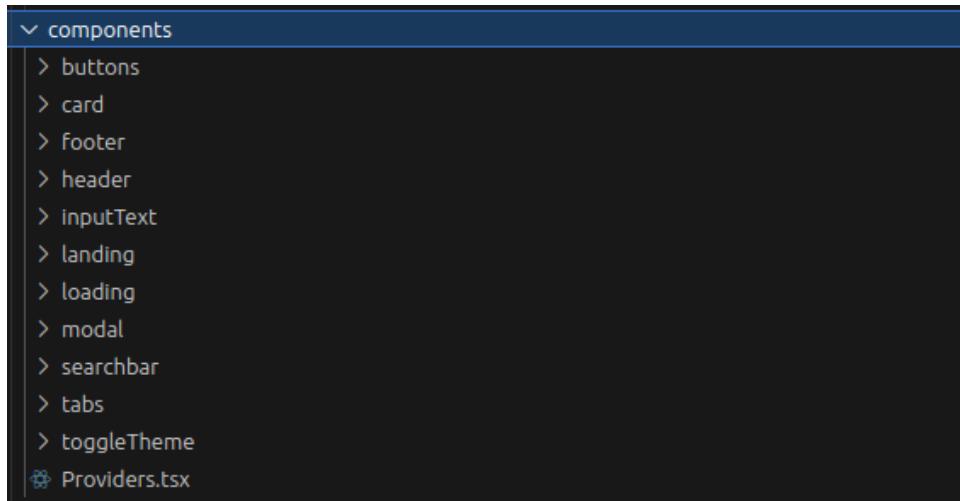
Comme précisé précédemment, nous utilisons l'App router de Next.js (v14). On retrouve au top-level les fichiers utilisés pour configurer l'application, gérer les dépendances, exécuter un middleware, intégrer des outils de surveillance ou encore définir des variables d'environnement : package.json et package-lock.json, .eslintrc.json (pour le linter), .prettierrc (pour la configuration de Prettier), le fichier nextconfig.js et les .env et .env.local pour les variables d'environnement.



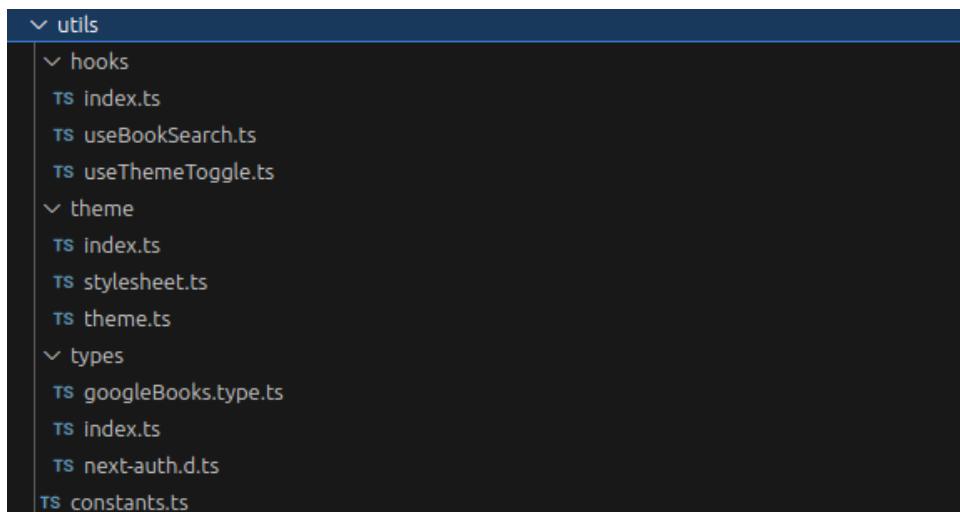
Next.js use d'un routeur basé sur un système de fichiers dans lequel les dossiers sont utilisés pour définir les routes. Chaque dossier représente un segment de route qui correspond à un segment d'URL. Pour créer des routes imbriquées, on peut créer des dossiers les uns dans les autres. C'est cette architecture que l'on va retrouver dans le dossier app au sein du fichier src. Chaque dossier ou sous-dossier contient un fichier page.tsx permettant de rendre le segment public. Avec mon collaborateur, nous avons également créé le dossier api contenant les fichiers route.ts. Un fichier de route permet de concevoir des gestionnaires de requêtes HTTP sur mesure pour une route spécifique. Nous les utilisons pour appeler nos endpoints côté back-end.



Dans le dossier components, on va trouver tous les composants réutilisables que nous avons créés comme les boutons, les modales, etc. On y trouve aussi des composants Mui dont nous avons modifié le style.



Enfin dans le dossier utils, on va retrouver les outils (custom hooks, constantes appelées dans plusieurs composants ou pages, types, etc) qui nous serviront partout dans l'application.



Librairie UI : Material UI

Extrait de la documentation Material UI (<https://mui.com/material-ui/>) :

Material UI is an open source React component library that implements Google's Material Design. It's comprehensive and can be used in production out of the box.

Cette librairie propose des composants prêts à être utilisés ainsi que des outils de personnalisation de ces derniers. Nous avons fréquemment utilisé la méthode **styled()**

fournie par MUI ou l'objet sx passé en props des composants et permettant de modifier leur style directement. Lorsqu'un composant est modifié plusieurs fois de la même façon, on en crée un nouveau. À l'inverse, lorsqu'il s'agit d'une modification ponctuelle, nous utilisons l'objet sx.

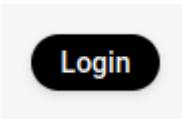
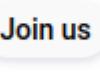
Prenons l'exemple du composant Button de Material UI, nous avons d'office créé le composant stylisé dès le lancement du projet, en sachant que c'est un élément qui allait être utilisé sur pratiquement toutes les pages, avec des variantes. Nous avons centralisé ces dernières dans le fichier Buttons.tsx situé dans le dossier /components/buttons.

```
export const DefaultButton = styled(Button)((sx) => ({
  fontSize: '14px',
  lineHeight: '16px',
  backgroundColor: COLORS.black,
  color: COLORS.white,
  borderRadius: '50px',
  boxShadow: '0px 2px 4px rgba(25, 49, 84, 0.15)',
  '&:hover:not(.Mui-disabled), &.Mui-focused': {
    backgroundColor: COLORS.pink,
    boxShadow: '0px 2px 4px rgba(25, 49, 84, 0.15)',
  },
  '&.Mui-disabled': {
    color: COLORS.disabledTextButton,
    backgroundColor: COLORS.disabledButton,
  },
  '&.MuiButtonBase-root': {
    textTransform: 'none',
  },
}))
```

```
export const ClearButton = styled(Button)((sx) => ({
  fontSize: '14px',
  lineHeight: '16px',
  backgroundColor: COLORS.clear,
  color: COLORS.black,
  borderRadius: '50px',
  boxShadow: '0px 2px 4px rgba(25, 49, 84, 0.15)',
  '&:hover:not(.Mui-disabled), &.Mui-focused': {
    backgroundColor: COLORS.pink,
    boxShadow: '0px 2px 4px rgba(25, 49, 84, 0.15)',
  },
  '&.Mui-disabled': {
    color: COLORS.disabledTextButton,
    backgroundColor: COLORS.disabledButton,
  },
  '&.MuiButtonBase-root': {
    textTransform: 'none',
  },
}))
```

```
export const TextButton = styled(Button)((sx) => ({
  '&.MuiButtonBase-root, .MuiButton-text': {
    color: COLORS.black,
  },
}))
```

Nous pouvons observer les variantes DefaultButton, ClearButton, TextButton:

		
DefaultButton	ClearButton	TextButton

Gestion des formulaires

L'association de React Hook Form et Yup pour la gestion et la validation de formulaires dans les applications React représente une solution puissante et efficace.

React Hook Form se distingue par son approche performante, réduisant les renders grâce à l'utilisation de hooks non contrôlés. Cette bibliothèque simplifie également le développement en minimisant la quantité de code nécessaire pour créer et gérer des formulaires, même ceux de complexité élevée. Sa capacité à s'intégrer de manière fluide avec des bibliothèques de validation comme Yup enrichit davantage son potentiel.

On utilise le hook `useForm()` qui va permettre de récupérer les éléments suivants :

- L'objet **control** contient les méthodes pour enregistrer les composants dans React Hook Form;
- **handleSubmit()** est une fonction qui gère la soumission du formulaire. Elle prend en charge l'exécution de la validation de tous les champs enregistrés via `register()`. Si la validation réussit, `handleSubmit()` passe les données du formulaire à une fonction callback qui a été définie, permettant ainsi de traiter les données soumises. Cette séparation claire entre la validation et le traitement des données soumises assure une logique d'application bien organisée et facile à maintenir ;
- La fonction **reset ()** est utilisée pour réinitialiser les champs du formulaire à leurs valeurs initiales ou à des valeurs spécifiques. Cela est particulièrement utile dans les scénarios post-soumission, où l'on souhaite effacer le formulaire ou le remettre à son état initial après l'envoi des données. `reset()` contribue à une meilleure expérience utilisateur en fournissant un moyen simple de nettoyer le formulaire après l'utilisation ;
- **formState** est un objet qui contient des informations sur l'état du formulaire, y compris les erreurs. En utilisant `formState`, on peut accéder facilement aux erreurs de validation spécifiques à chaque champ. Cela simplifie l'affichage des messages d'erreur à l'utilisateur, améliorant ainsi l'expérience utilisateur en fournissant des retours immédiats et pertinents sur la saisie des données.

```
//Form handler
const {
  handleSubmit,
  control,
  reset,
  formState: {errors},
} = useForm<Type_SignupData>({
  defaultValues: {
    firstname: '',
    lastname: '',
    email: '',
    password: '',
    confirm_password: '',
  },
  resolver: yupResolver(schema_SignUp),
})
```

Yup, de son côté, propose une manière déclarative et intuitive de définir des schémas de validation. Avec des fonctionnalités permettant des validations personnalisées et asynchrones, ainsi que la possibilité de définir des messages d'erreur spécifiques, Yup facilite la création d'expériences utilisateur cohérentes et instructives. Les développeurs peuvent ainsi établir des règles de validation complexes de manière claire et concise.

```
export const Schema_SignUp = Yup.object().shape({
  email: Yup.string().email('Invalid email').required('Email is required'),
  firstname: Yup.string().required('Firstname is required'),
  lastname: Yup.string().required('Lastname is required'),
  password: Yup.string()
    .matches(passwordRules, {message: 'Please create a stronger password'})
    .required('Required'),
  confirm_password: Yup.string()
    .oneOf([Yup.ref('password')], 'Passwords must match')
    .required('Required'),
})
```

Par ailleurs, le composant **Controller** de React Hook Form offre une manière efficace d'intégrer des champs de formulaires personnalisés ou de bibliothèques tierces, tels que ceux de Material UI, avec la gestion de formulaires. Utiliser Controller permet de contrôler entièrement les composants de saisie, tels que les TextField de Mui, en les enveloppant de manière qu'ils soient pleinement compatibles avec la logique de React Hook Form. Ceci est particulièrement utile car certains composants de Mui gèrent leur propre état interne et ne s'intègrent pas directement avec les méthodes standard de React Hook Form, comme register. L'utilisation de Controller assure que tous les changements de valeur sont correctement capturés et que les règles de validation sont appliquées, tout en conservant les fonctionnalités et l'apparence des composants Mui.

```
<form onSubmit={handleSubmit(onSubmit)}>
  <Grid container spacing={2}>
    <Grid item xs={12} sm={6}>
      <Controller
        name='firstname'
        control={control}
        render={({field}) => (
          <Styled_TextField
            {...field}
            label='Firstname'
            variant='outlined'
            fullWidth
            error={!errors.firstname}
            helperText={errors.firstname?.message}
          />
        )}
      />
    </Grid>
```

La combinaison de React Hook Form et Yup offre ainsi une synergie remarquable pour la gestion des formulaires et la validation côté client. Cette alliance permet non seulement d'accroître l'efficacité du développement grâce à des formulaires plus réactifs et moins verbeux, mais aussi d'améliorer l'expérience utilisateur grâce à des retours immédiats et pertinents sur la validation des données.

Sécurité

Pour le côté frontend de notre application, nous avons renforcé les mesures de sécurité, notamment dans le processus d'authentification, en utilisant NextAuth combiné à une expression régulière (regex) spécifique pour la validation des mots de passe dans les schémas de validation des formulaires de création de compte, de connexion et du changement de mot de passe dans la page de gestion du compte. Cette regex garantit que tous les mots de passe créés doivent comprendre au minimum 8 caractères, incluant au moins une majuscule, un caractère spécial, et un chiffre.

Tout comme pour le côté back-end, en vue du déploiement de notre interface utilisateur est sécurisé par l'utilisation d'un fichier .env. Ce fichier contient des variables d'environnement essentielles au fonctionnement de l'application mais est exclu du versionning pour empêcher la divulgation d'informations sensibles. Cette stratégie de gestion des configurations sensibles assure une couche supplémentaire de sécurité à notre application.

Tests

Pour la couverture de tests, nous utilisons la librairie de tests Jest qui nous a permis de réaliser des tests unitaires sur une de nos fonctionnalités. Ce test sera présenté plus tard dans le dossier.

Backend

Pourquoi NestJs ?

Extrait de la documentation de NestJs (<https://docs.nestjs.com/>):

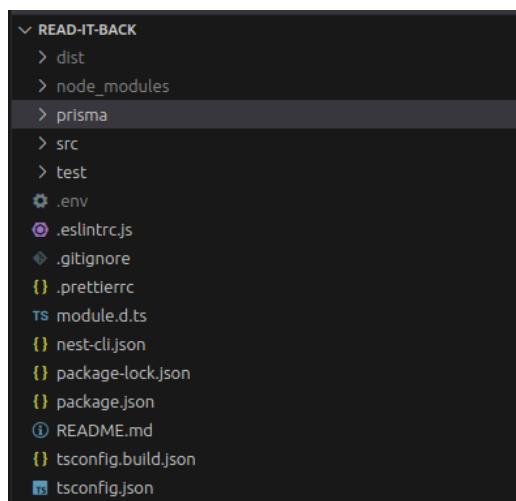
Nest provides a level of abstraction above these common Node.js frameworks (Express/Fastify), but also exposes their APIs directly to the developer. This gives developers the freedom to use the myriad of third-party modules which are available for the underlying platform.

NestJS se distingue comme un framework backend progressif pour Node.js, particulièrement apprécié pour son architecture inspirée d'Angular, offrant une structure modulaire et extensible qui encourage les bonnes pratiques de programmation. Cette structure est particulièrement avantageuse pour les grandes applications et les équipes, car elle permet une organisation claire et maintenable du code.

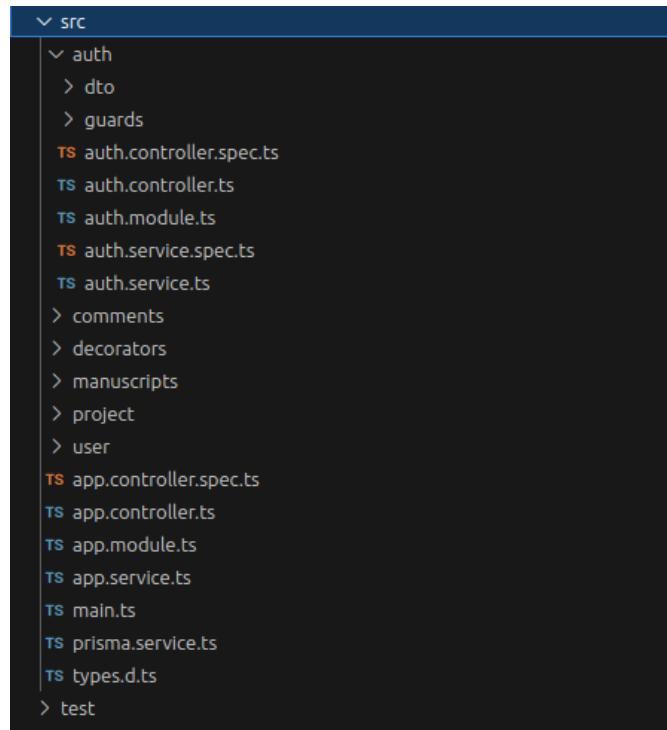
NestJS intègre également TypeScript par défaut, offrant ainsi les avantages du typage statique pour une plus grande fiabilité et une meilleure expérience de développement. De plus, NestJS facilite l'intégration avec d'autres bibliothèques et outils, grâce à son système de modules, et propose un écosystème riche en fonctionnalités pour le développement d'API RESTful, GraphQL, microservices, et WebSocket. Sa prise en charge complète des dernières fonctionnalités de JavaScript et TypeScript, combinée à une documentation exhaustive et une communauté active, fait de NestJS un choix solide pour le développement backend.

Structure du projet

Au top-level du projet se trouvent le dossier racine pour le code source, src, ainsi que les fichiers utilisés pour configurer l'application, gérer les dépendances, exécuter un middleware, intégrer des outils de surveillance ou encore définir des variables d'environnement et les node_modules.



Ensuite, dans le dossier src, on trouve le controller de base de l'application, gérant les requêtes et les réponses (app.controller.ts), le module racine de l'application, qui organise l'application en blocs compacts et réutilisables (app.module.ts), les services de base, contenant la logique métier (app.service.ts). Le point d'entrée de l'application, où l'instance de l'application Nest est créée et exécutée est le fichier main.ts. Un dossier test contenant les tests de l'application est aussi présent.



Chaque dossier sous src représente un module distinct comme auth ou user, et contient des éléments spécifiques tels que des controllers, des services, et des guards, qui gèrent respectivement les requêtes entrantes, la logique métier, et la sécurité. Les dossiers dto et decorators sont utilisés pour définir la structure de données des objets transmis et étendre les fonctionnalités de base, respectivement.

Les fichiers .module.ts relient les composants d'un module, tandis que app.controller.ts et app.service.ts pourraient être utilisés pour gérer les fonctionnalités au niveau de l'application. main.ts est le point d'entrée de l'application, et types.d.ts peut contenir des déclarations de types personnalisés.

L'ORM Prisma

Prisma est un ORM (Object-Relational Mapping). C'est une méthode qui offre aux développeurs la possibilité de gérer les données d'une base de données en utilisant des objets dans leur code, plutôt que de faire des requêtes SQL. Cela simplifie l'utilisation de la

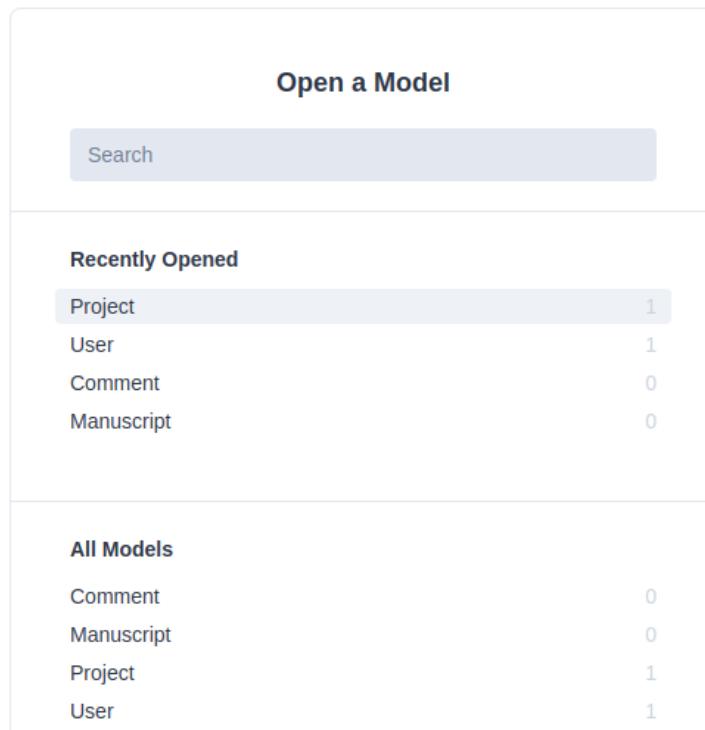
base de données. En bref, l'ORM sert de pont entre le monde des objets de l'application et celui des données stockées dans une base de données relationnelle.

Dans les projets NestJS, Prisma est fréquemment choisi en raison de sa simplicité d'utilisation et de son approche contemporaine de la gestion des bases de données. Avec un modèle de schéma déclaratif, une intégration poussée avec TypeScript, et des outils de migration sophistiqués, Prisma accélère le développement et renforce la sécurité de type.

J'aurais pu utiliser l'ORM TypeORM qui est également relativement simple à configurer avec NestJS, mais il peut nécessiter un peu plus de configuration manuelle pour certaines fonctionnalités avancées. Il est largement adopté dans la communauté NestJS, ce qui signifie qu'il y a beaucoup de ressources et de guides disponibles.

Cependant, Prisma est conçu pour être performant et utilise un moteur de requête compilé en Rust, offrant ainsi des performances optimisées pour les opérations de base de données (explications dans la partie "Under the hood" de la documentation de Prisma). Il fournit un schéma de base de données déclaratif et un client de base de données auto-généré qui rendent les interactions avec la base de données simples et prévisibles. Sa conception vise à réduire les erreurs courantes de manipulation de base de données en offrant un typage fort.

Prisma est relativement nouveau par rapport à TypeORM mais a rapidement gagné en popularité grâce à sa facilité d'utilisation et à son approche innovante de la gestion de base de données. La communauté est active et en croissance, avec un support solide et une documentation claire.



Voici une capture d'écran de l'interface Prisma Studio me permettant de visualiser la base de données (en lançant la commande `npx prisma studio` dans mon terminal).

Sécurité

Côté back-end avec NestJS, nous optons pour une approche sécuritaire en ne stockant pas directement les mots de passe dans notre base de données, mais en enregistrant plutôt un hash de ceux-ci, grâce à la bibliothèque Bcrypt et la fonction `hash()`.

```
async createUser(dto: CreateUserDto) {
  const user = await this.prisma.user.findUnique({
    where: {
      email: dto.email,
    },
  });
  if (user) throw new ConflictException('email duplicated');

  const newUser = await this.prisma.user.create({
    data: {
      ...dto,
      password: await hash(dto.password, 10),
    },
  });
  const { password, ...result } = newUser;

  return result;
}
```

Cette méthode est reconnue pour sa robustesse : elle est conçue pour être intentionnellement lente à exécuter, rendant ainsi la tâche extrêmement difficile pour toute personne tentant de décrypter un mot de passe. Bcrypt ajuste sa complexité en fonction de la capacité du serveur, assurant une sécurité scalable. Lors d'une tentative de connexion, si le mot de passe fourni ne correspond pas au hash stocké, la vérification échoue, empêchant l'authentification. Cette comparaison se fait avec la méthode `compare()` de Bcrypt.

```
async validateUser(dto: LoginDto) {
  const user = await this.userService.findByEmail(dto.username);

  if (user && (await compare(dto.password, user.password))) {
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    const { password, ...result } = user;
    return result;
  }
  throw new UnauthorizedException();
}
```

En cas de succès, l'utilisateur reçoit un token d'authentification généré par une autre bibliothèque que nous utilisons : JWT (Json Web Token). Ce token, crypté et temporaire, contient des informations essentielles comme l'ID de l'utilisateur et est utilisé pour authentifier de manière sécurisée les requêtes ultérieures nécessitant une vérification d'identité. Pour sécuriser certaines routes et s'assurer de l'authenticité de l'utilisateur, nous

avons également mis en place un middleware d'authentification, facilement ajoutable aux routes nécessitant une protection. Nous rafraîchissons aussi le token.

Base de données

Conception de la BDD et diagramme UML

Cette base de données est conçue pour gérer les projets d'écriture où les utilisateurs peuvent travailler sur différents projets, chacun potentiellement lié à un manuscrit. Les manuscrits peuvent être commentés par les utilisateurs, facilitant la collaboration et la révision. La structure de la base de données soutient les relations claires entre les entités et utilise la suppression en cascade pour éliminer les données orphelines, tout en garantissant que les informations critiques ne soit pas perdues par inadvertance.

Le schéma de base de données décrit un système où les utilisateurs (**User**) peuvent créer des projets (**Project**) et des manuscrits (**Manuscript**). Les projets peuvent avoir zéro ou un manuscrit, tandis que chaque manuscrit peut être associé à de multiples commentaires (**Comment**). Les relations entre les utilisateurs, les projets, les manuscrits et les commentaires sont définies avec une suppression en cascade configurée pour maintenir l'intégrité référentielle lorsqu'un utilisateur ou un projet est supprimé.

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model User {
  id        Int      @id @default(autoincrement())
  email     String   @unique
  firstname String
  lastname  String
  password  String
  projects  Project[]
  manuscripts Manuscript[]
  comments   Comment[]
  created_at DateTime @default(now())

  @@map("users")
}

model Project {
  id        Int      @id @default(autoincrement())
  name     String   @unique
  image    String?
  created_at DateTime @default(now())
  user_id  Int
  user     User     @relation(fields: [user_id], references: [id], onDelete: Cascade)
  manuscript Manuscript?

  @@map("projects")
}
```

```

model Manuscript {
    id          Int      @id @default(autoincrement())
    title       String
    file_url   String   @unique
    project_id Int      @unique
    project    Project  @relation(fields: [project_id], references: [id], onDelete: Cascade)
    comments   Comment[]
    user_id    Int
    user       User     @relation(fields: [user_id], references: [id])
    created_at DateTime @default(now())

    @@map("manuscripts")
}

model Comment {
    id          Int      @id @default(autoincrement())
    content     String
    manuscript_id Int
    manuscript  Manuscript @relation(fields: [manuscript_id], references: [id], onDelete: Cascade)
    user_id    Int
    user       User     @relation(fields: [user_id], references: [id], onDelete: Cascade)
    created_at DateTime @default(now())

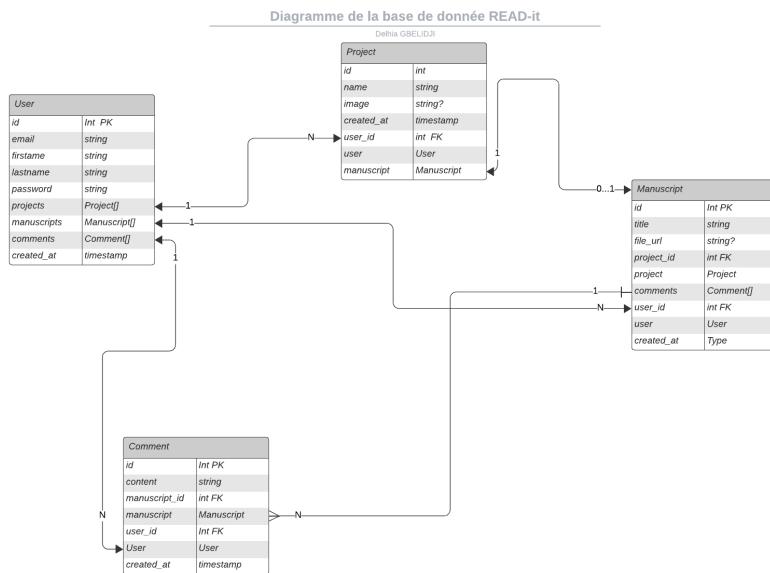
    @@map("comments")
}

```

Le schéma de base de données actuel est conçu de telle sorte qu'il ne requiert pas de tables intermédiaires pour les liens entre les utilisateurs, leurs projets, manuscrits et commentaires, car ces relations sont de type one-to-many. Les tables intermédiaires sont généralement employées pour les associations many-to-many ou quand des relations complexes exigent des informations additionnelles, telles que des attributs de rôles ou de permissions.

Des fonctionnalités nécessitant de telles structures complexes sont à mettre en œuvre à l'avenir, donc l'intégration de tables intermédiaires sera alors nécessaire.

Ci-dessous le diagramme ULM, réalisé sur la plateforme LucidChart :



Pourquoi SQLite ?

SQLite est une solution de gestion de base de données légère et autonome, idéale pour l'environnement de développement. Sans nécessiter d'installation ou de configuration de serveur séparé, SQLite offre une portabilité exceptionnelle grâce à son stockage de données dans un unique fichier, facilitant ainsi le transfert et le partage de données. SQLite est parfait pour les projets nécessitant une base de données simple, efficace, et facilement intégrable, comme notre MVP.

Bien que SQLite excelle dans les applications à trafic modéré et les scénarios où la simplicité et la facilité de déploiement sont cruciales, une base de données plus robuste comme PostgreSQL ou MySQL peut être préférable pour les futures itérations du projet, afin de gérer des charges de travail élevées.

Développement de l'application

L'authentification côté back

Le backend est construit pour la création d'API REST. L'architecture REST (Representational State Transfer) est un ensemble de principes de conception pour le développement d'API web, sans être liée à un protocole ou une norme spécifique. Dans une API conforme à REST, le client envoie une requête pour accéder ou modifier des ressources sur le serveur, et en réponse, le serveur transmet l'état de la ressource dans un format standard tel que JSON, HTML, ou XML, souvent en utilisant JSON pour sa compatibilité universelle et sa lisibilité.

Les en-têtes et les paramètres des requêtes HTTP jouent un rôle crucial, car ils fournissent des metadata essentielles telles que l'authentification, les informations de cache, et les instructions de réponse. Une API RESTful adhère à plusieurs critères clés : elle opère dans une architecture client-serveur où les communications sont sans état, permettant ainsi à chaque requête d'être traitée de manière indépendante. Elle supporte la mise en cache pour optimiser les interactions et assure une interface uniforme qui facilite le transfert d'informations de manière standardisée.

Premièrement, j'ai créé le projet Nest, appelé **read-it-back**, en lançant les commandes dans mon terminal :

```
npm install -g @nestjs/cli  
nest new read-it-back
```

Puis, j'ai installé prisma au sein du projet avec ces commandes :

```
npm install prisma --save-dev  
npx prisma init
```

Cette dernière commande permet d'initialiser le schéma prisma: elle crée un dossier /prisma contenant un fichier schema.prisma. Dans un fichier .env j'ajoute la variable d'environnement DATABASE_URL et le chemin vers le fichier dev.db généré se trouvant dans le dossier /prisma. Ensuite, la commande ci-dessous va permettre d'ajouter le module ConfigModule.forRoot() dans le fichier app.module.ts ce qui permet d'accéder aux variables d'environnement.

```
npm i - --save-dev @nestjs/config
```

Une fois ces étapes d'installation et de configuration, nous définissons le schéma User dans le fichier schema.prisma.

```

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}

model User {
  id        Int      @id @default(autoincrement())
  email    String   @unique
  firstname String
  lastname String
  password String
  projects Project[]
  manuscripts Manuscript[]
  comments Comment[]
  created_at DateTime @default(now())
}

@@map("users")
}

```

Le modèle User défini dans le schéma Prisma, ci-dessus, représente une table users dans la base de données. Chaque utilisateur est identifié de manière unique par un id qui s'incrémente automatiquement. Les utilisateurs ont des champs pour stocker leur email, qui doit être unique, leur firstname, lastname, et password.

La structure permet également de relier chaque utilisateur à plusieurs projects, manuscripts, et comments, créant ainsi des relations one-to-many avec ces tables. Le champ created_at enregistre la date et l'heure de création de l'utilisateur et est automatiquement défini sur le moment de l'insertion de l'utilisateur dans la base de données. Ce modèle est essentiel pour gérer les informations d'identification des utilisateurs et leur association avec divers éléments de la plateforme.

Une fois le schéma défini, on lance :

```
npx prisma migrate dev --name init
```

Cette commande permet de créer la table users dans la base de données et de créer le fichier dev.db dans le dossier /prisma.

La fonctionnalité de connexion est conçue pour être à la fois sécurisée et efficace. Elle commence par la création d'une route API spécifique pour l'authentification. Elle utilise UserService pour récupérer et valider les informations de l'utilisateur lors de la connexion. La méthode login() prend en entrée un objet LoginDto contenant les informations de connexion, telles que le nom d'utilisateur et le mot de passe.

Après validation, un payload contenant l'identifiant de l'utilisateur (son email) et le reste de ces informations est créé et utilisé pour générer deux jetons JWT contenus dans un objet backendTokens : un accessToken avec une courte durée de vie (1 heure) pour les sessions actives et un refreshToken avec une durée de vie plus longue (7 jours). Ces tokens permettent à l'utilisateur de continuer à interagir avec l'application sans avoir à se reconnecter fréquemment. Le refreshToken peut être utilisé pour obtenir un nouvel

accessToken une fois que celui-ci a expiré, sans nécessiter une nouvelle authentification. La date d'expiration réelle du token est calculée et renvoyée avec les tokens.

```
async login(dto: LoginDto) {
  const user = await this.validateUser(dto);
  const payload = {
    id: user.id,
    username: user.email,
    sub: {
      firstname: user.firstname,
      lastname: user.lastname,
    },
  };

  return {
    user,
    backendTokens: {
      accessToken: await this.jwtService.signAsync(payload, {
        expiresIn: '1h',
        secret: process.env.JWT_SECRET_KEY,
      }),
      refreshToken: await this.jwtService.signAsync(payload, {
        expiresIn: '7d',
        secret: process.env.JWT_REFRESH_TOKEN_KEY,
      }),
      expiresIn: new Date().setTime(new Date().getTime() + EXPIRE_TIME),
    },
  };
}
```

La méthode validateUser() est utilisée pour vérifier l'exactitude du mot de passe fourni en comparant le hash stocké avec le mot de passe soumis. Si la vérification échoue, une exception UnauthorizedException est lancée, empêchant l'accès non autorisé.

```
async validateUser(dto: LoginDto) {
  const user = await this.userService.findByEmail(dto.username);

  if (user && (await compare(dto.password, user.password))) {
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    const { password, ...result } = user;
    return result;
  }
  throw new UnauthorizedException();
}
```

Enfin, la méthode refreshToken() permet de régénérer les tokens JWT en utilisant les informations de l'utilisateur, garantissant ainsi que l'utilisateur dispose de tokens valides pour les requêtes ultérieures. Cet ensemble d'outils donne une base solide pour gérer l'authentification et la protection des routes dans votre application.

```
async refreshToken(user: any) {
  const payload = {
    id: user.id,
    username: user.username,
    sub: user.sub,
  };
  return {
    accessToken: await this.jwtService.signAsync(payload, {
      expiresIn: '1h',
      secret: process.env.JWT_SECRET_KEY,
    }),
    refreshToken: await this.jwtService.signAsync(payload, {
      expiresIn: '7d',
      secret: process.env.JWT_REFRESH_TOKEN_KEY,
    }),
    expiresIn: new Date().setTime(new Date().getTime() + EXPIRE_TIME),
  };
}
```

La sécurité des routes API est renforcée par une implémentation personnalisée de la validation JWT, réalisée sans utiliser de bibliothèques comme Passport. Cette approche "from scratch" offre une maîtrise et une compréhension complète des mécanismes d'authentification et de protection des routes.

La classe JwtGuard sert de gardien d'authentification pour protéger les routes. Elle extrait le token JWT de l'en-tête de la requête et utilise JwtService pour vérifier sa validité. Si le token est valide, la requête est autorisée à continuer, sinon, une UnauthorizedException est lancée, bloquant la requête. Ce gardien s'assure que seules les requêtes avec un token valide peuvent accéder aux routes protégées.

```
@Injectable()
export class JwtGuard implements CanActivate {
  constructor(private jwtService: JwtService) {}
  async canActivate(context: ExecutionContext): Promise<boolean> {
    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);

    if (!token) throw new UnauthorizedException();

    try {
      const payload = await this.jwtService.verifyAsync(token, {
        secret: process.env.JWT_SECRET_KEY,
      });
      request['user'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
  }

  return true;
}

private extractTokenFromHeader(request: Request) {
  const [type, token] = request.headers.authorization?.split(' ') ?? [];
  return type === 'Bearer' ? token : undefined;
}
```

Le fichier auth.controller.ts regroupe les endpoints vers l'API auth. On y trouve les types de requêtes HTTP Get et Post. On trouvera les autres types de requêtes dans les controllers des autres ressources du projet (users, projects, manuscripts, comments).

```
@Controller('auth')
export class AuthController {
  constructor(
    private userService: UserService,
    private authService: AuthService,
  ) {}

  @Post('register')
  async registerUser(@Body() dto: CreateUserDto) {
    return await this.userService.create(dto);
  }

  @Post('login')
  async login(@Body() dto: LoginDto) {
    return await this.authService.login(dto);
  }

  @UseGuards(RefreshJwtGuard)
  @Post('refresh')
  async refreshToken(@Request() req) {
    await this.authService.refreshToken(req.user);
  }
}
```

Les APIs manipulent les ressources, comme les tokens d'accès et de rafraîchissement, en utilisant des requêtes HTTP typiques des API REST, telles que POST pour le login et pour le rafraîchissement des tokens. L'usage du JwtGuard pour sécuriser les routes suit les principes REST en maintenant le serveur sans état, en transférant l'état de l'utilisateur via le token à chaque requête. Ce design permet une interaction claire et structurée entre le client et le serveur, typique des API REST.

En résumé, le processus de connexion est un élément clé de l'architecture de sécurité de votre application, permettant une authentification robuste et une protection efficace des ressources de l'API grâce à l'utilisation de tokens JWT et d'un système de refresh tokens.

Chaque CRUD réalisé (user, projects, manuscripts et comments) suivent cette architecture avec un fichier *name.service.ts* implémentant la logique de la route, un fichier *name.controller.ts* définissant les endpoints des différents API, un fichier *name.module.ts* permettant d'importer les différents modules utilisés par la ressource. On retrouve aussi les dto permettant de créer une classe pour la validation des types des données envoyées dans les requêtes. On utilise la librairie **class-validator**.

Ci-dessous des échantillons de code de la ressource users:

user.service.ts :

```
@Injectable()
export class UserService {
  constructor(private prisma: PrismaService) {}

  async create(dto: CreateUserDto) {
    const user = await this.prisma.user.findUnique({
      where: {
        email: dto.email,
      },
    });
    if (user) throw new ConflictException('email duplicated');

    const newUser = await this.prisma.user.create({
      data: {
        ...dto,
        password: await hash(dto.password, 10),
      },
    });
    const { password, ...result } = newUser;

    return result;
  }

  async findByEmail(email: string) {
    return await this.prisma.user.findUnique({
      where: {
        email: email,
      },
    });
  }

  async findById(id: number) {
    return await this.prisma.user.findUnique({
      where: {
        id: id,
      },
    });
  }
}
```

user.controller.ts :

```
@Controller('user')
export class UserController {
  constructor(private userService: UserService) {}

  @UseGuards(JwtGuard)
  @Get(':id')
  async getUser(@Param('id', ParseIntPipe) id: number) {
    return await this.userService.findById(id);
  }

  @UseGuards(JwtGuard)
  @Patch(':id/update')
  async update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateUserDto: UpdateUserDto,
  ) {
    await this.userService.update(id, updateUserDto);
    return { message: 'User updated successfully' };
  }

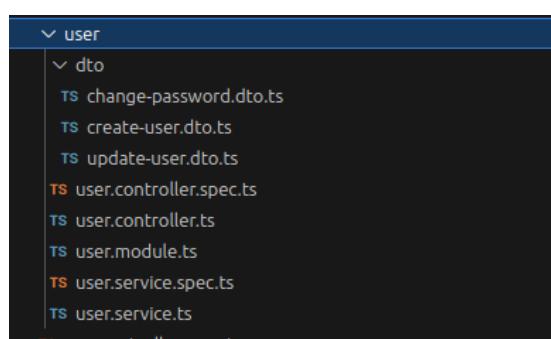
  @UseGuards(JwtGuard)
  @Patch(':id/change-password')
  async changePassword(
    @Param('id', ParseIntPipe) id: number,
    @Body() changePasswordDto: ChangePasswordDto,
  ) {
    await this.userService.changePassword(id, changePasswordDto);
    return { message: 'Password updated successfully' };
  }

  @UseGuards(JwtGuard)
  @Delete(':id')
  async remove(@Param('id', ParseIntPipe) id: number) {
    return await this.userService.remove(id);
  }
}
```

user.module.ts :

```
import { Module } from '@nestjs/common';
import { UserService } from './user.service';
import { UserController } from './user.controller';
import { PrismaService } from 'src/prisma.service';
import { JwtService } from '@nestjs/jwt';

@Module({
  providers: [UserService, PrismaService, JwtService],
  controllers: [UserController],
})
export class UserModule {}
```



Présentation d'une fonctionnalité front : la page Account

La page Account permet à l'utilisateur de modifier ces informations personnelles, changer son mot de passe et supprimer son compte. Elle est organisée en trois onglets correspondant au composant Tabs de Material UI. Chaque onglet render un formulaire géré avec react-hook-form et yup. La fonction onSubmit() de chaque composant permet d'appeler la fonction correspondante dans le fichier app/api/users/route.ts.

Dans le dossier app, on déclare un fichier page.tsx qui va vérifier la session de l'utilisateur et appeler la route API via une requête GET pour récupérer l'utilisateur connecté. S'il n'y a pas de session, l'utilisateur est redirigé vers la page d'authentification.

```
const AccountPage = async () => {
  const session = await getServerSession(authOptions);
  const response = await fetch(BACKEND_URL + `/user/${session?.user.id}`, {
    method: 'GET',
    headers: {
      authorization: `Bearer ${session?.backendTokens.accessToken}`,
      'Content-Type': 'application/json',
    },
  });
  if (!session) redirect('/auth');

  const user = await response.json();

  return (
    <Box
      display='flex'
      justifyContent='center'
      alignItems='center'
      minHeight='90vh'
    >
      <AccountTabs session={session} user={user} />
    </Box>
  );
};

export default AccountPage;
```

Ensuite, le composant AccountTabs est render. On déclare un state value qui est changé lorsque que l'on change d'onglet via la fonction handleChange(). Ensuite, chaque formulaire est appelé dans l'onglet correspondant, dans le composant Styled_TabPanel.

```
5  export const Styled_TabList = styled(TabList)(() => ({
6    '&.MuiTab-root': {
7      '&.Mui-selected': {
8        color: COLORS.black,
9        backgroundColor: COLORS.white,
10       },
11     },
12     '&.MuiTabs-indicator': {
13       backgroundColor: COLORS.pink,
14     },
15   }));
16
17 export const Styled_TabPanel = styled(TabPanel)(() => ({
18   '&.MuiTabPanel-root': {
19     maxWidth: '500px',
20     margin: 'auto',
21   },
22 }));
```

```

export type Type_Props_AccountTabs = {
  session: Session;
  user?: Type_User;
};
const AccountTabs = ({session, user}: Type_Props_AccountTabs) => {
  const [value, setValue] = useState('1');

  const handleChange = (event: React.SyntheticEvent, newValue: string) => {
    setValue(newValue);
  };

  return (
    <Stack direction='column' spacing={2} alignItems='center'>
      <Typography fontSize={33} sx={{fontWeight: 700}}>
        Update your information
      </Typography>

      <TabContext value={value}>
        <Box>
          <Styled_TabList
            onChange={handleChange}
            aria-label='account-tabs'
            centered>
            <Tab label='Personal information' value='1' />
            <Tab label='Change password' value='2' />
            <Tab label='Delete profile' value='3' />
          </Styled_TabList>
        </Box>
        <Styled_TabPanel value='1'>
          <UpdateUserForm session={session} user={user} />
        </Styled_TabPanel>
        <Styled_TabPanel value='2'>
          <ChangePasswordForm session={session} />
        </Styled_TabPanel>
        <Styled_TabPanel value='3'>
          <DeleteUserForm session={session} />
        </Styled_TabPanel>
      </TabContext>
    </Stack>
  );
};

export default AccountTabs;

```

Le formulaire `UpdateUserForm` permet de modifier le nom, prénom et email de l'utilisateur. La requête effectuée est une requête Patch. Lors de la soumission du formulaire, la fonction `updateUser()` est appelée depuis le fichier `route.ts`. Si la réponse est une erreur, on notifie l'utilisateur via un toast, sinon on affiche un message de succès.

Ci-dessous, la fonction `updateUser()` dans `route.ts` :

```

export const updateUser = async (userId: number, data: Type_UpdateUser, token: string) => {
  const response = await fetch(`${BACKEND_URL}/user/${userId}/update`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${token}`,
    },
    body: JSON.stringify(data),
  });

  if (!response.ok) {
    const errorData = await response.json();
    throw new Error(errorData.message || 'Something went wrong');
  }

  return await response.json();
};

```

Et ici la logique du formulaire avec le schéma de validation, l'appel du hook `useForm()` et la fonction `onSubmit()` :

```
const Schema_UpdateUserForm = Yup.object().shape({
  email: Yup.string().email('Invalid email').optional(),
  firstname: Yup.string().optional(),
  lastname: Yup.string().optional(),
});

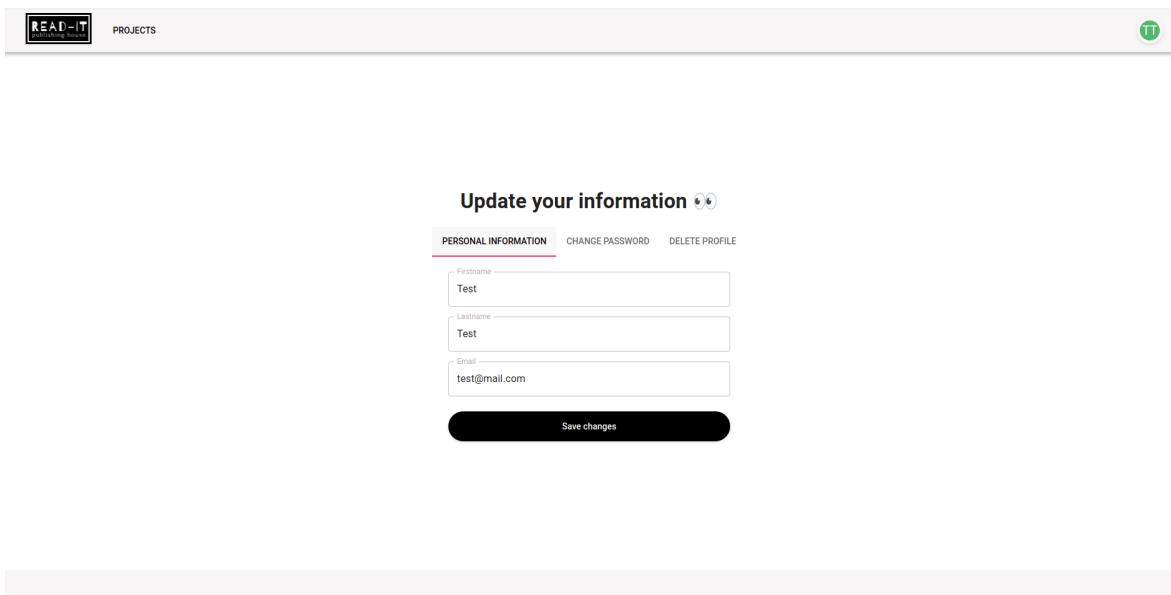
const UpdateUserForm = ({session, user}: Type_Props_AccountTabs) => {
  //Form handler
  const {
    handleSubmit,
    control,
    formState: {errors},
  } = useForm<Type_UpdateUserData>({
    defaultValues: {
      firstname: user?.firstname,
      lastname: user?.lastname,
      email: user?.email,
    },
    resolver: yupResolver(Schema_UpdateUserForm),
  });

  const onSubmit = async (formData: Type_UpdateUserData) => {
    const data: Type_UpdateUser = {
      firstname: formData.firstname,
      lastname: formData.lastname,
      email: formData.email,
    };

    try {
      const {error, response} = await updateUser(
        session.user.id,
        data,
        session.backendTokens.accessToken,
      );

      if (error) {
        notifyError(error);
      }
      notifySuccess(response);
    } catch (error) {
      console.error(error);
    }
  };
}
```

Voici le résultat sur l'interface :





PROJECTS



Update your information ⓘ ⓘ

[PERSONAL INFORMATION](#) [CHANGE PASSWORD](#) [DELETE PROFILE](#)

Current Password	<input type="password"/>
Password	<input type="password"/>
Confirm password	<input type="password"/>

[Update password](#)

Your information have been
successfully updated.



PROJECTS



Update your information ⓘ ⓘ

[PERSONAL INFORMATION](#) [CHANGE PASSWORD](#) [DELETE PROFILE](#)

We are sad to see you leave... 😢

Once you delete your profile, it is definitive. Are you sure?

I confirm I want to delete my profile

[Delete permanently](#)

Présentation d'un test

```
// Mock pour Axios
const mockAxios = new MockAdapter(axios);

// Configuration du mock pour useRouter
jest.mock('next/navigation', () => ({
  useRouter: jest.fn(),
}));

describe('Page', () => {
  let pushMock: jest.Mock;

  beforeEach(() => {
    // Réinitialisation du mock Axios pour chaque test
    mockAxios.reset();

    // Mock de réponse spécifique si nécessaire
    mockAxios
      .onGet('https://www.googleapis.com/books/v1/volumes')
      .reply(200, {});

    // Réinitialisation du mock pour useRouter pour chaque test
    pushMock = jest.fn();
    (nextNavigation.useRouter as jest.Mock).mockReturnValue({
      push: pushMock,
    });
  });

  it('renders without crashing', () => {
    render(<Page />);
  });

  it('redirects to /about-us when clicking the About us button', () => {
    const {getByText} = render(<Page />);
    fireEvent.click(getByText('About us'));
    expect(pushMock).toHaveBeenCalledWith('/about-us');
  });

  it('redirects to /auth when clicking the Join us button', () => {
    const {getByText} = render(<Page />);
    fireEvent.click(getByText('Join us'));
    expect(pushMock).toHaveBeenCalledWith('/auth');
  });

  // Réinitialisez les mocks après chaque test
  afterEach(() => {
    jest.clearAllMocks();
  });
});
```

Ce script de test utilise jest et @testing-library/react pour vérifier le comportement de la landing page.

Il commence par configurer des mocks pour axios et le hook useRouter de Next.js. Le mock d'axios est utilisé pour simuler des réponses à des requêtes GET, garantissant que les tests ne dépendent pas de requêtes réelles à des services externes.

Le mock de useRouter permet de simuler la navigation au sein de l'application sans changer réellement de page durant les tests. Trois tests sont définis : le premier vérifie que la page se rend sans erreur, et les deux autres testent si des clics sur des boutons spécifiques ("About us" et "Join us") entraînent bien des redirections attendues, en utilisant le mock de push pour vérifier que les bonnes URLs sont visées. Ce faisant, ces tests s'assurent que les interactions utilisateur clés déclenchent les navigations prévues, contribuant à valider le routage et les réactions de l'interface utilisateur sans nécessiter un serveur ou une navigation réelle.

Résultat du test après lancement de la commande *npm test* dans le terminal :

```
(use --trace-deprecation ... to show where the warning was created)
PASS  src/app/test/page.test.tsx
  Page
    ✓ renders without crashing (96 ms)
    ✓ redirects to /about-us when clicking the About us button (23 ms)
    ✓ redirects to /auth when clicking the Join us button (20 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:  0 total
Time:        0.989 s, estimated 1 s
Ran all test suites.
read-it ► |
```

Préparation du déploiement

Déploiement du front

Nous avons pensé à un déploiement sur Vercel qui offre un déploiement facile et rapide pour les applications Next.js. Voici un plan étape par étape pour déployer le front :

- Préparation du projet

S'assurer que le projet fonctionne correctement en local. Tester toutes les fonctionnalités pour éviter de déployer des bugs.

Configurer le fichier package.json avec les scripts de build nécessaires s'assurer que le script "build" ne génère pas d'erreurs.

- Création d'un compte Vercel

S'inscrire via Github.

- Configuration du projet sur Vercel

Importer le projet sur Vercel en le connectant au repo GitHub

Configurez les options du projet, telles que les variables d'environnement nécessaires pour le build et l'exécution de l'application.

Let's build something new.

To deploy a new Project, import an existing Git Repository or get started with one of our Templates.

The screenshot shows the Vercel dashboard interface. On the left, under 'Import Git Repository', there is a dropdown menu set to 'Delhiagbelidji' and a search bar labeled 'Search...'. Below the dropdown are five repository cards with names like 'read-it-back', 'shiny-agency', 'la-maison-jungle', and 'Portfolio', each with an 'Import' button. At the bottom of this section is a link 'Import Third-Party Git Repository →'. On the right, under 'Clone Template', there are four template cards: 'NEXT.js' (Next.js), 'SvelteKit (v1)' (SvelteKit), 'Nuxt.js' (Nuxt.js), and 'Vite' (Vite). Each template card has a preview image and a small description below it. At the bottom of this section is a link 'Browse All Templates →'.

[← Back](#)

You're almost done.

Please follow the steps to configure your Project and deploy it.

The screenshot shows the Vercel project configuration interface. On the left, there's a sidebar with a tree view showing 'Configure Project' (selected), 'Deploy', and a 'GIT REPOSITORY' section for 'DelhiaGbelidji/read-it'. The repository has a 'main' branch selected. Below that are links to 'Import a different Git Repository' and 'Browse Templates'. The main area is titled 'Configure Project' and contains fields for 'Project Name' (set to 'read-it'), 'Framework Preset' (set to 'Next.js'), and 'Root Directory' (set to '/'). There are also 'Build and Output Settings' and 'Environment Variables' sections. At the bottom right is a large 'Deploy' button.

- Déploiement

Push le code vers la branche surveillée (par défaut main) pour déclencher automatiquement le déploiement. On peut surveiller l'avancement du déploiement sur l'interface utilisateur Vercel.

- Mise en production

Vercel fournit une URL unique pour chaque déploiement. On peut utiliser cette URL pour tester l'application en conditions réelles avant de la mettre en production.

Si tout fonctionne comme prévu, on clique sur "Promote to Production" sur le tableau de bord Vercel ou la commande CLI correspondante pour mettre à jour le domaine de production.

Pour mettre à jour l'application, il suffit de répéter le processus de déploiement. Vercel gère les mises à jour et les rollbacks de manière fluide.

Déploiement du back

Pour la partie backend, nous avons choisi de déployer sur le serveur interne de l'école. Il faut pour cela conteneurisé le projet backend et la base de données car le serveur héberge d'autres applications.

Voici le plan de déploiement :

- Docker compose et le Dockerfile

Créer un fichier docker-compose.yaml à la racine du projet. Le fichier docker-compose.yaml est utilisé par Docker Compose, un outil pour définir et gérer des applications multi-conteneurs. On va y déclarer les différents services à conteneuriser, l'application et la base de données.

Chaque service peut être configuré pour utiliser une image Docker spécifique, des ports, des volumes, et d'autres paramètres réseau. Il prend en charge la définition de variables d'environnement dans le fichier ou via des fichiers externes, ce qui est essentiel pour personnaliser les conteneurs sans modifier le code de l'application ou le fichier docker-compose.yaml lui-même.

```
version: '3.3'
services:
  nest-api:
    container_name: nest-api
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - 6100:6100
    env_file:
      - .env
    volumes:
      - ./prisma:/app/prisma
      - ./dev.db:/app/dev.db:rw
      - ./app
      - /app/node_modules
```

Ensuite, le fichier Dockerfile, créé aussi à la racine du projet, est un script de configuration qui permet d'automatiser la création d'images Docker. On y inclut les instructions et commandes nécessaires pour assembler une image et donc exécuter l'application.

```
Dockerfile M x
Dockerfile > FROM
1 FROM node:18-alpine3.16 AS builder
2
3 # Create app directory
4 WORKDIR /app
5
6 # A wildcard is used to ensure both package.json AND package-lock.json are copied
7 COPY package*.json .
8 COPY prisma ./prisma/
9
10 # Install app dependencies
11 RUN npm install
12
13 COPY ..
14
15 RUN npx prisma generate
16 RUN npm run build
17
18 FROM node:18-alpine3.16
19
20 WORKDIR /app
21
22 COPY --from=builder /app/node_modules ./node_modules
23 COPY --from=builder /app/package*.json ../
24 COPY --from=builder /app/dist ./dist
25 COPY --from=builder /app/prisma ./prisma
26
27 EXPOSE 6100
28 CMD [ "npm", "run", "start:prod", "start:dev" ]
```

- Cloner le dépôt Git dans le shell du serveur après avoir ajouté le/les ports dont on a besoin dans un fichier app_ports.txt.

- Lancer les containers Docker avec `docker compose up -d` en vérifiant que la connexion entre le back et le front se font bien et qu'il n'y a pas d'erreur. Le `-d` permet de lancer en détaché afin que les conteneurs et donc le site continuent de tourner même après la déconnexion.

Gérer la connexion front/back

Pour connecter le frontend déployé sur Vercel à un backend sur un serveur utilisant Docker Compose, il faut s'assurer que le backend est accessible publiquement avec une adresse IP et des ports correctement configurés ce qui est notre cas. Il faut définir des variables d'environnement dans le projet déployer par Vercel pour stocker l'URL du backend, en s'assurant d'ajuster les règles CORS pour accepter les requêtes depuis notre domaine Vercel. Il faut également vérifier que la communication est sécurisée via HTTPS pour les deux parties.

Dans le cadre de notre projet, nous n'avons pas pu déployer notre application entièrement par manque de temps. Nous avons fait face à une configuration de Docker compliquées. Le plan de déploiement sera réalisé tel que décrit, le plus rapidement possible.

Nous veillerons aussi à assurer une intégration et un déploiement continu grâce aux Github Actions.

Et la suite ?

Nous avons réalisé un projet très simple avec des fonctionnalités minimales afin de rendre les futures itérations plus fluides. Nous avons plusieurs features que nous souhaiterions mettre en place, telles que :

- Un système de téléchargement, de stockage et de versionning du manuscrit,
- L'invitation d'un autre utilisateur au sein d'un projet,
- La mise en place des rôles et des permissions,
- La sécurisation de l'espace projet dans lequel on ne pourrait pas prendre de capture d'écran ou faire de copier/coller afin de préserver l'intégrité du manuscrit et la propriété intellectuelle de l'auteur et de ces collaborateurs,
- Un système de conversion du manuscrit en format epub afin de pouvoir le commercialiser comme livre électronique.

Cette liste, non exhaustive, est une invitation à de nouveaux défis et de nouvelles ambitions.

Pour ma part, j'ai appris énormément de choses mais je suis consciente que j'ai encore beaucoup à apprendre. Dans le but de m'améliorer et de pouvoir avancer en développant des features plus compliquées, je souhaiterais, en amont, effectuer un travail de refactorisation du code dans la partie Nextjs: isoler certains composants qui peuvent être réutilisables, revoir l'architecture du front et notamment l'utilisation des fichiers route.ts.

J'aimerais aussi continuer de travailler la partie backend pour laquelle je me suis découvert une appétence. Je souhaiterais migrer vers une base de données plus solide telle que PostgreSQL ou MySQL qui m'apparaissent plus robustes pour le produit que nous souhaitons continuer de développer.

Quant à Karim, il souhaiterait mettre en place une couverture de test automatisés, comme il apprend à le faire actuellement chez SNCF Connect.

Conclusion

Le projet READ-IT, malgré les obstacles et les défis rencontrés, s'est avéré être une expérience enrichissante et formatrice. Les changements de technologie (changement de Supabase vers un backend que j'ai construit) et les complications survenues durant le développement ont été des occasions d'apprendre et de s'adapter, mettant en évidence la dynamique et l'évolution constante du domaine du développement logiciel. Ce voyage à travers les différentes phases du projet m'a permis une montée en compétences significative, notamment dans les domaines du backend et des bases de données, qui étaient auparavant moins familiers.

Bien que l'échelle du projet ait été plus modeste, il a offert une perspective précieuse sur la conception globale d'un projet professionnel, simulant les défis et les choix stratégiques auxquels sont confrontées les équipes de développement à chaque étape de conception. Cette expérience a non seulement renforcé les compétences techniques, mais a également fourni une compréhension profonde des processus de développement, me préparant pour des projets futurs encore plus ambitieux.

Par ailleurs, j'ai également pris connaissance de ce que je pouvais améliorer, au sein de ce projet, mais aussi dans ma routine de développeuse. Qu'elles soient humaines ou techniques, ces améliorations me permettront de continuer ma montée en compétence. Mon apprentissage ne se termine pas à ce projet ni même mon alternance. Apprendre est fondamental dans ce métier. Je continuerai d'exercer ma curiosité et ma rigueur au quotidien.

Bibliographie

FRONT

- Next.js : <https://nextjs.org/docs>
- Next-Auth : <https://next-auth.js.org/getting-started/introduction>
- MUI : <https://mui.com/material-ui/getting-started/>
- React hook form : <https://react-hook-form.com/>
- Yup : <https://github.com/jquense/yup>
- Slick : <https://kenwheeler.github.io/slick/>
- Google API : <https://developers.google.com/books?hl=fr>

BACK

- Nest : <https://docs.nestjs.com/>
- Prisma : <https://www.prisma.io/docs> et <https://www.prisma.io/stack>
- Class-validator : <https://github.com/typestack/class-validator>
- SQLite : <https://www.sqlite.org/docs.html>
- Docker-compose : <https://docs.docker.com/compose/>

AUTRES RESSOURCES

- StackOverFlow : <https://stackoverflow.com/questions>
- ChatGPT : <https://chat.openai.com/>
- Mozilla : <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- <https://www.prisma.io/docs/orm/more/under-the-hood/engines>
- LucidChart : <https://www.lucidchart.com/pages/fr>
- Vercel: <https://vercel.com/home>