Dossier projet

RNCP 37873 – Concepteur développeur d’applications

Benjamin Dromard

Table of Contents

Introduction

Après une dizaine d'années d'études et de travail de recherche comme historien, spécialiste de l'histoire économique et sociale de la Mésopotamie antique, j'ai choisi de me réorienter vers le développement logiciel. Ceci fut alors motivé par une envie d'acquérir des compétences utiles dans le domaine des humanités numériques, pour la conception de bases de données et d'outils permettant leur exploitation à destination de la recherche en sciences sociales.

Pour commencer ma formation au développement web, j'ai choisi de suivre la formation d'Ada Tech School, afin d'obtenir le titre de *Concepteur et développeur d'applications* après un cursus de deux ans. Suite au premier module de neuf mois, j'ai rejoint l'entreprise *Hubblo* comme développeur logiciel en alternance.

Hubblo est une entreprise dont les activités se concentrent sur l'étude des impacts environnementaux des services numériques, dans le but d'améliorer leur mesure et d'essayer de les réduire. Si l'essentiel des activités d'Hubblo à mon arrivée sont plutôt celles d'un bureau d'études, avec la réalisation de travaux de recherche notamment pour l'*Agence de l'environnement et la maîtrise de l'énergie* (ADEME), *Hubblo* souhaite développer son activité de développement d'outils logiciels. Ceux-ci doivent notamment permettre de faciliter la mesure des impacts environnementaux de services numériques.

C'est dans le contexte d'accompagnement du développement de cette activité que j'ai donc rejoint *Hubblo*. Mon travail s'est partagé sur plusieurs projets, et depuis l'été 2024 il se concentre sur *Carenage*, un outil d'automatisation de la mesure d'impacts environnementaux d'un logicie dans le contexte d'une chaîne d'intégration continue. En termes de spécialisation, j'ai essentiellement travaillé comme développeur *back-end*, avec la reprise du travail sur une API répondant à des requêtes HTTP (*Boagent*), puis *Carenage*. Ce dernier projet comporte toutefois une couche *front-end*, avec la réalisation d'un tableau de bord pour visualiser les métriques produites au cours d'une exécution d'une chaîne d'intégration continue.

A côté de mon travail en alternance, avec Camille Hébert et Justine Lambert, nous avons conçu et implémenté une application web de prêts de livres entre ami-e-s, titrée *Boukin*. Ce projet est présenté dans ce dossier et constitue la principale réalisation pour l'obtention du titre de *Concepteur et développeur d'applications*. Il s'agit d'un projet web disposant d'une base de données, d'une API REST et d'une interface utilisateur-ice.

Résumé du projet

Version française

Boukin permet la gestion de prêts de livres entre ami-e-s. Peu de personnes tiennent un registre de ses livres prêtés, se retrouvant souvent face à la situation de ne plus savoir où se situe tel ouvrage… Et si certaines amitiés durent toute une vie, ce n'est pas toujours le cas et il peut être incommode de devoir demander le retour des objets dus.

Boukin a été conçu pour résoudre ce problème, comme un lieu neutre qui servirait d'aide-mémoire tout comme de point d'échange. Il permet de rester en contact avec toute personne à qui on a pu prêter ou emprunter un livre ; de garder en mémoire notre bibliothèque ; et d'avoir toujours à l'œil le statut de nos différents prêts / emprunts.

Boukin se constitue donc comme un site web, demandant un ensemble minime d'informations personnelles, qui permet de se constituer une bibliothèque, de retrouver nos ami-e-s, et de pouvoir leur demander ou leur accorder le prêt d'un livre. Une date de retour peut être fixée, et l'application permettra de remémorer à chacun-e ce qu'il doit rendre. Plus besoin dès lors de chercher où notre roman préféré a pu se cacher si l'on a oublié qu'on l'avait confié à quelqu'un d'autre…

English version

Boukin is a web application for managing book loans between friends. It is rare to encounter someone who keeps a register of his / her loaned books. Therefore, it becomes quite common to forget where one has put some publication or another. And if some friendships can last a lifetime, it is not always the case and it can become a pain to have to ask for the return of our possessions.

Boukin is meant to solve this problem, as a neutral place who can remind you of where such book or another is located, but also that could help you recover what was thought lost. It allows to keep in touch with anyone one may have loaned a book to or borrowed one from. *Boukin* can also serve a personal library, that can help quickly see the status of each of our different lended books.

As a website, asking for a limited number of personal data, *Boukin* helps you constitute your own library, to find your friends and register them, and then allows for books to be loaned and borrowed. An end date for the loan can be set and the website could remind someone to send a book back to its owner. It is therefore no longer needed to find where your favorite novel has been hiding if you had forgotten that it was being read by one of your friends…

Cahier des charges

Besoins et objectifs

Boukin veut avant tout résoudre, par le biais d'une interface disponible dans un navigateur web ou mobile, les problèmes liés à la gestion de prêts de livres. Parmi les utilisateur-ices potentiel-les, il peut y avoir des personnes qui prêtent / empruntent peu ou beaucoup de livres auprès de leur entourage. Cela n'a finalement que peu d'importance pour *Boukin* : ces deux usages peuvent trouver leur solution avec *Boukin*, pouvant aussi bien convenir à une utilisation très sporadique ou très fréquente. Il faut du temps pour lire un livre ; il devrait en falloir peu pour retrouver à qui on l'a prêté. Et plutôt que s'en tenir à l'organisation personnelle de chacun-e, *Boukin* doit permettre de simplifier la création de rappels quant au rendu de prêt, ou la possibilité de voir les livres de ses ami-e-s et pouvoir ainsi les emprunter.

Plusieurs objectifs fonctionnels sont donc à implémenter dans *Boukin* :

- **une simplicité d'utilisation** : il s'agit essentiellement d'un site web, dont l'interface s'adapte aussi bien aux écrans d'ordinateur que mobile et donc facilement accessible sur tous types d'appareil. Un navigateur suffit donc pour s'authentifier et avoir accès aux différentes fonctionnalités de *Boukin*.
- **création de bibliothèque** : chaque utilisateur-ice doit pouvoir ajouter les livres qu'elle / il souhaite prêter, et pouvoir modifier leur disponibilité et visibilité envers ses contacts. *Boukin* doit donc permettre la recherche d'ouvrages, avec les informations nécessaires pour leur identification.
- **gestion des contacts** : chaque utilisateur-ice doit pouvoir retrouver ses ami-e-s, sans pouvoir être sollicité-e par des personnes qu'elle ne connaît pas. N'étant pas un réseau social public, *Boukin* a vocation à conserver le caractère privé de ces relations.
- **gestion des prêts** : l'objectif principal de l'application, l'ensemble des possibilités quant au statut d'un prêt (demande en cours / actif / refusé / terminé) doit pouvoir être géré par *Boukin*. L'interface doit pouvoir montrer simplement et clairement ces différents cas de figure, et permettre à l'utilisateur-ice d'agir s'il y a besoin.

Fonctionnalités clés

Pour répondre à ces besoins et à ces objectifs, le projet demande l'implémentation d'un ensemble de fonctionnalités composant le *Minimum Viable Project* (MVP).

- **Création de compte** : il doit pouvoir être possible de créer un compte, avec des identifiants permettant l'authentification, la consultation des données personnelles. Toute donnée personnelle doit pouvoir être modifiée par l'utilisateur-ice.
- **Connexion au compte** : avec les identifiants adéquats, il est possible de se connecter à l'application et à l'ensemble de ses fonctionnalités.
- **Ajout et suppression de contacts** : il doit pouvoir être possible, avec l'adresse email d'une personne que l'utilsateur-ice connaît, de l'ajouter à ses contacts. Si cette personne accepte la

mise en contact, il est ensuite possible de voir ses livres, et de créer des prêts entre elles. De plus, il doit être possible de supprimer une personne de ses contacts.

- **Ajout et suppression de livres** : un-e utilisateur-ice doit pouvoir ajouter des livres dans sa bibliothèque personnelle, et choisir les ouvrages qu'il / elle veut rendre disponible et visible à ses contacts. La recherche de ces livres peut se faire par ISBN en premier lieu, sinon par titre ou nom d'auteur-ice.
- **Demande d'emprunt de livre** : si une personne fait partie des contacts d'un-e utilisateur-ice, il doit être possible de demander l'emprunt des livres qui sont disponibles dans la bibliothèque de cette personne. Cette personne peut refuser la demande d'emprunt.
- **Prêt de livre** : un-e utilisateur-ice doit pouvoir prêter n'importe lequel de ses livres à l'un de ses contacts. Cela permet de simplifier le processus de prêt de livre si les deux personnes se sont entendues à ce sujet, et que le livre prêté est déjà en possession de la personne emprunteuse.

Pour réprésenter l'expérience utilisateur-ice typique d'une fonctionnalité, il peut être utile de représenter le déroulement logique de cette fonctionnalité et les différentes possibilités selon les choix de l'utilisateur-ice. Un diagramme de type *user flow* permet de documenter ces possibilités et les décisions prises à ce sujet, afin de pouvoir s'y référer au moment de l'implémentation.

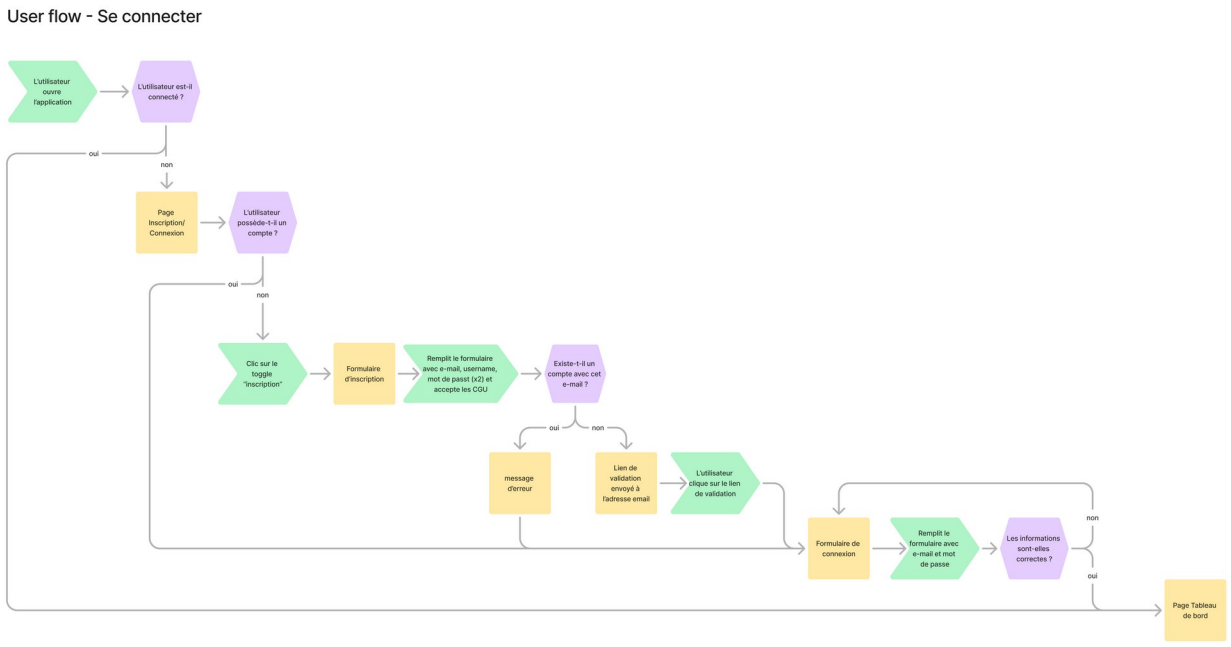


Figure 1: Diagramme de flux : se connecter à Boukin

Charte graphique

En parallèle de la réflexion concernant l'utilisation de l'application, et les différentes fonctionnalités de première importance à implémenter, son aspect visuel devait aussi être établi en amont. En premier lieu, une charte graphique a été conçue, permettant de conserver une cohérence visuelle pour l'ensemble des éléments de l'interface.

La charte graphique, outil de référence au cours du développement, détermine aussi l'atmosphère générale de l'application et le public visé par celle-ci. Nous avions essentiellement en tête des bibliophiles, ou en tout cas des personnes qui seraient attirées par une ambiance propre à une certaine perception de la bibliophilie : des bibliothèques calmes, de grande envergure, supposées typiques des universités médiévales ou modernes en Europe occidentale, avec une préférence pour des couleurs automnales.

Les couleurs primaires et secondaires ainsi que la police ont donc été choisies pour essayer de relayer cette atmosphère. Elle a été constituée par le biais de **Figma**, un outil permettant la conception de composants et de pages constituant l'interface utilisateur, sans l'implémenter.

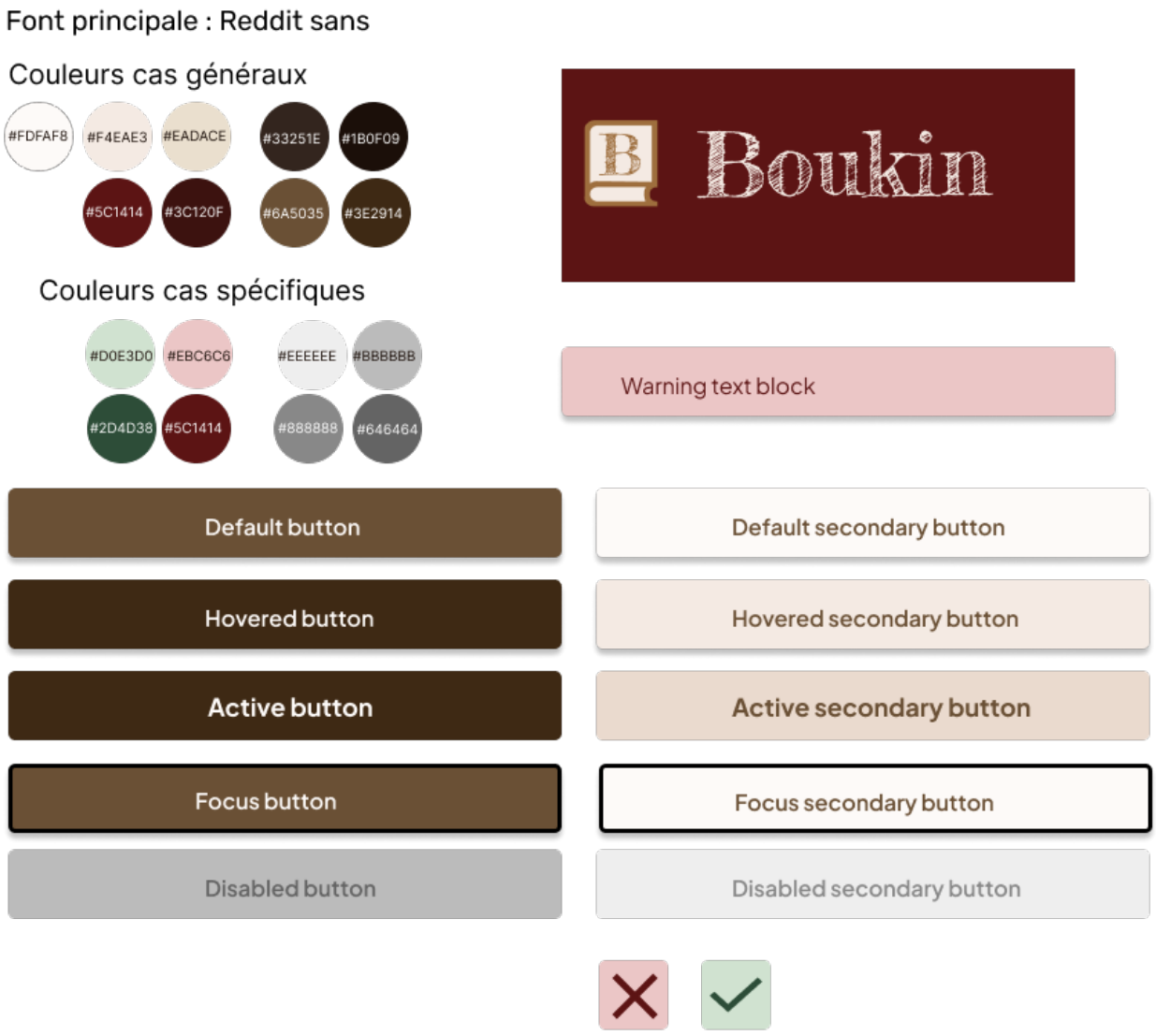


Figure 2: Charte graphique de Boukin

Maquette

Après nous être mis-es d'accord quant à la charte graphique, et donc sur l'atmosphère générale de l'application, il nous fallait élaborer ces composants et pages qui constitueraient l'interface utilisateur-ice. **Figma** permet de créer ces éléments, utiles ensuite comme références pour l'implémentation de la couche *front-end* proprement dite de l'application.

Par le biais de **Figma**, il est possible de mettre en place une élaboration collaborative d'éléments graphiques, facilitant ainsi la collaboration entre équipes de design et équipes de développement. La constitution de *mock-ups* est donc une étape clé de la conception de l'interface de l'application finale, permettant ainsi de visualiser en équipe le rendu final sans avoir à l'implémenter.

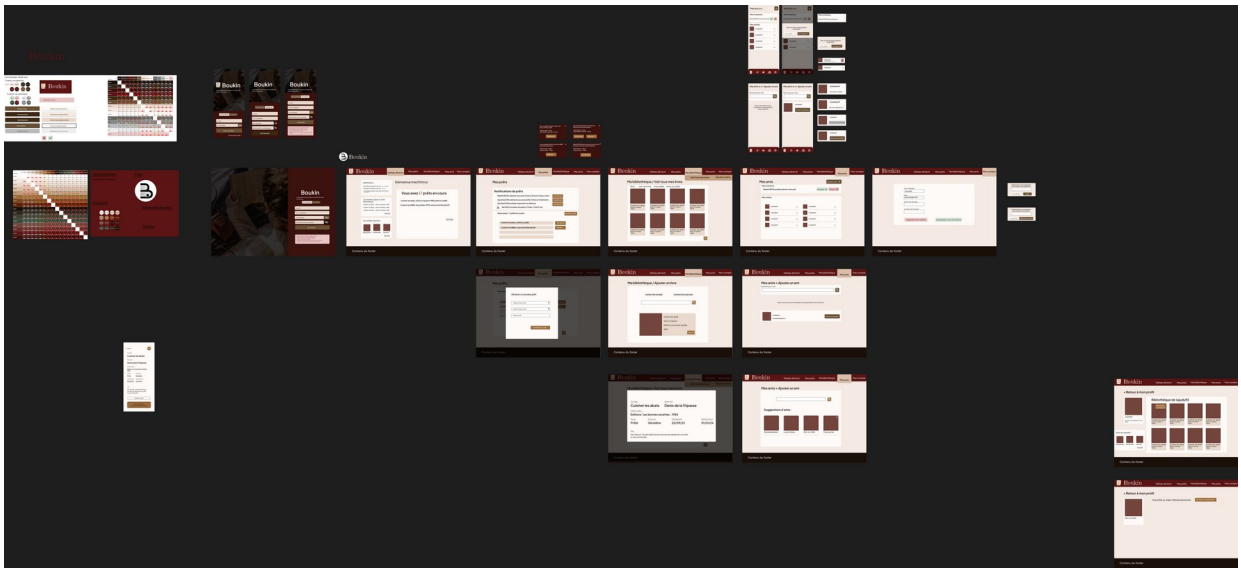


Figure 3: Maquette de Boukin

L'ensemble de la maquette permet aussi de parcourir l'ensemble du parcours utilisateur à travers l'application, en rassemblant les pages propres à une fonctionnalité ensemble et, si besoin est, de créer le rendu des changements d'état selon comment l'utilisateur-ice a interagi avec l'application. Ainsi, les différents *mockups* de la page d'inscription (qui est la page par défaut du site si l'on n'est pas authentifié-e) permettent de montrer ce qui peut être attendu aussi bien par l'utilisateur-ice dans certains cas, mais aussi au cours de l'implémentation de cette page : ici, on s'attend à avoir un message d'erreur dans le cas où un mot de passe choisi pour l'inscription ne correspond pas aux besoins de sécurité de l'application.

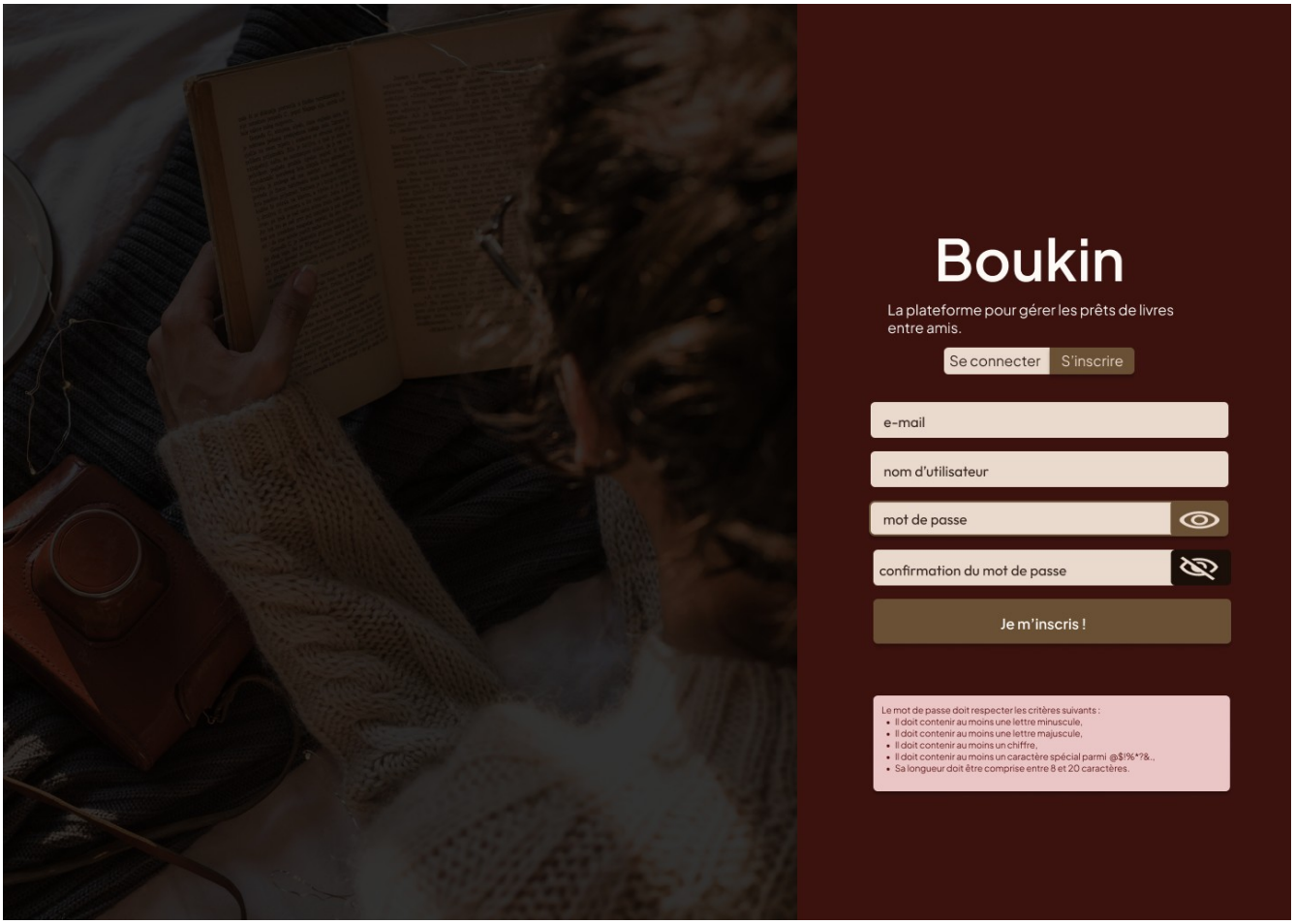


Figure 4: Mockup de la page d'inscription de Boukin

Les différents *mockups* permettent aussi de visualiser les différences entre la version sur grand écran, et la version sur écran mobile. Etant une application disponible dans un navigateur web, les différences de résolution d'écran doivent être prises en compte dans l'implémentation de l'interface. Pour chaque page (page des prêts, page de gestion des amitiés, de gestion des livres), une version grand écran et une version écran mobile ont donc été produites.

Figure 5: Mockup de la version mobile de la page d'inscription de Boukin

Si certaines informations devaient être visibles pour l'utilisateur-ice après avoir réalisé une action (comme la création d'un prêt, ou la confirmation d'une demande de prêt), les éléments de l'interface permettant de les visualiser sont aussi conçues aux côtés des pages correspondantes. Pour cet exemple, les différentes modales apparaissant à l'écran furent donc modélisées sur **Figma**.

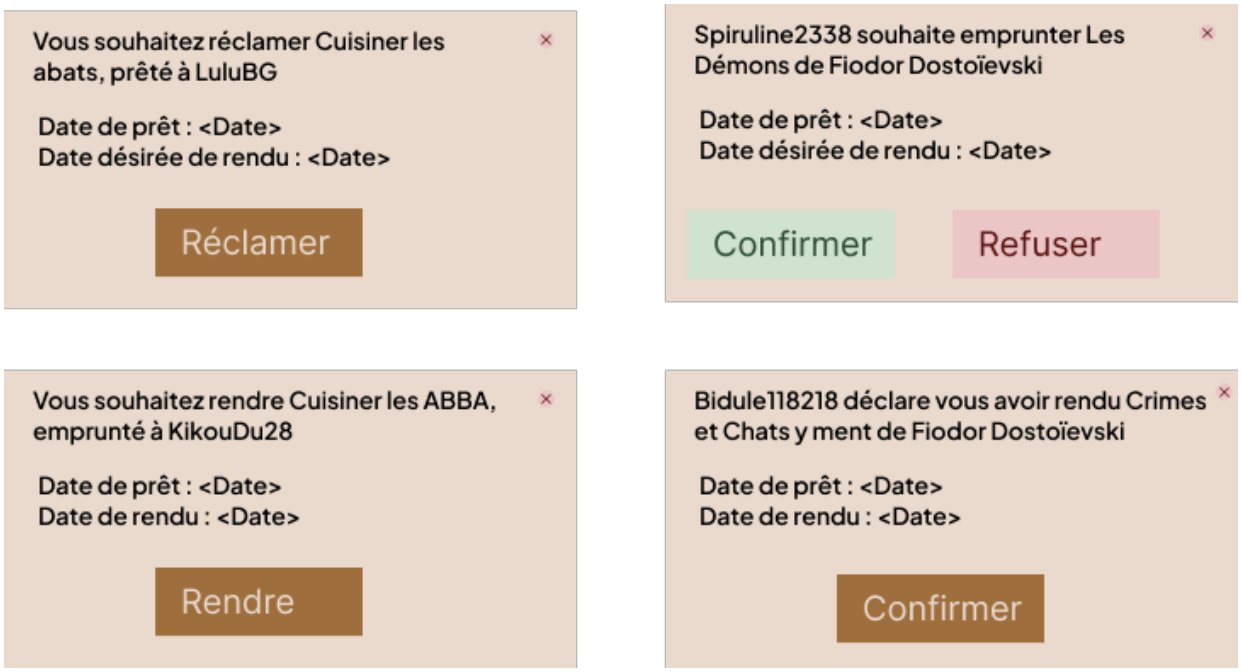


Figure 6: Différentes modales selon l'action à réaliser pour un prêt / emprunt

L'élaboration de la maquette, tout comme celle de la charte graphique, furent donc des étapes importantes de la conception de l'application, permettant ainsi à chacun-e d'entre nous de se référer à ces outils pour implémenter une identité visuelle et une interface utilisateur-ice cohérentes.

Gestion de projet

Pour la réalisation d'un *Minimum Viable Product* au cours des huit mois de temps disponible, une organisation du travail a été mise en place. Elle nous a permis de prioriser les fonctionnalités à implémenter, de bien se répartir les tâches tout en faisant en sorte que chacun-e soit satisfait-e de sa part de travail.

L'organisation générale du projet, en réponse au cahier des charges et aux besoins de la couche *back-end* comme de la couche *front-end*, s'est faite par l'intermédiaire de l'outil **Notion**. Sur un document spécifique, nous avons listé les fonctionnalités attendues du MVP, en ne détaillant pas l'implémentation à suivre. Pour chaque session de travail collective le vendredi, une entrée du journal de bord était rédigée pour documenter le travail fait et les décisions prises.

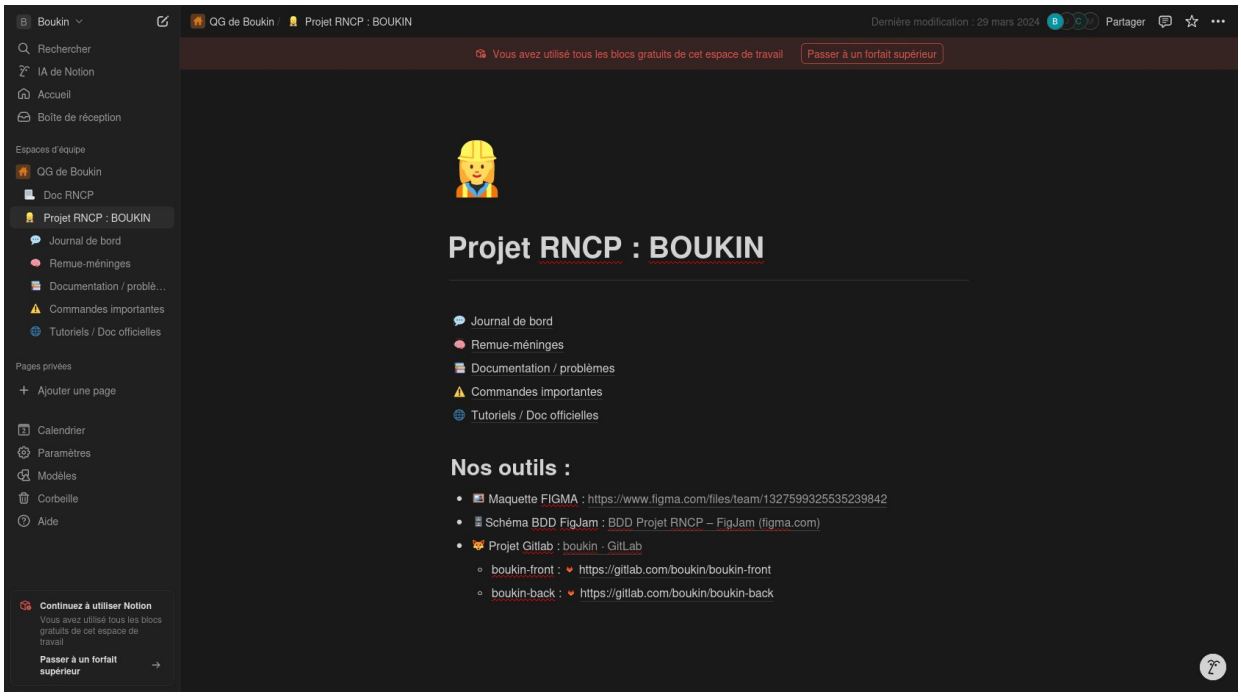


Figure 7: Page principale du Notion pour le développement de Boukin

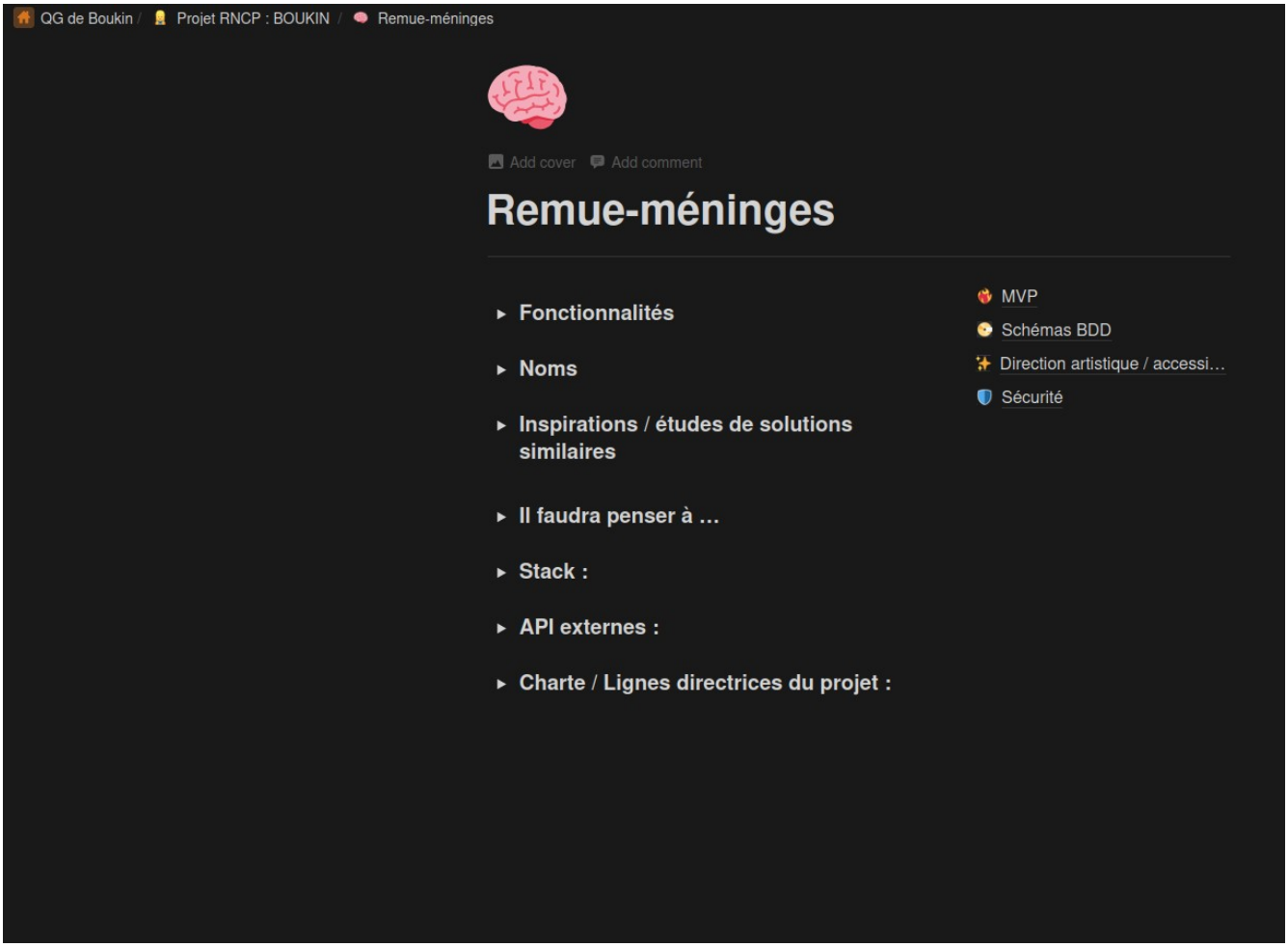


Figure 8: Page rassemblant la documentation des sessions de réflexions collectives

La rédaction des spécifications des fonctionnalités et le suivi des problèmes à résoudre furent essentiellement organisés à travers l'onglet *Issues* (tickets) de notre dépôt **Gitlab**. Chacun des tickets était décrit avec une liste des réalisations attendues, pouvant éventuellement faire l'objet de discussions en commentaires sur le ticket. Chaque ticket était attribué à l'un-e d'entre nous. Pour chaque ticket, une branche issue de la branche `dev` était créée.

Gitlab permet, grâce au nom d'une branche, de lier automatiquement la branche au ticket correspondant. En effet, chaque ticket dispose d'un numéro attribué par **Gitlab**. Ainsi, une branche dénommée `18-create-view-to-update-loan-status` est rattachée au ticket n°18, "Create view to update loan status". Lorsqu'une personne considérait avoir terminé le travail du ticket, une *merge request* liée à la branche, et donc au ticket, était proposée.

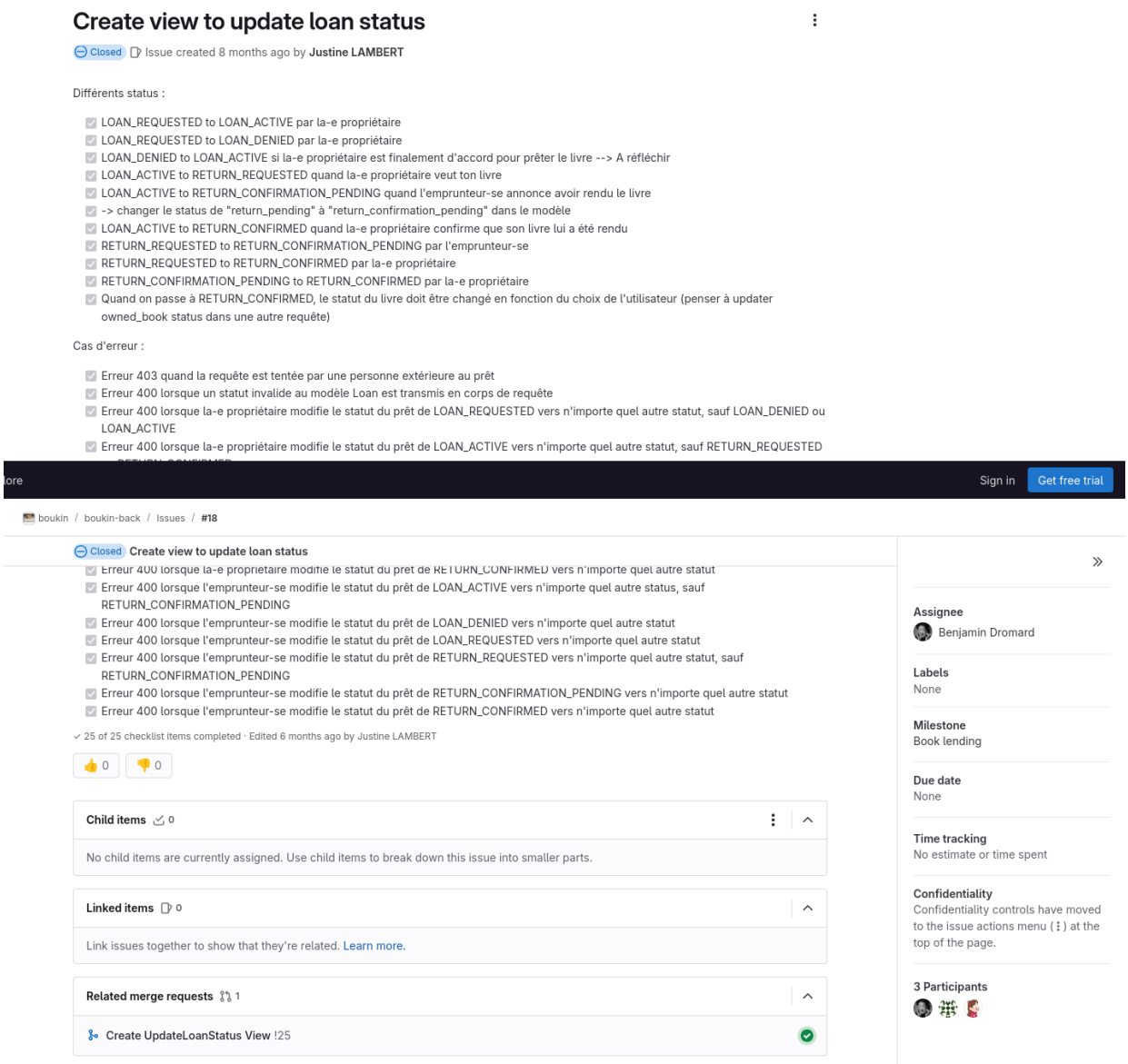


Figure 9: Exemple de ticket sur Gitlab, pour une fonctionnalité côté back-end

Toute *merge request* devait être lue et approuvée par chacun-e d'entre nous. Il n'était pas possible d'ajouter du nouveau code sans que les tests aient été exécutés avec succès dans la chaîne d'intégration continue. Tout nouveau code se devait d'être accompagné des tests correspondants.

Pour la rédaction des messages de *commit*, afin de disposer d'un historique clair, nous utilisions les règles de *conventional commits* propres à la notation **Angular** (https://github.com/angular/angular/blob/22b96b9/CONTRIBUTING.md#-commit-message-guidelines). A chaque *commit* est attribué une étiquette parmi les suivantes : *build* (configuration du projet, dépendances externes), *ci* (intégration continue), *docs* (documentation du projet), *feat* (ajout de

code applicatif), *fix* (correction de *bugs* dans le code applicatif), *refactor* (réécriture et optimisation du code applicatif), *test* (code de test).

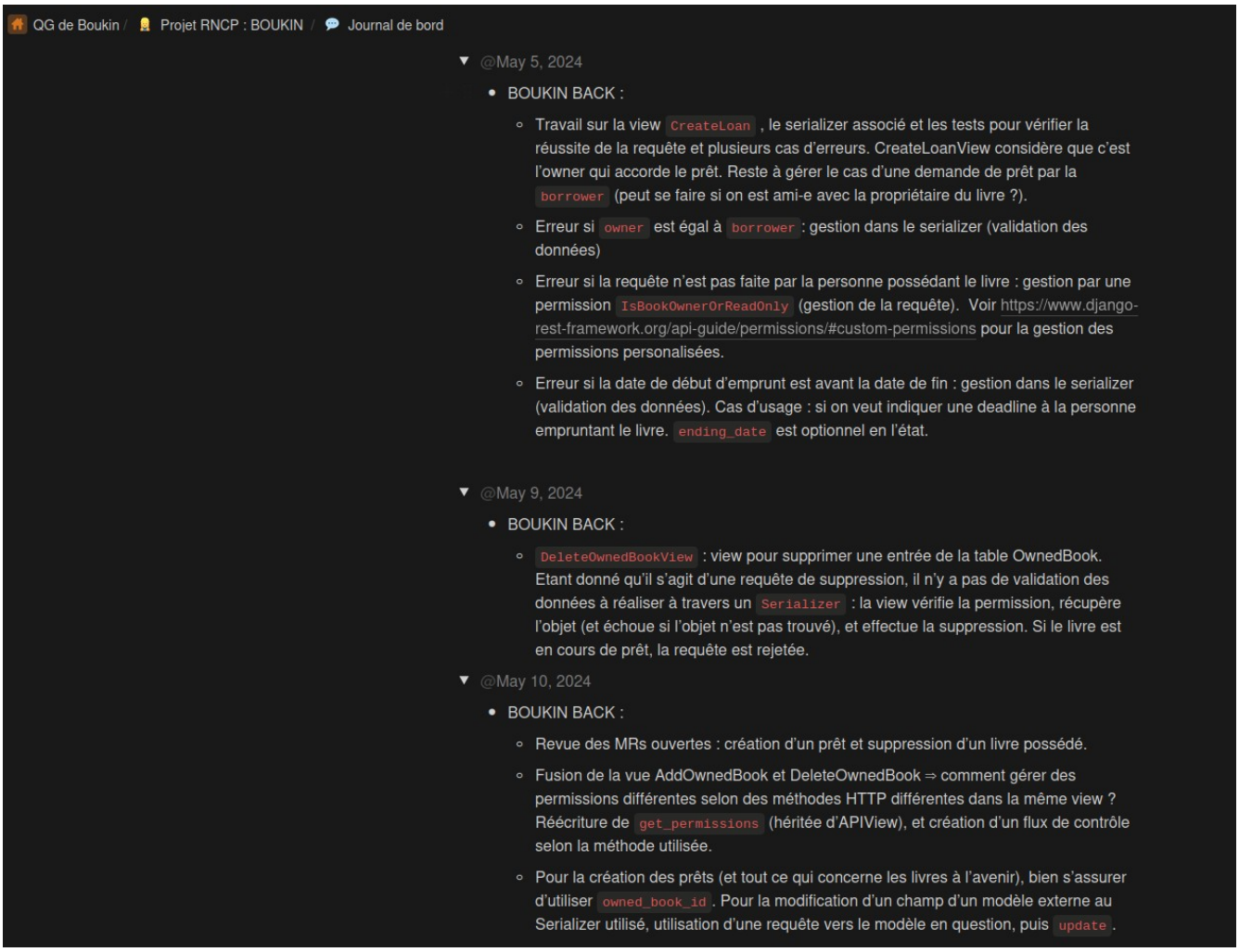


Figure 10: Exemple d'entrées du journal de bord du projet Boukin

Si le début du projet était surtout constitué de sessions de *mob-programming* au cours des vendredis, afin de pouvoir s'approprier au mieux les différents composants de notre spécification technique (**Django**, **Nuxt**, **PostgreSQL**, **Docker**), cela convenait moins à compter du moment où chacun-e pouvait avancer en autonomie sur ses tickets.

Dès lors, le vendredi permettait en premier lieu à la résolution des *merge requests* de manière collective, ou au travail à plusieurs si l'un-e d'entre nous rencontrait un problème dans la réalisation d'une fonctionnalité.

A plusieurs échéances au cours de l'année, et après qu'une session de tickets ait été résolue, de nouveaux tickets étaient rédigés, ce qui nous permettait de voir où nous en étions dans la réalisation générale du MVP.

Spécifications fonctionnelles du projet

Afin de présenter plus en détail le fonctionnement de l'application et son implémentation, les fonctionnalités clés de *Boukin* vont être présentées dans cette section. Cela me permet ainsi de dévoiler la manière dont nous avons conçu ces fonctionnalités et ce que cela a impliqué pour leur développement.

Inscription et authentification

Pour l'utilisation de *Boukin*, il est nécessaire de s'inscrire à l'application. Etant donné que *Boukin* permet de gérer ses prêts et emprunts de livres dans son réseau de connaissances, un-e utilisateur-ice doit pouvoir être retrouvé-e par le biais de son adresse email par quelqu'un qui la connaît. L'accès aux données permettant l'utilisation de l'application propres au compte d'une personne (livres, prêts, amitiés) n'est possible que si cette personne est authentifiée. Un-e ami-e de cette personne ne peut voir que les livres que celle-ci a décidé de rendre visibles.

Un minimum de données personnelles est demandé, permettant l'inscription et l'authentification : une adresse email, un nom d'utilisateur-ice. A cela s'ajoute un mot de passe qui doit respecter certaines conditions : il doit comporter au moins une lettre minuscule, une lettre majuscule, un caractère spécial et un chiffre, et doit être compris entre 8 et 20 caractères. Ces conditions assurent que le mot de passe ne puisse pas être facilement identifié par technique par force brute, et invitent à ce qu'il ne soit pas un schéma courant et peu sûr.

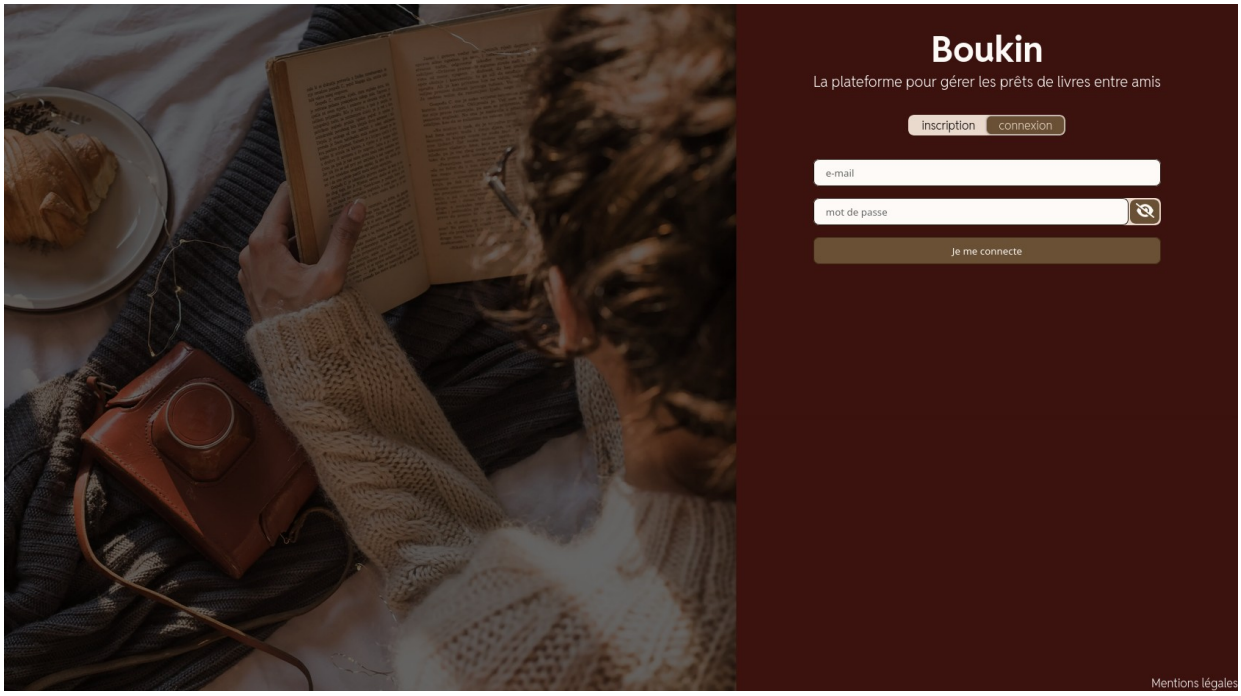


Figure 11: Page d'inscription et de connexion de Boukin

Lorsque une personne visite la page principale de *Boukin*, sans être authentifiée, elle doit voir un formulaire d'inscription et un formulaire de connexion à l'application. A l'inscription, si les identifiants sont valides (adresse email unique et mot de passe conforme), un email est ensuite envoyé à l'adresse renseignée pour confirmation de l'inscription, évitant ainsi les inscriptions automatiques par robots ou par personnes voulant usurper l'identité de quelqu'un. Après réception du message et avoir cliqué sur le lien de confirmation, une personne peut ensuite s'authentifier sur *Boukin* et avoir accès à l'ensemble de ses fonctionnalités.

Si une personne est déjà inscrite, l'utilisation du formulaire de connexion permet de renseigner les identifiants (email et mot de passe), qui permettent l'authentification sur le reste du site. Si une erreur a été identifiée dans l'un des deux identifiants, un message est visible, ne précisant pas l'identifiant erroné.

Gestion du profil

Toute donnée personnelle enregistrée auprès de *Boukin* doit pouvoir être modifiée. *Boukin* n'utilisant que peu d'éléments d'identification personnelle, cela se limite à la modification de l'adresse email, du nom d'utilisateur-ice et du mot de passe. Pour permettre cette modification des données, une page de profil est disponible. Elle présente des éléments interactifs, contenant les informations connues et où une personne peut entrer la nouvelle donnée correspondante. Là aussi, la nouvelle adresse email enregistrée ne doit pas être déjà connue, et le mot de passe doit être conforme aux règles de validation.

Après validation par mot de passe de l'action, la donnée est modifiée en base de données. Si l'on modifie l'adresse email, une confirmation par l'adresse email renseignée doit être faite. C'est aussi sur cette page qu'il est possible de se déconnecter de l'application, et à terme de supprimer son compte de *Boukin*.

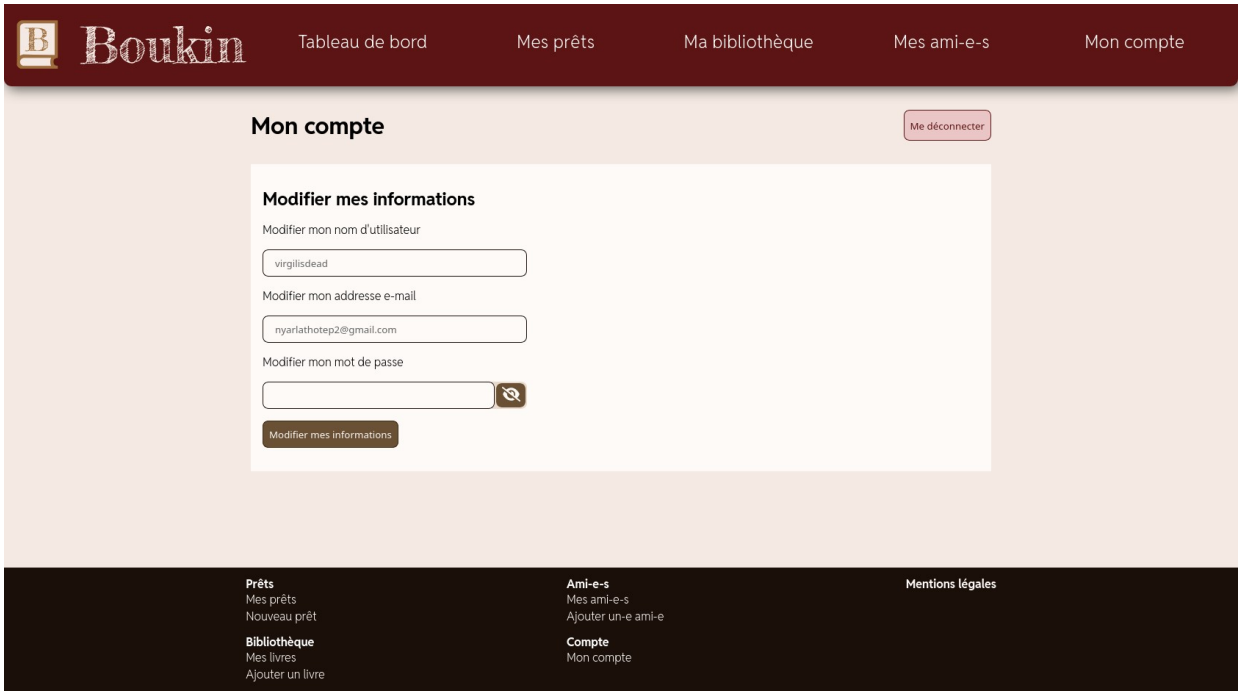


Figure 12: Page de profil d'un utilisateur de Boukin

Gestion des contacts

Après inscription et authentification, une personne doit pouvoir ajouter des contacts. Ces autres personnes, inscrites sur *Boukin*, constituent les amitiés de l'utilisateur-ice. La page "Mes ami-e-s" permet ainsi la gestion de ces amitiés. Plusieurs éléments sont disponibles sur cette page et permettent plusieurs fonctionnalités :

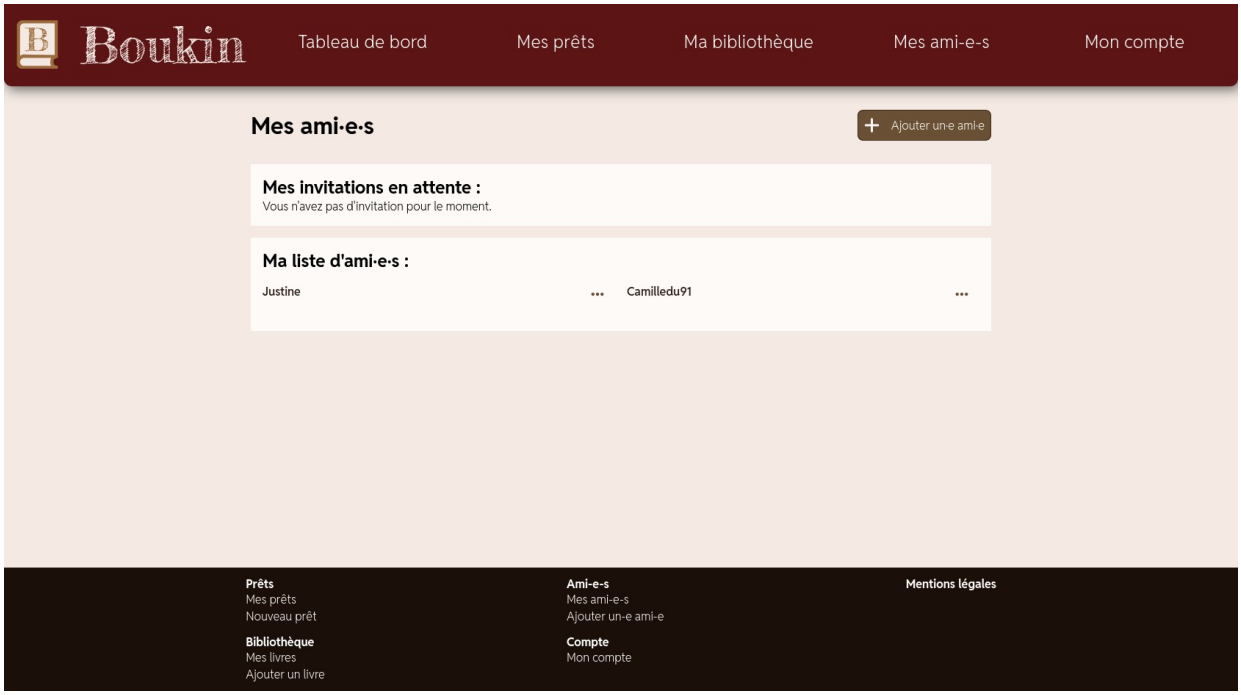


Figure 13: Page des amitiés d'un utilisateur de Boukin

- **Ajouter un-e ami-e** : un bouton est visible qui permet d'ajouter un contact. Lorsque l'on clique sur ce bouton, on arrive sur une autre page, qui permet de rechercher grâce à une adresse email une personne que l'on connaît. Si la recherche effectuée renvoie un résultat (signifiant que la personne en question est bien inscrite sur *Boukin*), une demande d'amitié peut alors être envoyée auprès de la personne retrouvée. Après envoi de la demande, il faut que cette personne l'accepte pour que l'amitié soit effectivement créée entre les deux utilisateur-ices.
- **Lister les demandes d'amitié reçues** : sur la page "Mes ami-e-s", la liste des demandes d'amitié reçues est visible. Cet ensemble de notifications permet ainsi de visualiser ces requêtes en attente. Lorsque l'on clique sur l'une de celles-ci, une modale avec les informations résumées de la personne demandant la mise en contact sont présentées. Deux boutons, permettant d'accepter ou de refuser la demande, sont présents. Si l'utilisateur-ice accepte la demande, l'amitié est créée et est ensuite visible. Si elle est refusée, l'amitié n'est pas créée ; seule la personne ayant refusé l'amitié peut modifier cela.
- **Lister les amitiés** : enfin, cette page liste l'ensemble des amitiés en cours de l'utilisateur-ice. En cliquant sur le bouton adéquat, il est possible de retirer la personne de la liste des amitiés ; une modale apparaît pour confirmer cette action. Si l'on clique sur le nom de l'ami-e, la navigation est faite vers une autre page listant la bibliothèque des livres de cette personne. Grâce à cette page, il est alors possible d'envoyer une demande d'emprunt d'un livre si celui-ci est marqué comme disponible. Une demande d'emprunt doit être confirmée par l'ami-e avant que le prêt soit considéré comme actif.

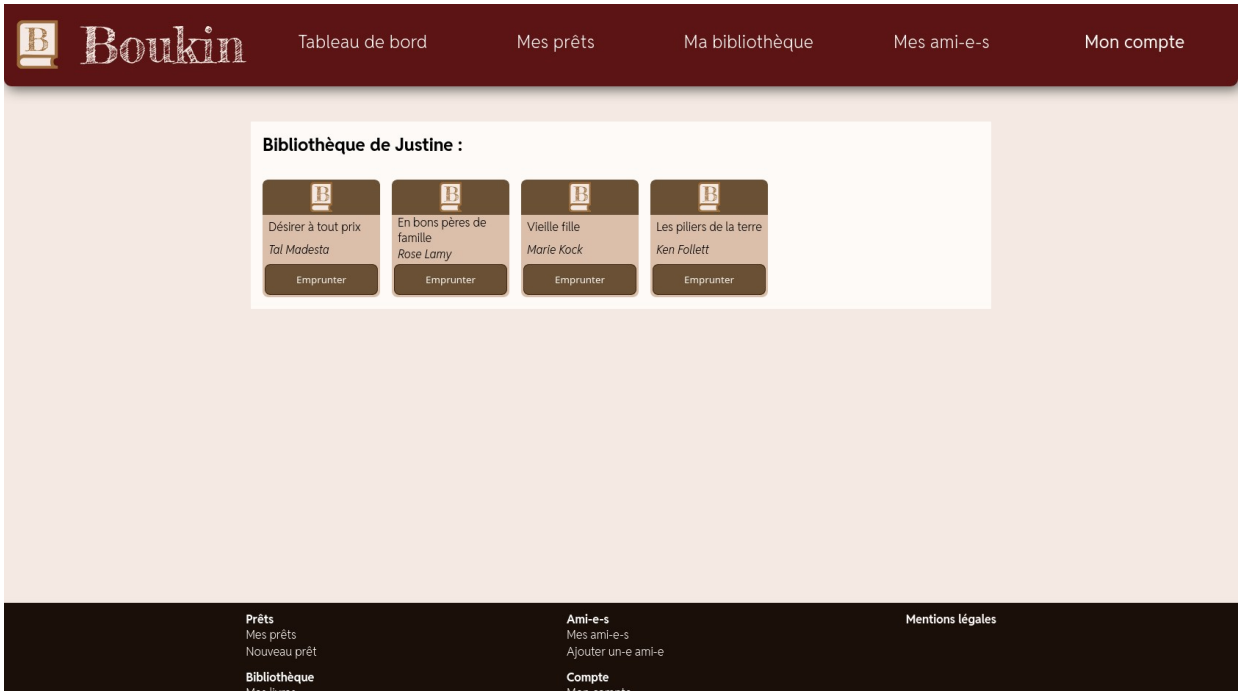


Figure 14: Bibliothèque d'une amie d'un utilisateur de Boukin

Gestion des livres

Un-e utilisateur-ice doit pouvoir ajouter des livres dans sa bibliothèque et choisir lesquels de ces livres sont visibles à son réseau d'ami-e-s. Elle doit pouvoir rechercher un livre grâce à la fonctionnalité correspondante de *Boukin*. Il doit être possible de filtrer simplement sa bibliothèque afin de faciliter la consultation de celle-ci, entre livres prêtés, disponibles, et cachés au réseau de contacts.

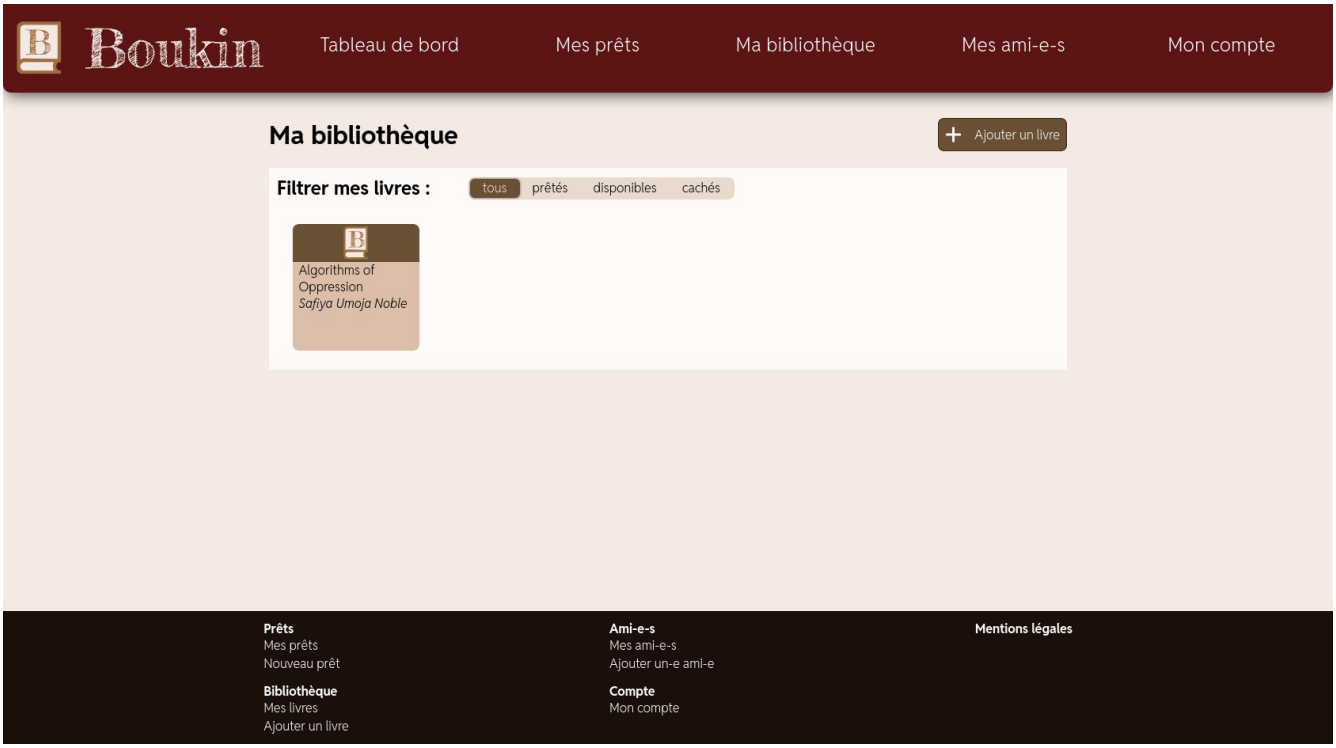


Figure 15: Page de la bibliothèque d'un utilisateur de Boukin

- **Ajouter un livre** : sur la page "Ma bibliothèque", un bouton est visible permettant la navigation vers une autre page, permettant la recherche d'un livre selon trois critères différents : par *International Standard Book Number* (ISBN, un identifiant unique pour l'édition d'un livre), par titre, par nom d'auteur-ice. Une liste de résultats apparaît si des livres correspondent au critère de recherche. Un bouton permet l'ajout du livre choisi à la bibliothèque personnelle. Si aucun résultat n'apparaît, il est possible d'ajouter manuellement une référence bibliographique, la rendant par la suite disponible pour les autres utilisateur-ices. Un formulaire doit être rempli, avec l'ISBN, le titre du livre et les nom et prénom de son auteur-ice. Si le livre n'est pas déjà enregistré en base de données, alors il est ajouté à celle-ci.
- **Afficher la bibliothèque** : chaque livre de la bibliothèque personnelle est listé sur la page "Ma bibliothèque". Chacune des cartes comporte les informations bibliographiques essentielles : Titre, nom de l'auteur-ice. Par défaut, tous les livres sont affichés sur la page. Il est possible de filtrer les livres selon plusieurs statuts, en cliquant sur le bouton correspondant : "prêté", "disponible", "caché". "Disponible" est le statut par défaut. "Caché" constitue le statut pour un livre ayant été ajouté à la bibliothèque, mais que l'utilisateur-ice ne veut pas rendre visible à son réseau de contacts afin d'éviter les demandes de prêts à son sujet. Enfin, tout livre prêté à une autre personne reçoit automatiquement le statut de "prêté".

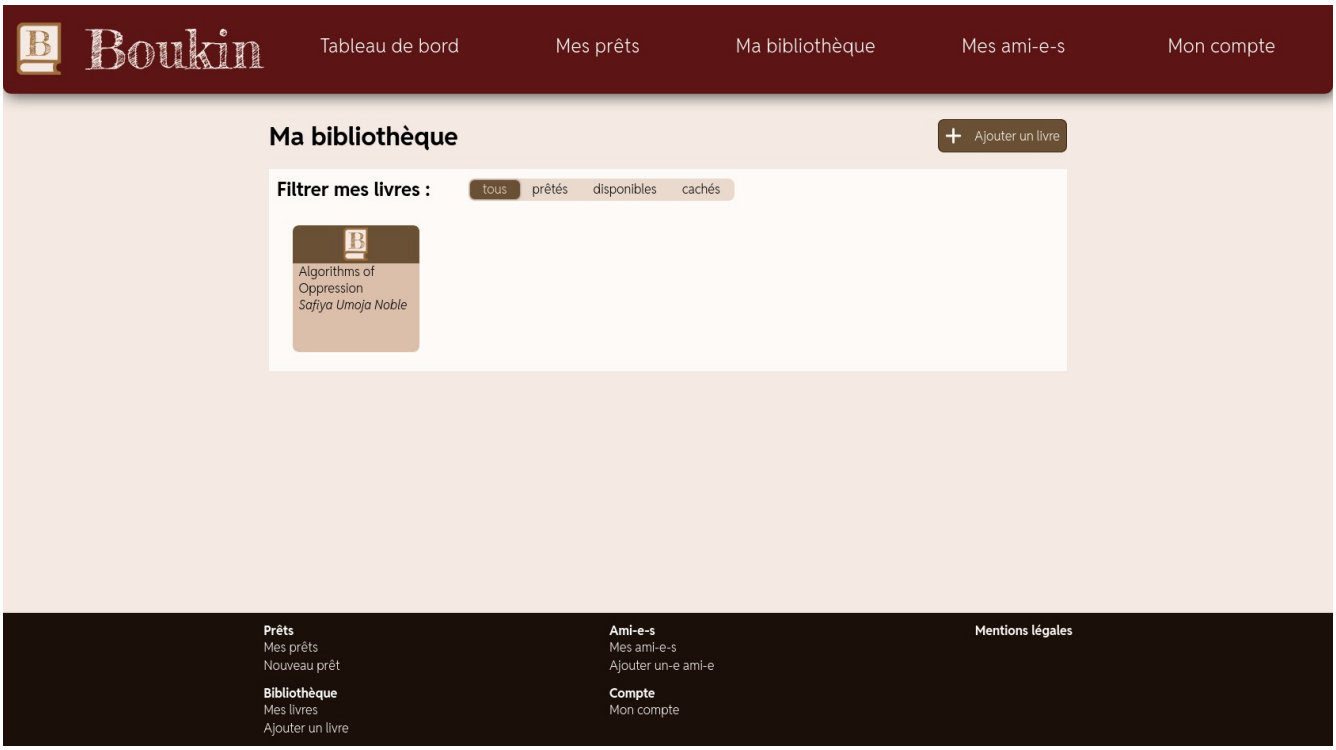


Figure 16: Page de la bibliothèque d'un utilisateur de Boukin

Gestion des prêts

La page "Mes prêts" rassemble les fonctionnalités de gestion des prêts et emprunts de livres auprès du réseau d'ami-e-s de l'utilisateur-ice. Il doit pouvoir être possible de visualiser les prêts / emprunts en cours, ainsi que les notifications concernant des demandes de prêts, les demandes de retours de livres empruntés à un-e ami-e, et la confirmation du retour d'un livre auparavant prêté à un-e ami-e. Enfin, il doit pouvoir être possible de prêter un livre de sa bibliothèque personnelle à un-e ami-e de son choix.

- **Visualiser les prêts / emprunts en cours** : si l'utilisateur-ice a prêté un livre ou a emprunté un livre, chacun de ces prêts est listé sur la section correspondante de la page, titrée "Prêts en cours". Les informations essentielles à ce sujet sont directement visibles : le nom et titre du livre, la personne a qui l'on a prêté ou emprunté le livre. S'il s'agit d'un prêt à un-e ami-e, un bouton est cliquable pour pouvoir effectuer une demande de retour du livre ; une modale apparaît pour confirmer cette demande. S'il s'agit d'un emprunt auprès d'un-e ami-e, un bouton est cliquable pour pouvoir notifier le retour du livre à cette personne.

- **Visualiser les différentes notifications** : pour les autres cas, une section “Notifications de prêts” permet de lister les notifications concernant des actions à effectuer par l’utilisateur-ice. Il s’agit des demandes de retour d’un livre ; d’une demande de prêt par un-e ami-e ; d’une demande de confirmation de retour. Selon le cas à traiter, un bouton peut être cliquable pour effectuer l’action demandée. S’il s’agit d’une demande de prêt, une modale apparaît pour accepter ou refuser la demande, et spécifier une date de retour (optionnel). S’il s’agit d’une confirmation de retour, une modale apparaît pour confirmer le retour du livre auparavant prêté. S’il s’agit d’une demande de retour, l’action peut être effectuée dans la section “Prêts en cours” de la page.
- **Créer un prêt** : un bouton est visible sur la page, permettant de naviguer vers une autre page, où un formulaire peut être rempli pour prêter un livre à un-e ami-e. Les champs disponibles sont la liste des livres de la bibliothèque personnelle effectivement disponibles, la liste des ami-es, et enfin deux champs pour les dates de début et de fin du prêt (optionnels). Après avoir rempli ces champs et cliqué sur le bouton correspondant, le prêt est créé et le livre est considéré comme étant prêté à l’ami-e choisi-e.

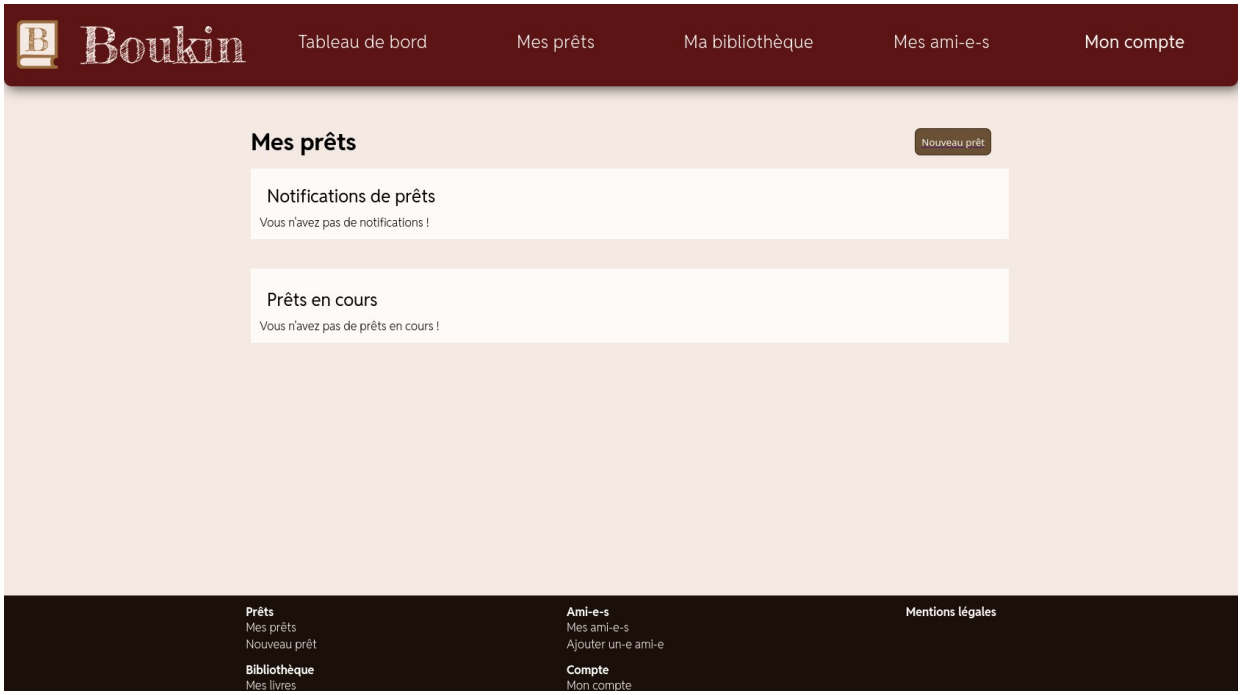


Figure 17: Page de gestion des prêts d’un utilisateur de Boukin

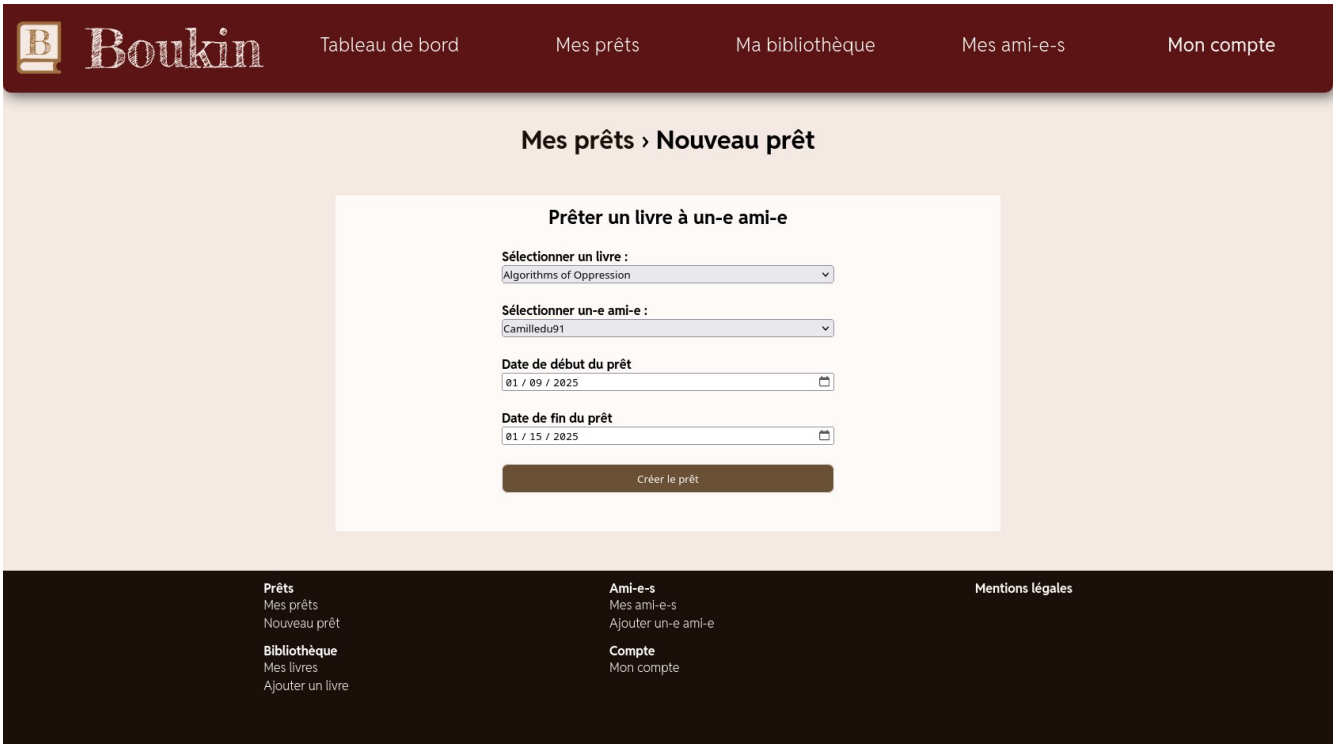
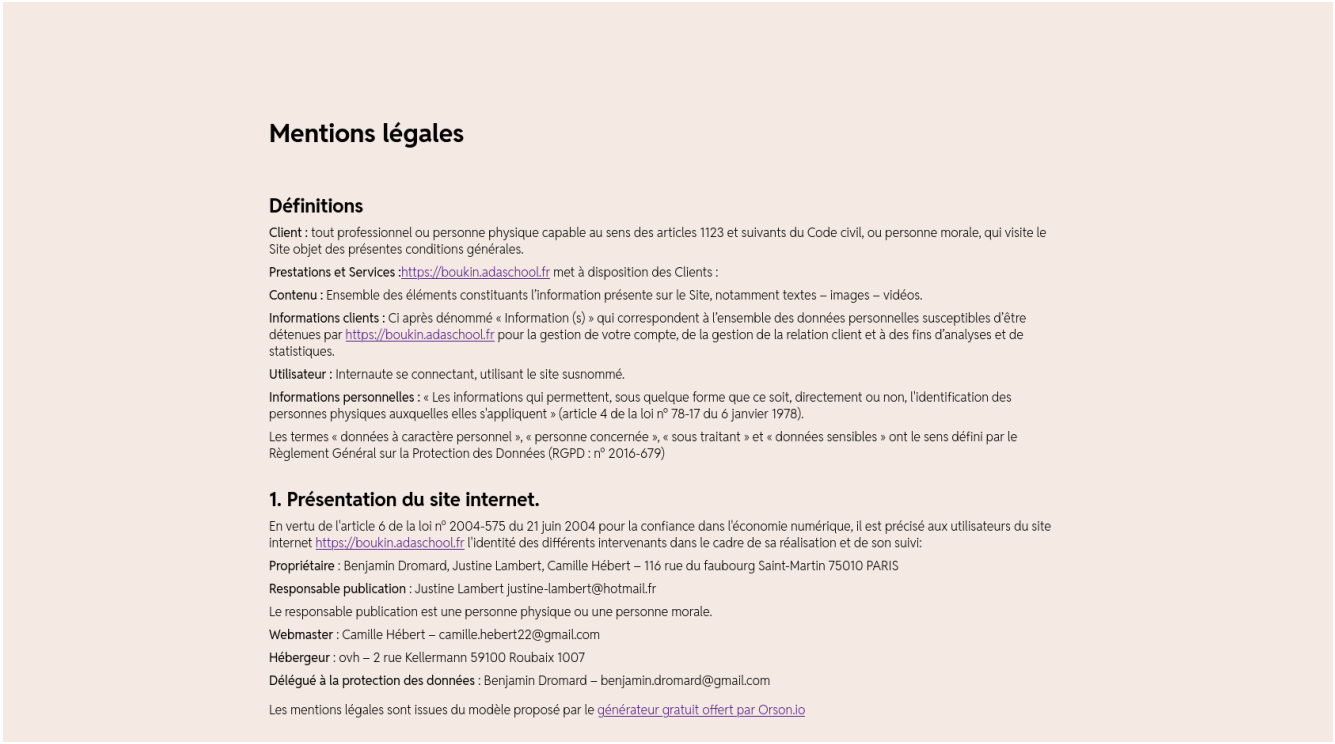


Figure 18: Formulaire de création de prêt

Mentions légales

Il est nécessaire de présenter à l'utilisateur-ice les informations concernant la structure responsable légalement de l'application *Boukin*. Une page "Mentions légales" est donc disponible sur le site, que l'on soit authentifié-e ou non. Cette page rassemble ainsi les conditions générales d'utilisation de *Boukin*, ainsi que les politiques liées à l'utilisation des *cookies* et à la conservation des données personnelles, conformément au *Réglement général sur la protection des données*, élément de législation européenne implémenté comme loi en France. Tout service numérique se devant de respecter les dispositions légales de cette loi, il est utile de présenter ce dont *Boukin* est responsable et les choix faits à ce sujet.

Ainsi, seul un *cookie* d'authentification est utilisé, nécessaire pour la récupération des données liées aux livres, ami-e-s et prêts. Etant un cookie nécessaire au fonctionnement du site, le consentement de l'utilisateur-ice n'est pas à obtenir (voir à ce sujet le référentiel de la *Commission Nationale de l'Informatique et des Libertés* : https://www.cnil.fr/fr/cookies-et-autres-traceurs/regles/cookies/que-dit-la-loi). Un nombre limité de données personnelles et sensibles est conservé par l'application. Il s'agit essentiellement de l'adresse email de l'utilisateur-ice. Après trois ans à compter de la dernière connexion à *Boukin*, ces informations doivent être supprimées de la base de données.



Mentions légales

Définitions

Client : tout professionnel ou personne physique capable au sens des articles 1123 et suivants du Code civil, ou personne morale, qui visite le Site objet des présentes conditions générales.

Prestations et Services :https://boukin.adaschool.fr met à disposition des Clients :

Contenu : Ensemble des éléments constituants l'information présente sur le Site, notamment textes – images – vidéos.

Informations clients : CI après dénommé « Information (s) » qui correspondent à l'ensemble des données personnelles susceptibles d'être détenues par https://boukin.adaschool.fr pour la gestion de votre compte, de la gestion de la relation client et à des fins d'analyses et de statistiques.

Utilisateur : Internaute se connectant, utilisant le site susnommé.

Informations personnelles : « Les informations qui permettent, sous quelque forme que ce soit, directement ou non, l'identification des personnes physiques auxquelles elles s'appliquent » (article 4 de la loi n° 78-17 du 6 janvier 1978).

Les termes « données à caractère personnel », « personne concernée », « sous traitant » et « données sensibles » ont le sens défini par le Règlement Général sur la Protection des Données (RGPD : n° 2016-679)

1. Présentation du site internet.

En vertu de l'article 6 de la loi n° 2004-575 du 21 juin 2004 pour la confiance dans l'économie numérique, il est précisé aux utilisateurs du site internet https://boukin.adaschool.fr l'identité des différents intervenants dans le cadre de sa réalisation et de son suivi:

Propriétaire : Benjamin Dromard, Justine Lambert, Camille Hébert – 116 rue du faubourg Saint-Martin 75010 PARIS

Responsable publication : Justine Lambert justine-lambert@hotmail.fr

Le responsable publication est une personne physique ou une personne morale.

Webmaster : Camille Hébert – camille.hebert22@gmail.com

Hébergeur : ovh – 2 rue Kellermann 59100 Roubaix 1007

Délégué à la protection des données : Benjamin Dromard – benjamin.dromard@gmail.com

Les mentions légales sont issues du modèle proposé par le générateur gratuit offert par Orson.io

Figure 19: Page des mentions légales de Boukin

Spécifications techniques du projet

Après avoir décrit les fonctionnalités de l'application constituant le *Minimum Viable Project* pour *Boukin*, je vais m'attarder désormais sur les spécifications techniques du projet. Pour les différentes couches de l'application, nous avons dû faire des choix d'outils et d'implémentations pour la réalisation de l'application. Avant cela, il est nécessaire de présenter la méthode de travail que nous avons adoptée pour le développement proprement dit.

Méthode de travail : Test-driven development

Dès le début des discussions quant à l'implémentation des fonctionnalités, que ce soit côté *front-end* comme côté *back-end*, et en parallèle des choix techniques à effectuer en matière de *stack* technique, nous nous sommes mis d'accord sur la volonté de développer en adoptant la méthode de *test-driven development*.

Il s'agit d'une méthode de travail où l'écriture des tests unitaires et d'intégration précédent l'implémentation du code applicatif. Elle peut être résumée de la manière suivante : suivant une spécification préalable pour la réalisation d'une fonctionnalité attendue de l'application, un test est rédigé. Un test comporte le contexte d'exécution, ses conditions et les assertions permettant de considérer que le code testé répond effectivement aux demandes du test. Après exécution du test, qui doit logiquement échouer du fait de l'absence d'implémentation, il est possible d'écrire le code applicatif. Après écriture du code et exécution du test, si ce dernier est concluant, alors il est possible d'améliorer le code en termes de lisibilité ou d'optimisation si besoin est (on parle de *refactoring*).

A mesure qu'une fonctionnalité, un composant, une page, etc., sont développées, plusieurs tests peuvent être rassemblés ensemble. Cela constitue une suite de tests. L'optimisation du code, possible si les tests réussissent, doit permettre d'enlever les répétitions de code, la simplification du code, etc.

Quels sont les avantages d'une telle méthode de travail ? Cela permet de s'assurer d'avoir du code fonctionnel, sans avoir à exécuter l'application dans son ensemble. C'est le but des tests en soi, mais en plaçant la rédaction des tests en première étape, il est possible de construire une application sans jamais avoir à l'exécuter avant l'étape des tests *end-to-end*. De cette manière, il est possible de se concentrer sur la rédaction de code applicatif fonctionnel, et de ne pas avoir à vérifier manuellement à chaque étape que telle ou telle fonctionnalité de l'application soit effectivement utilisable.

Le *test-driven development* permet aussi de s'assurer que chaque nouvel élément de code ne va pas empêcher l'exécution du reste de l'application ou créer de nouveaux *bugs*. Il est nécessaire de constamment exécuter l'ensemble des tests à partir du moment où de nouveaux éléments de code testé sont ajoutés. Ainsi, on s'assure que des effets de bord ne soient pas introduits par le code applicatif produit.

Les tests rédigées avec une telle méthode de travail ont aussi un autre avantage : la documentation de code proprement dite. Chaque test, s'il est bien décrit par son nom, et s'il est clair dans sa rédaction, montre ce qui est attendu en entrée puis en sortie de l'exécution du test. On décrit ainsi la fonctionnalité

de chaque nouvelle méthode, du composant pour la suite de tests qui lui correspond, etc. A mesure que le dépôt de code croît, cette documentation se révèle précieuse si l'on doit revenir sur des éléments de code plus anciens.

Le *test-driven development* ne règle toutefois pas tous les problèmes de développement logiciel. L'ensemble de l'application, avec ses multiples couches, se doit de disposer d'un environnement de test, dit *end-to-end* ; cela dépasse le cadre du TDD, car les tests *end-to-end* sont généralement rédigés après que le code applicatif ait été écrit. Ceci peut aussi se faire "manuellement", c'est à dire avec les serveurs du front-end et du back-end en cours d'exécution, comme si l'on utilisait effectivement l'application. C'est à ce moment que la connexion fonctionnelle entre toutes les couches peut être vérifiée, que les fonctionnalités ne comportent pas de *bugs* dans un tel contexte, etc. Si du nouveau code applicatif doit être rédigé, ou des corrections doivent être faites, alors on peut revenir dans un cycle TDD proprement dit, en rédigeant en premier lieu les tests unitaires / d'intégration utiles à la résolution du problème.

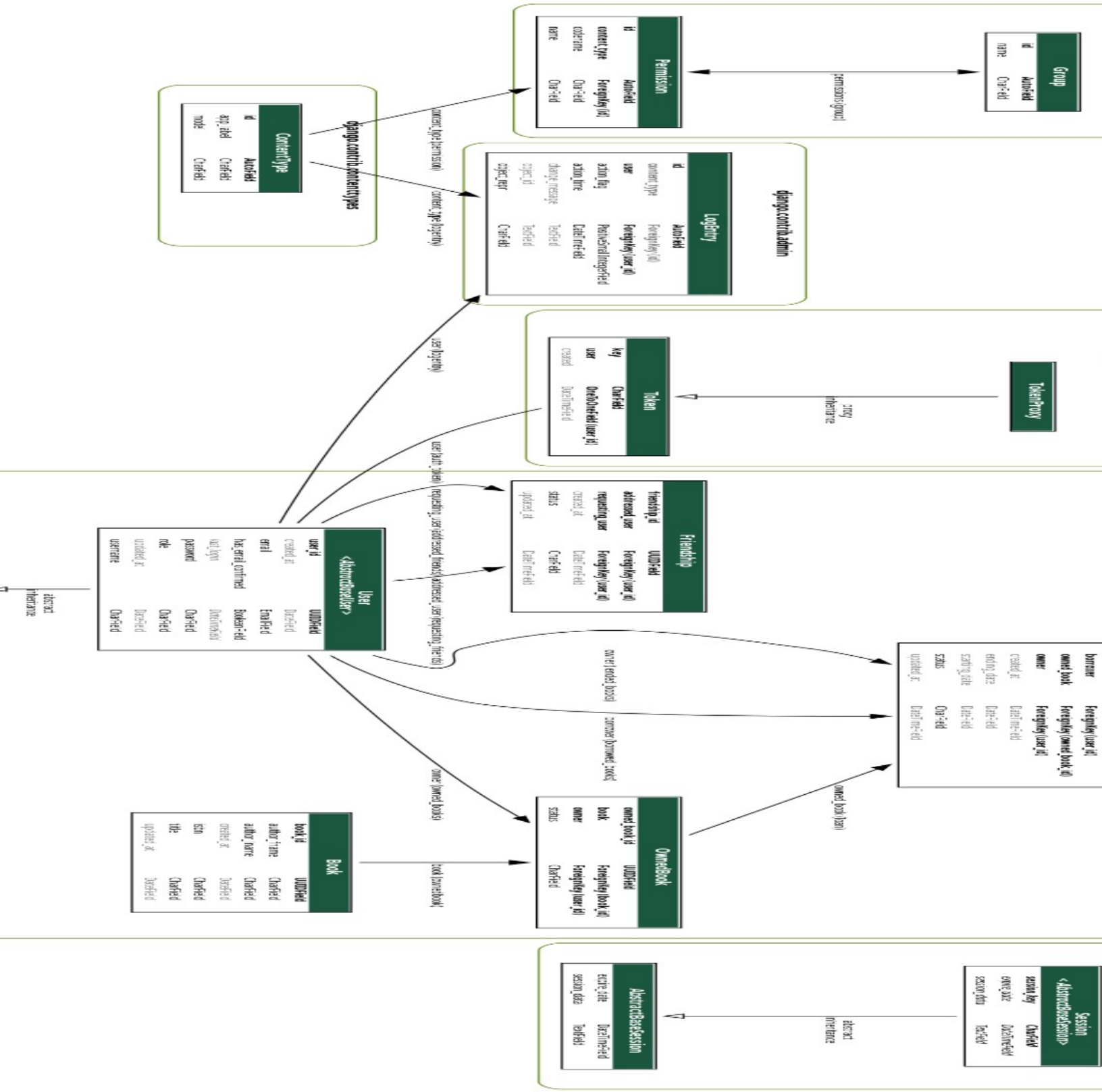
Enfin, le *test-driven development* peut poser problème si l'on ne connaît pas encore assez bien les outils techniques utilisés, que ce soit le langage de programmation, les *frameworks* applicatifs ou de test choisis, ou les bonnes pratiques de test propres à eux. Cela requiert une certaine période d'appropriation des outils en question. Toutefois, le temps investi vaut, de mon point de vue, largement le coût, du fait de l'assurance dans le code rédigé. Qui plus est, ces difficultés disparaissent progressivement avec la pratique.

Base de données

Le gestionnaire de base de données relationnelle choisi est **PostgreSQL**, utilisant le langage de requêtes *Structured Query Language*. Nous nous sommes portés sur ce choix du fait d'une certaine connaissance par certain-e-s d'entre nous de ce gestionnaire, de la large documentation disponible le concernant, ainsi que pour son caractère libre et open-source.

Du fait de l'utilisation de **Django** et **Django REST Framework**, que je présente plus bas, et de la logique *Object-relational Mapping* (ORM) implémentée directement avec **Django**, peu d'interactions directes avec la base de données ont eu lieu au cours du projet. **Django** demande quelques éléments de configuration pour la connexion à la base de données (mot de passe, nom de la base de données, port de connexion, etc.). Tout ce qui concerne la construction des tables, des colonnes et le typage de leurs entrées, etc., se fait par le biais de **Django**. Toutefois, la conception de la base de données doit être pensée en amont et avoir connaissance du fonctionnement d'une base de donnée relationnelle, ainsi que du typage utilisé par le moteur SQL de **PostgreSQL**.

Figure 20: Diagramme de classes de la dernière version de Boukin



bibliothèque personnelle. Un *OwnedBook* a un `status` : disponible, en cours de prêt ou caché aux autres utilisateur-ices.

- les *Loans* : les prêts de livres. On inscrit dans cette table la personne possédant le livre, celle l'empruntant, et enfin l'exemplaire du livre prêté, par le biais de relations par clés étrangères vers les tables *Users* et *OwnedBooks*. Un `status` est donné au prêt, pour distinguer une demande de prêt, son refus, un prêt en cours, et enfin les différents cas de figure pour enregistrer un retour de prêt. Il est possible de préciser des dates de début et de fin de prêt.

```
class UserManager(BaseUserManager):
    def create_user(self, email, username, has_email_confirmed, password):

        user = self.model(
            email=email,
            username=username,
            password=password,
            has_email_confirmed=has_email_confirmed,
        )
        user.is_superuser = False
        user.is_admin = False
        user.is_staff = False
        user.set_password(password)
        user.save()
        return user

    def create_superuser(self, email, password, **extra_fields):
        extra_fields.setdefault("is_superuser", True)
        if extra_fields.get("is_superuser") is not True:
            raise ValueError("Superuser must have is_superuser=True.")

        return self.create_user(email, password, **extra_fields)


class User(AbstractBaseUser):
    """
    user model
    """

    class UserRole(models.TextChoices):
        """
        user role class
        """

        USER = "user"
        ADMIN = "admin"
        OWNER = "owner"
        EXTERNAL = "external"

    objects = UserManager()
    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["username", "has_email_confirmed"]
    is_active = True
    has_email_confirmed = models.BooleanField(default=False)
```

```
    user_id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    username = models.CharField(max_length=30)
    email = models.EmailField(unique=True, null=True)
    password = models.CharField(max_length=255)
    role = models.CharField(choices=UserRole, default=UserRole.USER)
    created_at = models.DateField(auto_now_add=True)
    updated_at = models.DateField(auto_now=True)


class Book(models.Model):
    """
    book model
    """

    book_id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    title = models.CharField(max_length=255)
    author_fname = models.CharField(max_length=30)
    author_lname = models.CharField(max_length=30)
    isbn = models.CharField(max_length=13, unique=True)
    created_at = models.DateField(auto_now_add=True)
    updated_at = models.DateField(auto_now=True)


class OwnedBook(models.Model):
    """
    owned_book model
    """

    class OwnedBookStatus(models.TextChoices):
        """
        OwnedBook status class
        """

        AVAILABLE = "available"
        LOANED = "loaned"
        NOT_VISIBLE = "not visible"

    owned_book_id = models.UUIDField(
        primary_key=True, default=uuid.uuid4, editable=False
    )
    owner = models.ForeignKey(
        "User", on_delete=models.CASCADE, related_name="owned_books"
    )
    book = models.ForeignKey("Book", on_delete=models.PROTECT)
    status = models.CharField(
        choices=OwnedBookStatus, default=OwnedBookStatus.NOT_VISIBLE
    )


class Loan(models.Model):
    """
    loan model
    """
```

```
    class LoanStatus(models.TextChoices):
        """
        Loan status class
        """

        LOAN_REQUESTED = "loan_requested"
        LOAN_DENIED = "loan_denied"
        LOAN_ACTIVE = "loan_active"
        RETURN_REQUESTED = "return_requested"
        RETURN_CONFIRMATION_PENDING = "return_confirmation_pending"
        RETURN_CONFIRMED = "return_confirmed"

    loan_id = models.UUIDField(primary_key=True, default=uuid.uuid4,
editable=False)
    owner = models.ForeignKey(
        "User", on_delete=models.PROTECT, related_name="lended_books"
    )
    borrower = models.ForeignKey(
        "User", on_delete=models.PROTECT, related_name="borrowed_books"
    )
    owned_book = models.ForeignKey("OwnedBook", on_delete=models.CASCADE)
    status = models.CharField(choices=LoanStatus,
default=LoanStatus.LOAN_REQUESTED)
    starting_date = models.DateField(blank=True, null=True)
    ending_date = models.DateField(blank=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)


class Friendship(models.Model):
    """
    friendship model
    """

    class Meta:
        constraints = [
            models.UniqueConstraint(
                fields=["requesting_user", "addressed_user"],
                name="unique_friendship_couple",
            ),
        ]

    class FriendshipStatus(models.TextChoices):
        """
        Friendship status class
        """

        FRIENDSHIP_REQUESTED = "friendship_requested"
        FRIENDSHIP_DENIED = "friendship_denied"
        FRIENDSHIP_ACTIVE = "friendship_active"
        FRIENDSHIP_OVER = "friendship_over"

    friendship_id = models.UUIDField(
        primary_key=True, default=uuid.uuid4, editable=False
    )
    requesting_user = models.ForeignKey(
```

```
        "User", on_delete=models.PROTECT, related_name="addressed_friends"
    )
    addressed_user = models.ForeignKey(
        "User", on_delete=models.PROTECT, related_name="requesting_friends"
    )
    status = models.CharField(
        choices=FriendshipStatus, default=FriendshipStatus.FRIENDSHIP_REQUESTED
    )
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

Back-end

Django / Django REST Framework

Pour le développement de l'*Application Programming Interface* structurée selon la spécification *Representational State Transfer*, nécessaire à la récupération des données utilisées par la couche *front-end*, nous avons choisi le *framework* **Django** et son extension **Django REST Framework**.

De fait, **Django** est un framework web complet, utilisable avec le langage de programmation **Python**. Il permet la conception et le développement de sites web, aussi bien pour la couche *front-end* que la couche *back-end*. Toutefois, pour *Boukin*, nous utilisons **Django** essentiellement pour sa gestion d'une base de données par la technique *Object-relational Mapping*. Cela repose sur l'utilisation de classes et méthodes typiques du paradigme de la programmation orientée objet pour construire, communiquer avec et consulter une base de données relationnelle utilisant le langage de requêtes SQL.

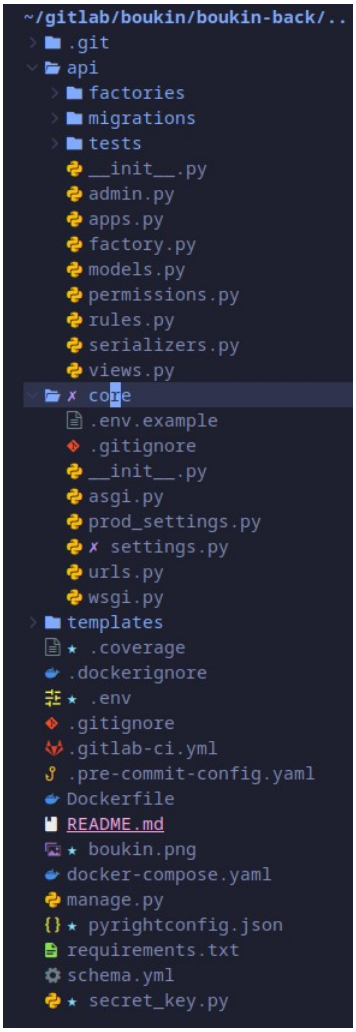


Figure 21: Structure du dépôt de la couche back-end de Boukin

Django demande ainsi un minimum d'éléments de configuration (gestionnaire de base de données relationnelle choisi, mode de connexion) pour pouvoir se connecter à une base de données. Tout le reste du code applicatif permettant d'architecturer et d'utiliser la base de données se fait par la suite par le biais du langage **Python**, sans avoir à rédiger du pur SQL. Cela permet d'abstraire de nombreux besoins dans le développement d'une application, comme ceux de sécurité et d'authentification. On délègue ainsi cette responsabilité essentiellement envers l'outil ORM de **Django**.

Django fonctionne essentiellement sur un modèle de développement associé à des *views*. Ces *views* sont des fonctions **Python** permettant de recevoir des requêtes HTTP, pour ensuite renvoyer une réponse par le protocole HTTP. Si l'on utilisait purement **Django**, cela permettrait de renvoyer des documents HTML / CSS en réponse à une requête vers une URL associée. Mais notre besoin est de développer une API qui renverrait, selon les paramètres HTTP envoyés, une structure de données au format JSON, lisible côté *front-end* par JavaScript et ensuite distribuée sur une page dans un navigateur. S'il est possible de faire cela avec seulement **Django**, le projet d'extension **Django REST**

Framework met à disposition un ensemble de classes et de méthodes qui permettent de structurer une API REST de manière intuitive, tout en disposant de l'outil ORM de **Django** pour tout ce qui concerne la communication avec la base de données.

Django REST Framework utilise bien des *views* pour analyser des requêtes HTTP associées à une URL. Le routeur de Django est l'outil qui associe URLs et *views*. La logique d'analyse se trouve ensuite dans le code de la *view* proprement dite.

```
# Le routeur associant URLs et views.

router = routers.DefaultRouter()

urlpatterns = [
    path("", include(router.urls)),
    path("api/schema/", SpectacularAPIView.as_view(), name="schema"),
    path(
        "api/swagger/",
        SpectacularSwaggerView.as_view(url_name="schema"),
        name="swagger-ui",
    ),
    path("api/redoc/", SpectacularRedocView.as_view(url_name="schema"),
name="redoc"),
    path("admin/", admin.site.urls),
    path("auth-token/", views.LoginUserView.as_view()),
    path("api-auth/", include("rest_framework.urls")),
    # User related routes:
    path("users/search/", views.GetUserView.as_view()),
    path("users/", views.CreateUserView.as_view()),
    path("users/<uuid:user_id>/", views.UpdateUserView.as_view()),
    path("users/<uuid:user_id>/loans/", views.GetAllLoansView.as_view()),
    path(
        "users/<uuid:user_id>/activation/",
        views.ActivateUserView.as_view(),
        name="user_activation",
    ),
    path("users/external/", views.CreateExternalUserView.as_view()),
    # Book related routes:
    path("books/", views.BooksView.as_view()),
    # Ownedbook related routes:
    path("owned-books/", views.OwnedBookView.as_view()),
    path("owned-books/<uuid:owned_book_id>/",
views.ModifyOwnedBookView.as_view()),
    path("users/<uuid:user_id>/owned-books/",
views.ListFriendOwnedBooksView.as_view()),
    path("loans/", views.CreateLoanByBookOwnerView.as_view()),
    path("loans/request/", views.RequestLoanByFriendView.as_view()),
    path("loans/<uuid:loan_id>/", views.UpdateLoanView.as_view()),
    # Friendship related routes:
    path("friendships/", views.CreateFriendshipView.as_view()),
    path("friendships/<uuid:friendship_id>/",
views.ModifyFriendshipView.as_view()),
    path("users/<uuid:user_id>/friendships/",
views.GetUserFriendshipsView.as_view()),
    path(
        "users/<uuid:user_id>/friendships/<uuid:friendship_id>/",
```

```
        views.GetUserSingleFriendshipView.as_view(),
    ),
]

urlpatterns += router.urls
```

C’est ensuite au niveau de la *view* que **Django REST Framework** vérifie l’authentification de la requête reçue. La plupart des points d’accès de l’API REST de *Boukin* nécessite une authentification, car ils permettent de retrouver les données personnelles d’utilisateur-ice-s de *Boukin*, ou celles de leurs ami-e-s. Nous avons choisi une authentification par *token* : à chaque utilisateur-ice dont l’inscription a été confirmée est associée un *token*, qui peut être obtenu avec la réception des bons identifiants (adresse email et mot de passe). Toute requête après authentification pour recevoir des données doivent avoir ce *token* dans l’en-tête HTTP `Authorization`.

A cela, **Django REST Framework** rajoute la fonctionnalité de *serializer* qui s’occupe de la conversion des données reçues en requête pour renvoyer, si la requête est valide, une réponse en JSON. Cette classe de *Serializer* peut donc associer des données au typage propre à Python, vers des données au typage propre à PostgreSQL, pour ensuite renvoyer des données formatées en JSON. C’est ce que l’on entend essentiellement par les opérations de sérialisation / désérialisation, que l’on peut retrouve dans d’autres langages de programmation ou *frameworks*, implémentée de cette manière par **Django REST Framework**.

Un *Serializer* associé à une *view* permet donc de choisir les données à inscrire ou à renvoyer, mises en relation avec les colonnes d’une table de la base de données. C’est aussi dans un *serializer* que l’on inscrit les règles de validation de données : par exemple, si l’on veut inscrire un mot de passe dans la colonne `password` pour un *User*, si la chaîne de caractères reçue ne convient pas aux règles établies à l’avance, alors celle-ci n’est pas convertie selon la fonction de dérivation cryptographique choisie par **Django**.

Grâce à ces deux outils, nous pouvions ainsi structurer notre API REST et disposer d’URLs vers lesquelles les requêtes du *front-end* sont reçues puis interprétées. Pour chaque table de la base de données, plusieurs méthodes HTTP peuvent être associées à la *view* correspondante : typiquement, il est possible de récupérer les données d’un-e utilisateur-ice avec la méthode `GET` placée en en-tête HTTP sur l’URI `/users/<user_id>`, ou créer un-e nouveau-lle utilisateur-ice avec la méthode `POST` sur l’URI `/users`. Pour mettre à jour certaines données de l’utilisateur-ice, ce serait la méthode `PATCH` par requête sur l’URI `/users/<user_id`.

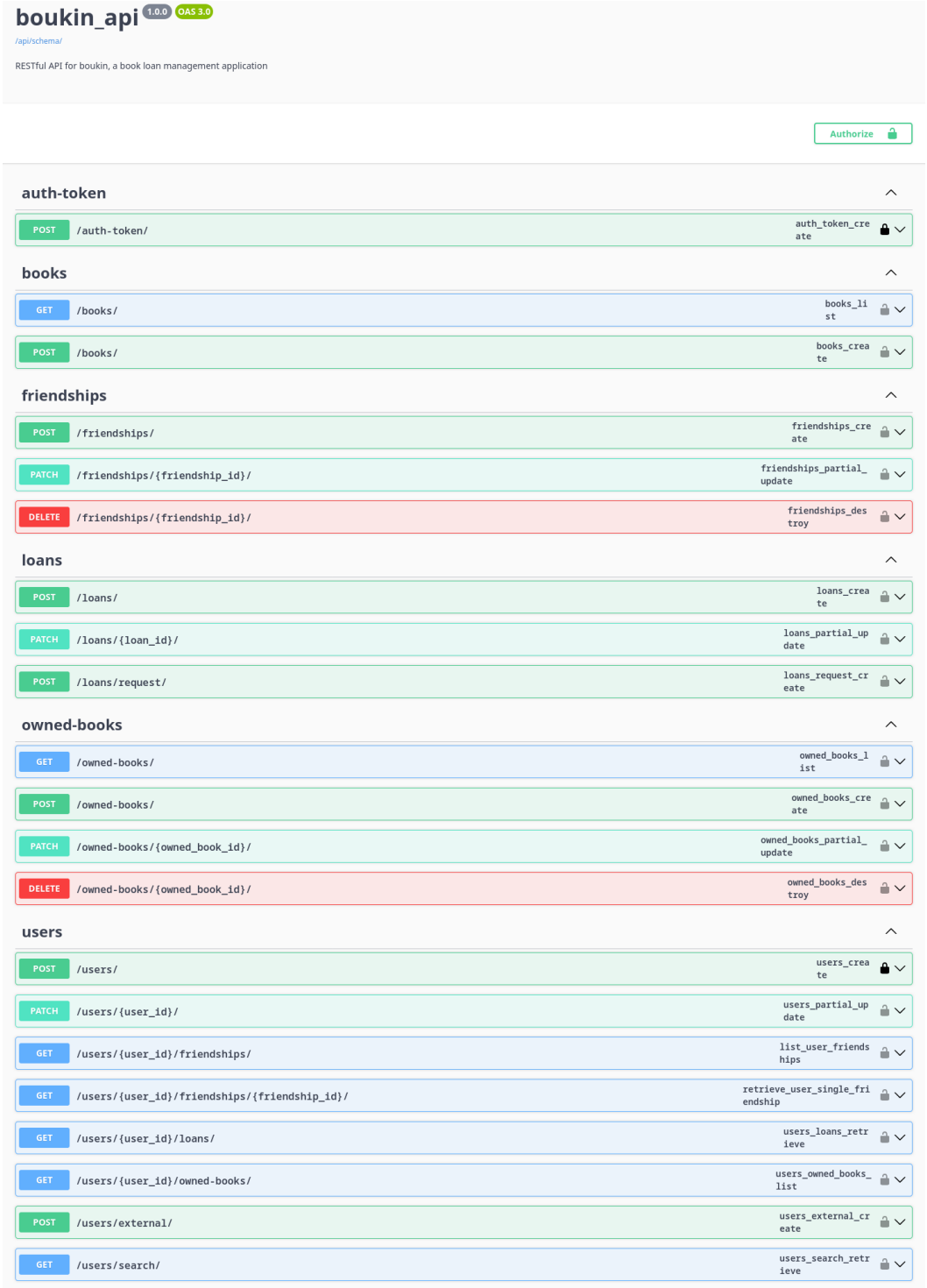


Figure 22: Schéma OpenAPI de l'API REST de Boukin, visualisable par Swagger

Enfin, mentionnons les outils liés aux tests pour le travail consacré au développement de l'API REST. Un des grands avantages de Python, et par extension de **Django** et **Django REST Framework** est l'incorporation de modules de rédaction de tests, sans installation de dépendances supplémentaires.

Pour l'écriture de tests unitaires et d'intégration, le module *unittest* de Python met à disposition un ensemble de classes et méthodes assez conséquent. Du côté de **Django** et **Django REST Framework**, ces deux outils construisent leurs méthodes et classes de test à partir d'*unittest*. Cela permet de conserver une syntaxe commune, que ce soit pour le code applicatif comme pour les lignes de commande à utiliser pour l'exécution des tests.

Plusieurs classes de test se révèlent de première importance pour développer l'API REST de *Boukin* : `TestCase` de **Django**, qui permet la création d'une méthode `setUp` où peuvent être créées des propriétés de classe utiles pour l'ensemble d'une suite de tests ; `APIRequestFactory` de **Django REST Framework**, qui permet de simuler le comportement de requêtes HTTP dans un environnement de test ; et enfin `APITestCase` de **Django REST Framework**, utile dans les cas où l'on veut tester précisément les méthodes d'authentification pour la requête HTTP testée.

`TestCase` est la plus utilisée dans notre dépôt de code, car plus rapide dans son exécution et permettant de tester essentiellement la logique à implémenter dans le code applicatif (validation des données, cas d'erreur hors erreur d'authentification, données renvoyées en réponse). `APITestCase` simule une pile réseau plus proche du comportement d'un client HTTP et d'un serveur ; de ce fait, cette classe est très utile si l'on veut tester l'authentification d'une requête et les erreurs associées, mais est plus lourde à exécuter. Un fichier de tests à part a été constitué pour ces cas de figure.

Front-end

Nuxt / Vue

Pour le développement de l'interface de l'application, notre choix s'est porté sur la bibliothèque JavaScript **Nuxt**, construit en extension du *framework* JavaScript **Vue**. Comme tout *framework* JavaScript, il permet la construction de pages et de composants en uniques fichiers, rassemblant **HTML** pour la structure du document, **CSS** pour le style, et **JavaScript** pour la récupération de données depuis l'API REST et la modification du *Document Object Model*.

Pour les portions de code en **JavaScript**, nous avons utilisé le langage **TypeScript**. Il ajoute notamment une vérification statique du typage des données utilisées. Cela signifie qu'après rédaction d'un fichier de code en **TypeScript**, le compilateur de **TypeScript** peut vérifier le typage des données et leur utilisation, pour pouvoir transpiler le code en **JavaScript**. S'il n'y a pas d'erreur de typage, le code peut donc être transpilé. Il est ensuite interprétable par un navigateur ou un environnement d'exécution **JavaScript** comme **Node**. Cela assure une plus grande sûreté du code rédigé avant son exécution et prévenir la création de *bugs* liés aux opérations entre données de types différents.

Nuxt / **Vue** mettent à disposition un ensemble de méthodes **JavaScript**, de pratiques quant à la structuration du dépôt de code, et d'éléments de configuration permettant le fonctionnement de l'application dans son ensemble. Du fait que plusieurs d'entre nous dans le groupe avions travaillé auparavant avec **Vue**, voire avec **Nuxt** en cadre professionnel, nous avions une certaine envie de

capitaliser sur ces connaissances pour la couche *front-end* du projet. **Nuxt** comporte aussi certains avantages permettant de faciliter le développement de *Boukin*.

Nuxt priorise le *Server-side rendering* dans sa logique d'exécution du code applicatif. Lorsque l'on navigue sur une page, comme par exemple "Mes prêts", le fichier de code correspondant sera exécuté dans un premier temps côté serveur : c'est le serveur qui génère le document HTML / CSS affichable dans le navigateur. Etant donné que la plupart des pages de *Boukin* effectuent des appels vers l'API REST de l'application, afin de récupérer toutes les données utiles pour l'utilisateur-ice comme ses prêts en cours, ces requêtes sont effectuées par le serveur afin de créer ce document. Ainsi, cela limite le code à exécuter dans le navigateur (côté client), ce qui a un avantage en termes d'éco-conception si l'on veut limiter la mobilisation des éléments matériels de l'ordinateur de l'utilisateur-ice pour afficher une page de l'application.

Nuxt permet aussi d'exécuter du code côté client, s'il y a besoin. A différents niveaux de l'application, des requêtes vers l'API de *Boukin* doivent être effectuées, afin de modifier ou supprimer des informations présentes en base de données. Le code JavaScript permettant cela doit être disponible et exécutable dans le navigateur dans ces cas de figure. Dans le cadre d'une application comme *Boukin*, avoir une certaine souplesse quant aux différents modes de rendus d'une page peut être avantageux. **Nuxt** met à disposition des méthodes, construites à partir de l'API `Fetch` de JavaScript, permettant de configurer les requêtes HTTP à effectuer vers l'API de manière précise.

Pour récupérer l'ensemble des prêts d'un-e utilisateur-ice, c'est la méthode `fetchLoans` qui est utilisée :

```
export const fetchLoans = async () => {
  const userCookie = useCookie<AuthenticationToken>("userCookie");
  const config = useRuntimeConfig();
  const loansEndpoint = `/users/${userCookie.value.user_id}/loans/`;
  const error = ref();
  const { data, error: _error } = await useFetch(loansEndpoint, {
    baseURL: config.public.apiBoukinBaseUrl,
    headers: {
      Authorization: `Token ${userCookie.value.token}`,
    },
  });
  if (data.value) {
    const totalNumberOfLoans = (data.value as LoansBody).total_number_of_loans;
    const loans = (data.value as LoansBody).loans;
    return { totalNumberOfLoans, loans };
  } else {
    error.value = _error.value;
    return { error };
  }
};
```

Ainsi invoquée côté serveur, pour la page "Mes prêts" :

```
<script setup lang="ts">

const { totalNumberOfLoans, loans, error: fetchError } = await fetchLoans();
```

```
</script>
```

Nuxt dispose aussi d'outils utiles pour la navigation entre les pages du site : le système de fichiers du dépôt de code correspond essentiellement à la navigation. Ainsi, dans le dossier `/pages`, il est possible de créer un dossier `/loans`, contenant un fichier `index.vue` pour la page principale lorsque l'on navigue vers `https://boukin.adaschool.fr/loans`. Si l'on ajoute un fichier `new.vue`, soit la page pour la création d'un nouveau prêt, alors il est possible de naviguer vers `https://boukin.adaschool.fr/loans/new` et voir la page correspondante, sans nouveau chargement de page. Cela permet de créer une navigation intuitive, rapide et ce de manière simple.

Pour ce qui concerne les outils de développement, **Nuxt** intègre *Vitest*, un environnement **JavaScript** d'exécution de test. Pour compléter cela, nous avons choisi d'ajouter les dépendances *testing-library* et *jest-dom*. La première est un *framework* de test qui met en avant les questions d'accessibilité pour l'écriture des contextes et des assertions des tests. La seconde rajoute des assertions utiles pour s'assurer de l'accessibilité des éléments créés dans les composants et les pages de l'application. La documentation de **Nuxt** conseille d'ailleurs l'utilisation de *testing-library* pour les tests des composants et pages.

Nuxt, par le biais de *Vitest*, facilite les tests d'intégration grâce à des méthodes pour le *mocking* d'utilisations de méthodes externes au code du composant ou de la page. Ainsi, il est possible de *mock* les requêtes vers l'API et les réponses de cette dernière. Pour la réalisation de tests, cela est primordial : des tests ne doivent pas effectuer de réelles requêtes vers un serveur externe. Cela compliquerait les tâches de développement qui devraient ainsi reposer sur l'exécution d'un service externe, potentiellement encore non fonctionnel.

De plus, pouvoir *mock* les réponses de cette API permet de gérer différents scénarios, notamment les cas d'erreur : si le *back-end* ne trouve pas de ressource associée à une URL vers laquelle une requête est faite, elle peut renvoyer une erreur de statut HTTP 404. La page à développer doit pouvoir avoir un comportement spécifique dans ce cas. De même, il est possible que des informations soient enregistrées dans un cookie ; l'environnement de test ne permet pas forcément d'effectuer cette opération, toutefois, il est possible de *mock* ce comportement et faciliter ainsi le développement, sans avoir à reposer sur une implémentation réelle d'une écriture dans un cookie.

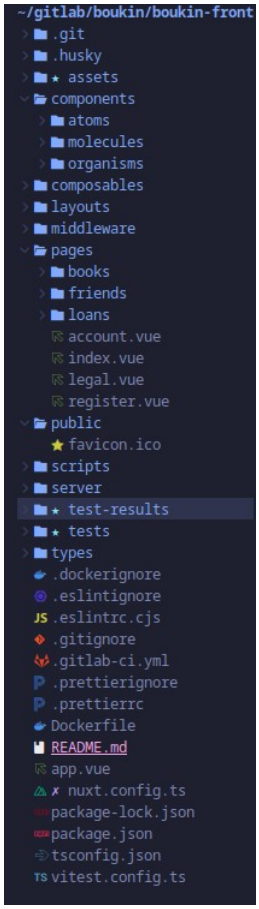


Figure 23: Structure du dépôt de la couche front-end de Boukin

De la manière suivante, on peut simuler la méthode effectuant une requête vers l'API REST de *Boukin*, et modifier facilement sa valeur de retour :

```
const twoLoans = require("./fixtures/loans/twoloans.json");
const userId = "3c48ecd4-d0bd-4171-a034-4444b768fbd8";
const mocks = vi.hoisted(() => {
  return { mockFetchLoans: vi.fn() };
});
registerEndpoint(`/users/${userId}/loans/`, mocks.mockFetchLoans);
mockNuxtImport("useCookie", () => {
  return () => {
    return {
      value: { email: "wowilovebooks@boukin.org", user_id: userId },
    };
```

```
  };
});
mocks.mockFetchLoans.mockReturnValue(twoLoans);
```

Avec `registerEndpoint`, on identifie l'URI utilisée dans le code applicatif pour associer la méthode simulée à celle-ci. Et avec `mockNuxtImport`, on peut créer la valeur de retour de la méthode de **Nuxt** `useCookie`, qui permet de donner une valeur à un cookie, la modifier ou simplement la consulter.

Réalisations personnelles

Je vais présenter dans cette section deux exemples de code que j'ai réalisés, l'un côté *back-end* (la mise à jour du statut d'un prêt) et l'autre côté *front-end* (la page de gestion des prêts et emprunts).

Pour l'écriture de code, j'utilise **NeoVim**, un *fork* de **Vim** qui permet une installation d'extensions, utiles notamment pour l'analyse de code statique, le formatage, la mise à disposition de la documentation concernant les classes et méthodes utilisées, etc. Cela rend possible l'édition de texte telle que configurée par **Vim** tout en disposant de fonctionnalités propres à d'autres *Integrated Development Environments* (IDE) comme **VSCode**. ## Exemple de fonctionnalité sur la couche back-end : la mise à jour du statut d'un prêt

Réalisation personnelle pour la couche back-end : la mise à jour du statut d'un prêt

Nous utilisions donc, pour la couche *back-end*, le langage de programmation **Python** et le *framework* **Django** allié de son extension **Django REST Framework**. Ce sont ces outils qui nous permettent de développer l'API REST, recevant des requêtes HTTP pour ensuite structurer les réponses au format JSON, renvoyées elles-aussi par le biais du protocole HTTP. Les données sont reçues après établissement d'une connexion et l'exécution d'une requête auprès de la base de données, possibles par le biais de **Django** permettant le développement en *Object-relational mapping* (ORM).

Au cours du développement de *Boukin*, nous nous sommes entendus pour que je travaille sur les fonctionnalités liées aux prêts des livres. Chaque prêt enregistré en base de données dispose d'une propriété `status`. Pour refléter les différentes possibilités quant à ceux-ci, un ensemble de statuts a été conçu, attribuables à chacun des prêts. Il s'agit des suivants :

- `loan_requested` : une demande de prêt de livre a été faite.
- `loan_denied` : la demande de prêt a été refusée par la / le propriétaire du livre.
- `loan_active` : la demande de prêt a été acceptée.
- `return_requested` : le / la propriétaire du livre a demandé le retour du livre.
- `return_confirmation_pending` : l'emprunteur-se du livre dit avoir rendu le livre.
- `return_confirmed` : le retour du livre a été confirmé par son / sa propriétaire.

Il existe donc plusieurs cas de figure où une mise à jour du statut du prêt, enregistré en base de données, doit pouvoir être faite. Ces possibilités sont différentes si la personne à l'origine de la mise à jour du statut est propriétaire du livre ou en est l'emprunteuse. Si l'on emprunte un livre, il n'est possible que de déclarer son rendu, si le prêt a été accepté. Si l'on possède un livre prêté, il y a d'avantage de possibilités, résumées dans le tableau suivant.

Mises à jour possibles par le / la propriétaire d'un livre :

Statut d'origine	**Statut(s) de mise à jour possibles**

LOAN_REQUESTED	LOAN_DENIED, LOAN_ACTIVE
LOAN_ACTIVE	RETURN_REQUESTED, RETURN_CONFIRMED
LOAN_DENIED	LOAN_ACTIVE
RETURN_REQUESTED	RETURN_CONFIRMED
RETURN_CONFIRMATION_PENDING	RETURN_CONFIRMED
RETURN_CONFIRMED	Aucun.

Mises à jour possibles par l'emprunteur-se d'un livre :

Statut d'origine	**Statut(s) de mise à jour possibles**
LOAN_REQUESTED	RETURN_CONFIRMATION_PENDING
LOAN_ACTIVE	Aucun.
LOAN_DENIED	RETURN_CONFIRMATION_PENDING
RETURN_REQUESTED	RETURN_CONFIRMATION_PENDING
RETURN_CONFIRMATION_PENDING	Aucun.
RETURN_CONFIRMED	Aucun.

Concrètement, cette mise à jour du statut d'un prêt se fait par une requête ayant la méthode `PATCH` en en-tête HTTP sur l'URI `/loans/<loan_id>`. C'est la *view* `UpdateLoanView` qui est chargée du traitement de la requête et de la réponse :

```
class UpdateLoanView(APIView):
    permission_classes = [permissions.IsAuthenticated, IsLoanOwnerOrBorrower]
    renderer_classes = [JSONRenderer]
    serializer_class = UpdateLoanSerializer

    @extend_schema(
        responses={204: UpdateLoanSerializer},
    )
    def patch(self, request, format=None, **kwargs):

        allowed_fields = {"status", "starting_date", "ending_date"}
        request_fields = set(request.data.keys())
        disallowed_fields = request_fields - allowed_fields

        if disallowed_fields:
            error_message = "Vous ne pouvez modifier que le statut du prêt"
            return Response(
                {"message": error_message}, status=status.HTTP_400_BAD_REQUEST
            )

        try:
```

```
            loan = Loan.objects.get(loan_id=kwargs["loan_id"])
        except Loan.DoesNotExist as error:
            return Response(error, status=status.HTTP_404_NOT_FOUND)

        self.check_object_permissions(request, loan)
        serializer = UpdateLoanSerializer(
            loan, context={"user": request.user}, data=request.data, partial=True
        )
        if serializer.is_valid():
            serializer.save()
            resource_location = (
                f'/users/{request.user.user_id}/loans/{kwargs["loan_id"]}/'
            )
            headers = {"Content-Location": resource_location}
            return Response(
                headers=headers,
                status=status.HTTP_204_NO_CONTENT,
            )

        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Le code applicatif présenté s'occupe ainsi de vérifier si la requête est recevable, selon plusieurs critères :

- si la requête HTTP reçue contient bien les bons identifiants, c'est-à-dire si la personne à l'origine de la requête est bien authentifiée. Concrètement, cela signifie que l'en-tête HTTP `Authorization` de la requête contient bien la méthode d'authentification et le token associé à l'identifiant de l'utilisateur-ice (exemple : `Authorization: Token 9944b09199c62bcf9418ad846dd0e4bbdfc6ee4b`), et donc que cette personne a bien dans son navigateur le cookie créé après authentification avec un email et un mot de passe valides et correspondants. Dans le cas contraire, une réponse HTTP de statut `401 Unauthorized` est envoyée ;
- si la personne à l'origine de la requête est bien identifiable comme étant l'emprunteur-se du livre ou son / sa propriétaire (`IsLoanOwnerOrBorrower`) ;
- si l'objet JSON reçu comporte seulement la clé `status` et éventuellement les dates de début et de fin de prêt (renvoie une réponse HTTP de statut 400 dans le cas contraire) ;
- si l'identifiant unique de prêt existe effectivement en base de données (renvoie une réponse HTTP de statut 404 dans le cas contraire).

Si elle est recevable, c'est ensuite la validation des données reçues dans la requête qui doit être effectuée, par le biais du *serializer* `UpdateLoanSerializer`. Si les données sont valides, la mise à jour de la donnée située à la colonne `status` pour l'entrée identifiable par l'UUID reçue est effectuée, et l'API renvoie une réponse HTTP de statut 204, avec en en-tête la localisation de la ressource modifiée.

Comment est faite la validation des données reçues en requête ? C'est le travail d'`UpdateLoanSerializer`.

```
class UpdateLoanSerializer(LoanSerializer):
    class Meta(LoanSerializer.Meta):
        model = Loan
```

```
        fields = ["status", "starting_date", "ending_date"]

    def validate(self, data):
        loan = self.instance
        user = self.context["user"]
        loan_rules = LoanRules(loan, data)

        if user == loan.owner:

            validation_error_message = loan_rules.check_owner_rules()
            if validation_error_message:
                raise ValidationError(_(validation_error_message))

        if user == loan.borrower:
            validation_error_message = loan_rules.check_borrower_rules()
            if validation_error_message:
                raise ValidationError(_(validation_error_message))

        return data
```

On établit la relation avec la table `Loan` et les colonnes dont les données sont à valider. Le *serializer* effectue une association entre l'objet Python reçu en argument (`self.instance`) et l'entrée dans la base de données. Il est aussi nécessaire de consulter les données concernant la personne faisant la requête (`self.context["user"]`). Puis, on consulte l'objet JSON reçu afin d'en valider le contenu avant la mise à jour de l'entrée en base de données (`data`). Il faut enfin consulter les règles de validation existantes pour savoir s'il est possible de faire passer le prêt d'un statut existant à celui reçu dans la requête : une instance de la classe `LoanRules` est créée pour cela, recevant en argument l'objet Python du prêt et l'objet JSON de la requête (`LoanRules(loan, data)`). Si un message d'erreur est renvoyée par l'instance de classe, cela signifie qu'il y a erreur de validation des données et que la demande de modification reçue en requête HTTP n'est pas recevable. Que se passe-t-il plus précisément dans la classe `LoanRules`?

Si la personne est identifiable comme la propriétaire du livre dans le prêt, alors il faut utiliser la méthode `check_owner_rules` de l'instance de classe `loan_rules`. Si elle est l'emprunteuse du livre, c'est la méthode `check_borrower_rules` qui est utilisée. Le code applicatif pour ces règles est le suivant :

```
class LoanRules:

    def __init__(self, loan, data):
        self.loan = loan
        self.data = data
        self._common_rules = self.__set_common_rules()
        self._owner_rules = self.__set_owner_rules()
        self._borrower_rules = self.__set_borrower_rules()

    def __check_key(self, key):
        try:
            value = self.data[key]
        except KeyError:
            return None
        else:
```

```
            return value

    @property
    def starting_date(self):
        return self.__check_key("starting_date")

    @property
    def ending_date(self):
        return self.__check_key("ending_date")

    @property
    def update_status_value(self):
        return self.__check_key("status")

    def __set_common_rules(self):

        self.return_confirmed_cannot_be_modified_rules = [
            self.loan.status == Loan.LoanStatus.RETURN_CONFIRMED
        ]

    def __set_owner_rules(self):
        self.loan_requested_must_become_active_or_denied_rules = [
            self.loan.status == Loan.LoanStatus.LOAN_REQUESTED,
            self.update_status_value != Loan.LoanStatus.LOAN_ACTIVE,
            self.update_status_value != Loan.LoanStatus.LOAN_DENIED,
        ]
        self.loan_active_must_become_return_requested_or_return_confirmed_rules =
[
            self.loan.status == Loan.LoanStatus.LOAN_ACTIVE,
            self.update_status_value != Loan.LoanStatus.RETURN_REQUESTED,
            self.update_status_value != Loan.LoanStatus.RETURN_CONFIRMED,
        ]
        self.loan_denied_can_become_active_rules = [
            self.loan.status == Loan.LoanStatus.LOAN_DENIED,
            self.update_status_value != Loan.LoanStatus.LOAN_ACTIVE,
        ]
        self.return_requested_must_become_confirmed_rules = [
            self.loan.status == Loan.LoanStatus.RETURN_REQUESTED,
            self.update_status_value != Loan.LoanStatus.RETURN_CONFIRMED,
        ]
        self.return_confirmation_pending_must_become_confirmed_rules = [
            self.loan.status == Loan.LoanStatus.RETURN_CONFIRMATION_PENDING,
            self.update_status_value != Loan.LoanStatus.RETURN_CONFIRMED,
        ]
        self.loan_dates_cannot_be_modified_outside_of_loan_request_confirmation =
[
            self.loan.status != Loan.LoanStatus.LOAN_REQUESTED,
            self.update_status_value != Loan.LoanStatus.LOAN_ACTIVE,
            any([self.starting_date is not None, self.ending_date is not None]),
        ]

    def __set_borrower_rules(self):

        self.loan_active_must_receive_return_confirmation_pending_rules = [
            self.loan.status == Loan.LoanStatus.LOAN_ACTIVE,
```

```
            self.update_status_value !=
Loan.LoanStatus.RETURN_CONFIRMATION_PENDING,
        ]
        self.loan_requested_cannot_be_modified_rules = [
            self.update_status_value is not None,
            self.loan.status == Loan.LoanStatus.LOAN_REQUESTED,
        ]

        self.loan_denied_cannot_be_modified_rules = [
            self.loan.status == Loan.LoanStatus.LOAN_DENIED
        ]
        self.return_requested_must_receive_return_confirmation_pending_rules = [
            self.loan.status == Loan.LoanStatus.RETURN_REQUESTED,
            self.update_status_value !=
Loan.LoanStatus.RETURN_CONFIRMATION_PENDING,
        ]
        self.return_confirmation_pending_cannot_be_modified_rules = [
            self.loan.status == Loan.LoanStatus.RETURN_CONFIRMATION_PENDING
        ]
        self.starting_and_ending_dates_cannot_be_modified_rules = [
            self.starting_date is not None,
            self.ending_date is not None,
        ]

    def check_owner_rules(self):

        validation_error_message = ""

        if
all(self.loan_dates_cannot_be_modified_outside_of_loan_request_confirmation):
            validation_error_message += (
                "Vous ne pouvez modifier les dates du prêt si vous l'avez
accepté.\n"
            )

        if all(self.loan_requested_must_become_active_or_denied_rules):
            validation_error_message += (
                "Vous devez accepter ou refuser cette demande de prêt.\n"
            )
        elif all(

self.loan_active_must_become_return_requested_or_return_confirmed_rules
        ):
            validation_error_message += (
                "Un livre en cours de prêt ne peut être demandé en prêt.\n"
            )
        elif all(self.loan_denied_can_become_active_rules):
            validation_error_message += "Vous avez refusé de prêter ce livre, vous
ne pouvez modifier le statut de prêt, sauf si vous voulez autoriser le prêt.\n"
        elif all(self.return_requested_must_become_confirmed_rules):
            validation_error_message += "Vous avez demandé le retour de ce livre,
vous ne pouvez modifier le statut de prêt, sauf si vous voulez confirmer le
retour.\n"
        elif all(self.return_confirmation_pending_must_become_confirmed_rules):
```

```
            validation_error_message += "Votre confirmation de retour de prêt a
été demandée, vous ne pouvez modifier le statut de prêt, sauf si vous voulez
confirmer le retour.\n"
        elif all(self.return_confirmed_cannot_be_modified_rules):
            validation_error_message += "Vous avez confirmé le retour de ce livre,
vous ne pouvez modifier le statut de prêt.\n"

        return validation_error_message

    def check_borrower_rules(self):

        validation_error_message = ""

        if any(self.starting_and_ending_dates_cannot_be_modified_rules):
            validation_error_message += (
                "Vous ne pouvez modifier les dates de début ou de fin de prêt.\n"
            )

        if all(self.loan_denied_cannot_be_modified_rules):
            validation_error_message += (
                "Ce prêt vous a été refusé, vous ne pouvez le modifier.\n"
            )
        elif all(self.loan_active_must_receive_return_confirmation_pending_rules):
            validation_error_message += "Ce prêt est en cours, vous ne pouvez en
modifier le statut, sauf si vous rendez le livre prêté.\n"
        elif all(self.loan_requested_cannot_be_modified_rules):
            validation_error_message += (
                "Vous avez demandé ce prêt, vous ne pouvez en modifier le statut.\
n"
            )
        elif
all(self.return_requested_must_receive_return_confirmation_pending_rules):
            validation_error_message += "Vous avez une demande de retour en cours,
vous ne pouvez en modifier le statut, sauf si vous rendez le livre prêté.\n"
        elif all(self.return_confirmation_pending_cannot_be_modified_rules):
            validation_error_message += "Votre retour de prêt a bien été
enregistré, il est en attente de confirmation. Vous ne pouvez en modifier le
statut.\n"
        elif all(self.return_confirmed_cannot_be_modified_rules):
            validation_error_message += "Votre retour de prêt a bien été confirmé.
Vous ne pouvez en modifier le statut.\n"

        return validation_error_message
```

La classe est chargée ainsi, selon les données du prêt et de la requête, d'associer les deux valeurs de statut et vérifier si la mise à jour de l'un vers l'autre est possible. Cela se fait à partir des propriétés `_common_rules`, `_owner_rules` et `_borrower_rules`, selon le cas de figure. Selon les conditions vérifiées, les méthodes de la classe vérifient si l'ensemble des conditions pour chacune des règles sont vraies ; si c'est le cas, alors il y a erreur de validation et la mise à jour du statut ne peut être effectué. Un message d'erreur précis est intégré dans la réponse à renvoyer dans la réponse de statut HTTP 400.

Disposer de messages d'erreur pour chaque cas de figure et la mise en place d'une telle logique de vérification facilite le développement et l'utilisation de l'API, notamment dans un cas de figure relativement complexe où plusieurs statuts de prêt existent et sont incompatibles les uns les autres

lorsqu'il faut mettre à jour un prêt. ## Exemple de fonctionnalité sur la couche front-end : la page de gestion des prêts et emprunts

Réalisation personnelle pour la couche front-end : la gestion des prêts et emprunts

Au cours du développement de *Boukin*, nous nous sommes entendus pour que je m'occupe essentiellement des fonctionnalités liées aux prêts des livres. La page "Mes prêts" et les fonctionnalités associées sont donc de mon fait. Pour montrer un exemple plus précis, je vais m'intéresser à la section "Prêts en cours" de cette page, qui permet de visualiser les livres prêtés et empruntés par l'utilisateur-ice.

La page "Mes prêts" constitue un fichier dans le dossier `/pages/loans` du dépôt de code, nommé `index.vue`. A partir des éléments de navigation, il est ainsi possible de cliquer sur un lien, comme `<a href="/loans">Mes prêts</a>`, et **Nuxt** implémente ainsi la navigation vers cette page.

Le fichier `index.vue`, comme le veut la logique du framework **Vue**, intègre du code JavaScript exécuté côté serveur entre les balises `<script setup> ... </script>`. Le code effectuant une requête vers l'API REST de *Boukin* et toute la logique applicative de gestion de l'état ce trouve dans cette section du fichier.

Tout ce qui concerne la structure du document à présenter dans le navigateur se situe entre les balises `<template> ... </template>`. C'est dans cette section qu'est rédigé le code HTML de structuration sémantique de la page. Le langage HTML utilisé est étendu par **Vue** et il est possible d'y intégrer de la logique applicative, de placer les données reçues en réponse de l'API, d'itérer sur ces données, etc.

Enfin, les balises `<style> ... </style>` contient la feuille de style CSS propre à la page. La feuille de style globale à l'ensemble du site se situe dans le dossier `/assets/css`, nommée `style.css`.

Pour ce qui concerne la section "Prêts en cours", le code du *template* est le suivant :

```
    <section>
      <div id="on-going-loans">
        <div id="top-ongoing-loans">
          <h2>Prêts en cours</h2>
        </div>
        <div v-if="hasErrorOccured">
          <p>Erreur de chargement des prêts : {{ fetchError }}</p>
        </div>
        <div v-else-if="hasActiveLoans" id="on-going-loans-list">
          <p v-if="numberOfActiveLoans === 1">Vous avez un prêt en cours.</p>
          <p v-else-if="numberOfActiveLoans > 1">
            Vous avez {{ numberOfActiveLoans }} prêts en cours.
          </p>
          <ul id="loan-lists" aria-label="Liste de prêts">
            <div v-for="loan in loansAsOwner" :key="loan.loan_id">
              <li>
                <em> {{ loan.owned_book.book.title }} </em>, prêté à
                {{ loan.borrower.username }}
              </li>
              <OrganismsLoanDetails
```

```
                :loan="loan"
                :loan-update-type="LoanUpdateType.AskingBookReturn"
              />
            </div>
            <div v-for="loan in loansAsBorrower" :key="loan.loan_id">
              <li>
                <em> {{ loan.owned_book.book.title }} </em>, emprunté à
                {{ loan.owner.username }}
              </li>
              <OrganismsLoanDetails
                :loan="loan"
                :loan-update-type="LoanUpdateType.ReturningLoan"
              />
            </div>
          </ul>
        </div>
        <div v-else-if="!hasActiveLoans">
          <p>Vous n'avez pas de prêts en cours !</p>
        </div>
      </div>
    </section>
```

Plusieurs des éléments de cette section ne sont affichés que de manière conditionnelle, essentiellement si une erreur a eu lieu lors de la récupération des données ou que l'utilisateur-ice n'a pas de prêts / emprunts en cours, ou selon le nombre de prêts / emprunts en cours. Si des données ont été récupérées, selon qu'il s'agisse de prêts ou d'emprunts, une liste non-ordonnée est constituée, et le code itère sur la structure de données reçue pour générer les éléments de liste adéquats, comportant les informations sur le prêt en question. A chaque élément de cette liste est associé un composant `OrganismsLoanDetails`, dont le fichier est situé dans le dossier `/organisms` au fichier `loan-details.vue`. Il contient essentiellement un bouton et la modale à afficher si l'utilisateur-ice clique dessus.

Pour afficher les données dans cette section, plusieurs variables existent et conditionnent l'affichage des éléments : `hasErrorOccured`, `hasActiveLoans`, `loansAsOwner`, `loansAsBorrower` et enfin `numberOfActiveLoans`. Les deux premières sont des booléens dont les valeurs sont déterminés selon certaines conditions :

```
const hasErrorOccured = computed(() => {
  if (fetchError?.value && fetchError.value.statusCode != 404) {
    return true;
  } else {
    return false;
  }
});

const hasActiveLoans = computed(() => {
  if (
    (fetchError?.value.statusCode === 404 || numberOfActiveLoans.value === 0) &&
    !hasErrorOccured.value
  ) {
    return false;
  } else {
    return true;
  }
});
```

La méthode `computed` de **Vue** est utile si l'on a besoin de modifier des éléments du DOM selon une logique nécessitant la consultation de différentes valeurs dans le reste du code applicatif. Pour `hasErrorOccured`, on vérifie s'il y a eu une erreur au cours de la récupération des données (`fetchError`), et que le statut HTTP de cette n'est pas 404 (qui indiquerait simplement l'absence de prêts / emprunts pour l'utilisateur-ice). Si c'est le cas, un message d'erreur est affiché.

Dans le cas où il n'y a pas eu d'erreur dans la communication avec l'API, on vérifie l'existence de prêts / emprunts en cours. Si ce n'est pas le cas, on vérifie aussi si les données récupérées concernent bien des prêts actifs, et non des demandes de prêts, de retours de livres, etc. Ces derniers sont affichés dans l'autre section de la page. Si `hasActiveLoans` est faux, alors on affiche un message indiquant qu'il n'y a pas de prêts / emprunts à afficher. S'il est vrai, alors on peut itérer sur les données reçues et afficher la liste des prêts.

Pour récupérer les données, et déterminer s'il s'agit de prêts (`loansAsOwner`) ou d'emprunts (`loansAsBorrower`), on filtre sur l'objet JSON reçu en réponse :

```
const loansAsOwner = computed(() => {
  const result = loans?.filter(
    (loan: Loan) =>
      loan.owner.user_id === userCookie.value.user_id &&
      loan.status === "loan_active",
  );
  return result || null;
});

const loansAsBorrower = computed(() => {
  const result = loans?.filter(
    (loan: Loan) =>
      loan.borrower.user_id === userCookie.value.user_id &&
      loan.status === "loan_active",
  );
  return result || null;
});
```

On filtre la structure de données rassemblant les prêts selon le statut `loan_active` et la valeur de l'identifiant `user_id` présente dans un cookie. Avec la clé `user_id` dans les objets `owner` et `borrower`, on peut déterminer s'il s'agit d'un prêt ou d'un emprunt. On obtient alors un tableau contenant les prêts filtrés, sur lequel il est possible d'itérer et de construire les éléments de liste dans le document. `numberOfActiveLoans`, qui permet d'afficher le nombre total de prêts / emprunts, constitue simplement la somme des tailles des deux tableaux :

```
const numberOfActiveLoans = computed(() => {
  const result =
    (loansAsOwner.value?.length || 0) + (loansAsBorrower.value?.length || 0);
  return result;
});
```

L'affichage des prêts et emprunts doit permettre une action : s'il s'agit d'un prêt, de demander le retour du livre prêté ; s'il s'agit d'un emprunt, de pouvoir rendre le livre. C'est dans le composant `LoanDetails` que l'on retrouve le code permettant cette fonctionnalité.

```
<script setup lang="ts">
import type { Loan } from "~/types/loan";
import { ButtonText, LoanUpdateType } from "../../types/loan";

interface Props {
  loan: Loan;
  loanUpdateType: LoanUpdateType;
}

const props = defineProps<Props>();
const isModalVisible = ref(false);
const startingDate = defineModel<String>("startingDate");
const endingDate = defineModel<String>("endingDate");
const dates = computed(() => {
  const starting_date = startingDate.value;
  const ending_date = endingDate.value;
  return { starting_date, ending_date };
});

const buttonText = computed(() => {
  switch (props.loanUpdateType) {
    case LoanUpdateType.LoanRequest:
      return ButtonText.LoanRequest;
    case LoanUpdateType.ReturningLoan:
      return ButtonText.ReturningLoan;
    case LoanUpdateType.AskingBookReturn:
      return ButtonText.AskingBookReturn;
    case LoanUpdateType.BookReturn:
      return ButtonText.BookReturn;
  }
});

function showModal() {
  isModalVisible.value = true;
}
function closeModal() {
  isModalVisible.value = false;
}
</script>

<template>
  <button class="primary-button" @click="showModal">
    {{ buttonText }}
  </button>
  <AtomsModal v-if="isModalVisible" @close-modal="closeModal">
    <form
      v-else-if="loanUpdateType == LoanUpdateType.AskingBookReturn"
      aria-label="Demande de retour de prêt"
    >
      <p>
        Vous souhaitez réclamer <em>{{ loan.owned_book.book.title }} </em> de
        {{ loan.owned_book.book.author_fname }}
        {{ loan.owned_book.book.author_lname }}, prêté à
        {{ loan.borrower.username }}
      </p>
      <p>Date de prêt: {{ loan.starting_date }}</p>
```

```
      <p>Date désirée de rendu: {{ loan.ending_date }}</p>
      <MoleculesLoanUpdate
        :loan-update-type="LoanUpdateType.AskingBookReturn"
        :loan-id="loan.loan_id"
      />
    </form>
    <form v-else aria-label="Déclaration de rendu de livre emprunté">
      <p>
        Vous souhaitez rendre <em> {{ loan.owned_book.book.title }} </em> de
        {{ loan.owned_book.book.author_fname }}
        {{ loan.owned_book.book.author_lname }}, emprunté à
        {{ loan.owner.username }}
      </p>
      <p>Date de prêt: {{ loan.starting_date }}</p>
      <p>Date désirée de rendu: {{ loan.ending_date }}</p>
      <MoleculesLoanUpdate
        :loan-update-type="LoanUpdateType.ReturningLoan"
        :loan-id="loan.loan_id"
      />
    </form>
  </AtomsModal>
```

Ce composant permet aussi la gestion des cas de figure présentés dans la section “Notifications de prêts” de la page “Mes prêts”. Il comporte un bouton, qui permet l’affichage d’une modale avec l’exécution de la fonction `showModal`. La modale comporte un bouton pour la fermer, et surtout un formulaire, résumant les données connues du prêt, puis un autre composant `MoleculesLoanUpdate`, rédigé dans le fichier `loan-update.vue` du dossier `components/molecules`. C’est dans ce composant que l’on retrouve le code applicatif chargé d’envoyer la requête de mise à jour du statut du prêt après l’action de l’utilisateur-ice.

```
<script setup lang="ts">
import type { AuthenticationToken } from "~/types/authentication";
import {
  ButtonText,
  LoanStatus,
  LoanUpdateMessage,
  LoanUpdateType,
} from "../../types/loan";
const userCookie = useCookie<AuthenticationToken>("userCookie");

interface updateLoanDataValues {
  status: LoanStatus;
  starting_date?: String;
  ending_date?: String;
}
interface LoanDates {
  starting_date: String;
  ending_date: String;
}
interface Props {
  loanUpdateType: LoanUpdateType;
  dates?: LoanDates;
  loanId: String;
}
const props = defineProps<Props>();
```

```
const loanStatus = ref("");
const errorMessage = ref("");
const isUpdated = ref(false);
const hasFailed = ref(false);

const loanUpdateMessage = computed(() => {
  switch (loanStatus.value) {
    case LoanStatus.Active:
      return LoanUpdateMessage.Active;
    case LoanStatus.Denied:
      return LoanUpdateMessage.Denied;
    case LoanStatus.ReturnRequested:
      return LoanUpdateMessage.ReturnRequested;
    case LoanStatus.ReturnConfirmed:
      return LoanUpdateMessage.ReturnConfirmed;
    case LoanStatus.ReturnConfirmationPending:
      return LoanUpdateMessage.ReturnConfirmationPending;
  }
});

const url = `/loans/${props.loanId}/`;
async function patchLoanStatus(event: Event) {
  loanStatus.value = (event.target as HTMLInputElement).value;
  const updateLoanData: updateLoanDataValues = {
    status: loanStatus.value as LoanStatus,
  };
  if (props.dates?.starting_date && props.dates?.ending_date) {
    updateLoanData.starting_date = props.dates.starting_date;
    updateLoanData.ending_date = props.dates.ending_date;
  }
  const config = useRuntimeConfig();
  await useFetch(url, {
    baseURL: config.public.apiBoukinBaseUrl,
    method: "PATCH",
    headers: { Authorization: `Token ${userCookie.value.token}` },
    body: updateLoanData,
    onRequestError({ error }) {
      (hasFailed.value = true),
        (errorMessage.value = error.message),
        console.log(error);
    },
    onResponse({ response }) {
      if (response.status === 204) {
        isUpdated.value = true;
      }
    },
    onResponseError({ response }) {
      (isUpdated.value = false),
        (hasFailed.value = true),
        (errorMessage.value = `${response.status} ${response.statusText}: $
{response._data}`);
    },
  });
}
</script>
```

```
<template>
  <div v-if="!isUpdated">
    ...
    <button
      v-else-if="loanUpdateType == LoanUpdateType.AskingBookReturn"
      value="return_requested"
      @click="patchLoanStatus($event)"
      @click.prevent
    >
      {{ ButtonText.AskingBookReturn }}
    </button>
    <button
      v-else-if="loanUpdateType == LoanUpdateType.ReturningLoan"
      value="return_confirmation_pending"
      @click="patchLoanStatus($event)"
      @click.prevent
    >
      {{ ButtonText.ReturningLoan }}
    </button>

  </div>
  <div v-if="isUpdated">
    <p>{{ loanUpdateMessage }}</p>
   ...
  </div>
  <p v-if="hasFailed" class="error">
    Erreur durant la tentative de mise à jour du prêt : {{ errorMessage }}
  </p>
</template>
```

Ce composant contient le bouton qui effectue la requête effective vers l'API, en utilisant la méthode HTTP `PATCH`. La requête contient dans son corps la chaîne de caractères désignant le statut correspondant, utilisé pour mettre à jour l'entrée du prêt modifié en base de données. Si la requête réussit, l'utilisateur-ice reçoit un message de confirmation indiquant le nouveau statut du prêt.

Tests

Comme énoncé durant la présentation des spécifications techniques du projet, la méthode de développement adoptée pour la création de *Boukin* est le *test-driven development*. De ce fait, la grande majorité du code applicatif est couvert par du code de test, étant donné que l'écriture du code de test conditionne celle du code applicatif.

Dans cette section, je vais présenter des tests typiques aussi bien de la couche *back-end* que de la couche *front-end*.

Un test de la couche back-end : la création d'un prêt

Django et **Django REST Framework** incorporent dans leurs bibliothèques de code un ensemble de classes et de méthodes facilitant la rédaction de tests unitaires et leur exécution. Cet ensemble est construit à partir du module *unittest,* faisant partie de la bibliothèque standard du langage de programmation **Python**. Il permet la mise en place d'un contexte d'exécution de suites de tests, la rédaction d'assertions, et l'exécution des tests par lignes de commande.

Prenons l'exemple de la fonctionnalité de l'API REST de *Boukin* permettant la création d'un prêt en base de données, pour le cas de figure où un-e propriétaire d'un livre veut le confier à un-e ami-e. Il s'agit en soi de créer une méthode répondant à une requête utilisant la méthode HTTP `POST` sur la collection de ressources disponible par l'URI `loans/`.

Avant d'écrire le code applicatif, il faut donc créer une classe qui permet de rédiger la suite de tests, chaque test devant répondre à certaines conditions, et s'assurer que la réponse HTTP renvoyée est de bon statut et contient les informations ajoutées en base de données si la requête est recevable.

Cette classe se dénomme `AddLoanTest`, qui hérite de la classe `TestCase` mise à disposition par **Django**. Par le biais d'une méthode `setUp`, on peut créer un ensemble de propriétés qui sont disponibles pour chacun des tests. Cela évite de devoir réécrire ce qui peut nous être utile dans tous les tests.

```
class AddLoanTest(TestCase):
    def setUp(self):
        self.factory = APIRequestFactory()
        self.today = date.today()
        self.loaned_book = Book.objects.create(
            title="titre", author_fname="John", author_lname="Doe",
isbn="9781237894561"
        )
        self.book_owner = User.objects.create(
            username="book_owner", email="book_owner@boukin.org",
password="b00k_Own3r"
        )
        self.book_borrower = User.objects.create(
            username="book_borrower",
            email="book_borrower@boukin.org",
            password="b00k_Borrow3r",
```

```
        )
        self.another_user = User.objects.create(
            username="another_user",
            email="another_user@boukin.org",
            password="Another7_U53r",
        )
        self.ownedbook = OwnedBook.objects.create(
            owner=self.book_owner, book=self.loaned_book
        )
        self.loan_data = {
            "owner": self.book_owner.user_id,
            "borrower": self.book_borrower.user_id,
            "owned_book": self.ownedbook.owned_book_id,
            "starting_date": self.today,
        }
```

On créé ainsi plusieurs utilisateur-ices fictif-ves, un livre et un de ses exemplaires possédés par `book_owner`. On ajoute aussi un dictionnaire `loan_data`, qui contient les données reçues dans la requête HTTP. Enfin, avec `self.factory`, on instancie `APIRequestFactory`, une classe mise à disposition par **Django REST Framework**, qui permet de simuler des requêtes HTTP dans un environnement de test.

Si on veut tester la réussite d'une requête pour la création d'un prêt, il faut donc rédiger ce test.

```
    def test_post_loan_with_success(self):
        request = self.factory.post("loans/", self.loan_data)
        force_authenticate(request, self.book_owner)
        response = CreateLoanByBookOwnerView.as_view()(request)
        updated_owned_book = OwnedBook.objects.get(
            owned_book_id=self.ownedbook.owned_book_id
        )
        self.assertEqual(response.status_code, 201)
        self.assertEqual(response.data["status"], Loan.LoanStatus.LOAN_ACTIVE)
        self.assertEqual(updated_owned_book.status,
OwnedBook.OwnedBookStatus.LOANED)
```

Grâce à `self.factory`, on peut donc simuler une requête HTTP `POST` vers l'URI `loans/` contenant dans son corps les données de `loan_data`. On simule ensuite une authentification avec `force_authenticate`, les tests simulant une réelle authentification se trouvant dans un autre fichier, du fait du caractère plus lourd d'une telle opération. Le code applicatif testé se trouve dans la méthode `CreateLoanByBookOwnerView`, qui reçoit la requête HTTP `request`. Avec la variable `response`, on exécute cette méthode pour recevoir la valeur de la réponse HTTP créée par `CreateLoanByBookOwnerView`.

Plusieurs assertions permettent de vérifier que la réponse est bien celle attendue : on veut recevoir le code HTTP `201 Created`, qui confirme la création d'une ressource ; on s'assure que le corps de la réponse contient aussi le statut du prêt, et on vérifie en base de données, grâce à `updated_owned_book`, que le statut du `OwnedBook` a bien été modifié depuis `AVAILABLE` à `LOANED`.

Un test de la couche front-end : le formulaire de création de prêt

Pour cette action vue côté *back-end*, qu'en est-il du côté de l'interface utilisateur-ice ? Un formulaire doit être disponible pour pouvoir créer un prêt, avec les informations concernant un livre et la personne à qui on le prête. Ces actions doivent aussi pouvoir être testées.

Pour ce qui concerne le développement de la couche *front-end*, **Nuxt** met à disposition un ensemble d'outils facilitant la rédaction de tests, bâtis à partir de l'environnement d'exécution de tests *Vitest*. De plus, les bibliothèques *testing-library* et *jest-dom* sont utilisées pour permettre d'écrire des tests faisant attention à l'accessibilité des éléments d'une interface, sans avoir à écrire du code de test dépendant de l'implémentation du code applicatif, ce qui est à absolument éviter.

Le formulaire de création de prêt doit être accessible à partir d'une page distincte, dont on peut trouver le lien sur la page "Mes prêts". Un fichier de tests spécifique est créé, dénommé `newLoan.nuxt.test.ts`, les extensions `nuxt.test.ts` permettant à **Vitest** d'identifier le code qui y présent comme étant du code de test.

Pour ce qui concerne des tests pour une couche *front-end* d'une application, qui nécessite donc la création d'un rendu d'un composant ou d'une page, il peut être pertinent de distinguer les éléments statiques (toujours visibles dans la structure du document) de ceux dynamiques (dépendant de la récupération de données ou des changements de l'état du document). Dans le deuxième cas, l'apparition ou la modification des données peuvent prendre un certain temps, à prendre en compte dans le contexte d'exécution des tests.

Deux suites de tests sont donc décrites dans ce fichier de test, de la manière suivante :

```
describe("new loan page static elements test suite", () => {
  beforeEach(async () => {
    await renderSuspended(NewPage);
  });
  afterEach(() => cleanup());

// ...

describe("new loan page dynamic elements test suite", () => {
  const userFriendships: Friendship[] = require("./fixtures/friendships.json");
  const userOwnedBooks = require("./fixtures/books/multipleownedbooks.json");
  beforeEach(async () => {
    await renderSuspended(NewPage);
  });
  afterEach(() => cleanup());
  const mocks = vi.hoisted(() => {
    return {
      getUserFriends: vi.fn(),
      getOwnedBooks: vi.fn(),
      postLoan: vi.fn(),
    };
  });
  registerEndpoint(`/users/${userId}/friendships/`, mocks.getUserFriends);
  registerEndpoint(`/owned-books/`, mocks.getOwnedBooks);
  registerEndpoint(`/loans/`, mocks.postLoan);
```

```
  mockNuxtImport("useCookie", () => {
    return () => {
      return {
        value: {
          email: "wowilovebooks@boukin.org",
          user_id: userId,
          token: token,
        },
      };
    };
  });
  mocks.getUserFriends.mockReturnValue(userFriendships);
  mocks.getOwnedBooks.mockReturnValue(userOwnedBooks);

// ...
```

Dans la première suite de tests, le contexte est assez simple, demandant simplement de créer le rendu de la page pour ensuite vérifier les assertions dans un test proprement dit. Dans la deuxième, il s'agit de tests d'intégration demandant de simuler des interactions avec l'API REST de *Boukin*, il faut donc mettre en place d'avantage d'éléments permettant de simuler la récupération de données, la présence d'un cookie permettant d'authentifier les requêtes vers l'API, etc.

Prenons tout d'abord un exemple de test d'un élément statique : dans le formulaire, une entrée permettant la sélection d'un-e ami-e, identifiable clairement pour cette action. Le code de test permettant de vérifier la présence d'un tel élément est le suivant :

```
  it("displays a selection of friends to loan books to", () => {
    const friendSelection = screen.getByRole("combobox", {
      name: /Sélectionner un-e ami-e/i,
    });
    expect(friendSelection).toBeVisible();
  });
```

testing-library met à disposition l'élément `screen`, soit le rendu du DOM qui peut être testé. Avec la méthode `getByRole`, recommandée comme premier levier d'action pour les requêtes d'éléments du DOM, on veut pouvoir retrouver un élément du rôle d'accessibilité *Accessible Rich Internet Applications* (ARIA) `combobox`, utilisé pour l'identification des balises HTML `<select>`. Cet élément HTML permet de lister plusieurs options, possiblement dans un formulaire. Une règle d'accessibilité fondamentale avec un tel élément, utile notamment pour les outils de lecture d'écran, est d'associer un élément `<label>` avec `<select>`, rendant ce dernier accessible.

Avec `getByRole`, on vérifie déjà si l'élément est bien présent dans le DOM. L'assertion `toBeVisible`, disponible avec *jest-dom*, permet de s'assurer qu'aucune règle CSS n'empêche de rendre invisible cet élément et que son élément parent est lui aussi visible. On s'assure ainsi que l'entrée du formulaire est bien visible pour tout-e utilisateur-ice.

Pour un test d'un élément dynamique, il peut être intéressant de vérifier que cette liste des ami-e-s contient bien les noms des ami-e-s de l'utilisateur-ice. On doit pouvoir choisir l'un de ses ami-e-s en entrée du formulaire, afin de l'identifier comme la personne recevant le livre dans la structure de données envoyées vers l'API REST. Le test correspondant est le suivant :

```
  it("lists all friends to select a friend to loan a book to", async () => {
    const friendsUsernames: string[] = [];
    const userActiveFriendships = userFriendships.filter(
      (friendship) => friendship.status === "friendship_active",
    );
    userActiveFriendships.forEach(function (friendship: Friendship) {
      friendsUsernames.push(friendship["friend"]["username"]);
    });
    const friendSelection = screen.getByRole("combobox", {
      name: /Sélectionner un-e ami-e/i,
    });
    const { getAllByRole } = within(friendSelection);
    const friendOptions = getAllByRole("option");
    for (let index = 1; index < friendsUsernames.length; index++) {
      expect(friendOptions[index]).toHaveTextContent(
        friendsUsernames[index - 1],
      );
    }
  });
```

`userFriendships` est un dictionnaire contenant les données présentes dans un fichier à part, permettant de disposer d'informations testables dans un tel contexte. Avant de construire l'assertion s'assurant que chacune des `<option>` présentes dans l'élément `<select>` contient bien les noms des ami-e-s, on construit un tableau `friendsUsernames` contenant ces noms, filtrés depuis `userFriendships`.

Ensuite, une requête pour récupérer l'élément du DOM désiré est faite, puis, dans cette élément, on requête toutes les `<option>`. Enfin, on boucle sur la longueur du tableau `friendsUsernames`. Avec la méthode `toHaveTextContent`, il est possible de vérifier que chacune des `<option>` contient bien un nom d'ami-e.

Veille concernant les failles de sécurité

Plusieurs failles de sécurité courantes peuvent apparaître dans le cadre du développement d'une application web. Il est nécessaire de les connaître et de s'assurer d'avoir mis en place des moyens de les éviter. Au cours du développement de *Boukin*, il nous est paru nécessaire de vérifier que les outils que nous utilisions nous permettait d'éviter l'apparition de problèmes de sécurité, notamment ceux qui alloueraient un accès aux informations contenues en base de données.

La plupart des *frameworks* utiles au développement web contiennent par défaut des moyens de contenir des failles de sécurité courantes. Une des plus connues est celle désignée sous le nom d'**injection SQL** : il s'agit d'une technique permettant, par le biais par exemple d'un formulaire dont les données sont envoyées vers un serveur, d'injecter une requête SQL qui supprimerait partiellement ou totalement les informations contenues en base de données ou de les récupérer. Un bon point de départ pour comprendre le fonctionnement d'une telle technique est la documentation mise à disposition par la fondation Mozilla, une ressource riche pour tout ce qui concerne le développement web : https://developer.mozilla.org/en-US/docs/Glossary/SQL_Injection.

Concrètement, une personne peut, dans une entrée d'un formulaire, rédiger une requête SQL valide, qui pourrait ensuite être interprétée par le gestionnaire de base de données. C'est une faille de sécurité très bien documentée et qui peut poser de graves problèmes (accès à des données personnelles, dysfonctionnement de l'application). Elle est, heureusement, facilement évitable.

L'utilisation de **Django** et de l'*Object-relational mapping* associé prémunit de la possibilité d'une telle faille de sécurité. De quelle manière ? Par l'utilisation de requêtes paramétrées. La documentation de **Django** donne des informations à ce sujet, consultées à cette adresse: https://docs.djangoproject.com/en/5.1/topics/security/#sql-injection-protection. Les requêtes SQL effectuées par **Django** séparent le code SQL des données rédigées et reçues par un-e utilisateur-ice. Ces données deviennent des paramètres de la requête. Ainsi, toute donnée reçue ne peut être interprété comme du SQL mais comme des chaînes de caractère.

Autre enjeu de sécurité : l authentification. Pour sa sécurisation, **Django** s'occupe du chiffrement du mot de passe d'un-e utilisateur-ice. La documentation du *framework* est très complète à ce sujet et permet de mettre en place une configuration différente de celle par défaut si on le souhaite : https://docs.djangoproject.com/en/5.1/topics/auth/passwords/. Un mot de passe en base de données, sécurisé par **Django**, n'est donc pas stocké en clair. Par l'utilisation d'une fonction de dérivation cryptographique, la chaîne de caractères initiale du mot de passe est transformé en un *hash* selon un algorithme. Le mot de passe est ainsi chiffré après l'ajout d'un *salt*, une chaîne de caractères aléatoire qui permet de distinguer des mots de passe similaires. Dès lors, pour un-e utilisateur-ice donné, l'utilisation du bon mot de passe permet de s'authentifier, sans qu'on puisse le connaître si on obtient l'accès à la base de données.

Une autre faille courante est le **Cross-site scripting (XSS)**. Il s'agit d'une technique où une personne peut injecter du code interprétable dans un navigateur, afin qu'il soit exécuté par d'autres utilisateur-

ice-s du site. Cela est utilisé notamment pour récupérer les informations personnelles stockées dans un cookie, pour rediriger vers un autre site, etc. Cela peut se faire, par exemple, en utilisant une entrée de formulaire en y insérant du texte qui peut être interprété comme du code JavaScript. Ce code peut ensuite être stocké en base de données s'il en passe les règles de validation. La documentation de la fondation Mozilla donne une définition de ce type de faille de sécurité : https://developer.mozilla.org/en-US/docs/Web/Security/Types_of_attacks#cross-site_scripting_xss.

Cross-site scripting est un terme qui recouvre en réalité un ensemble de problèmes de sécurité, que l'on peut distinguer entre les scripts conservés côté serveur, les injections de code côté client (à partir d'URLs présentes sur un site, par exemple), ou par modifications du DOM.

Pour empêcher l'exécution de script malicieux, on peut effectuer une validation des données qui peuvent être entrées en base de données. Empêcher par exemple des caractères spéciaux d'être utilisés, comme < et > qui permettent d'insérer de l'HTML pouvant contenir du code JavaScript interprétable ensuite côté client. Mais ce n'est qu'une partie de la solution, et la gestion des failles rassemblées sous la domination **XSS** peut demander l'implémentation de politiques de sécurité au niveau de l'infrastructure (notamment avec certaines options de configuration de **Nginx**), permettant l'identification de scripts et d'éviter leur exécution. Pour ce qui concerne **Nuxt**, l'utilisation du module NuxtSecurity (https://nuxt-security.vercel.app/) ajoute automatiquement un ensemble d'en-têtes HTTP pour les requêtes faites auprès du serveur de l'application front-end. Cela définit une politique de sécurité moins permissive dans le code à exécuter côté client.

Description d’une situation de travail ayant nécessité une recherche

Lors de la mise en place de la politique d’authentification d’un-e utilisateur-ice, avoir choisi une identification par un critère différent de celui par défaut de **Django** a nécessité une recherche. En effet, la table `Users` et les identifiants à utiliser selon le module d’authentification de **Django** comportent un certain nombre d’éléments de configuration par défaut. Dès lors, si on veut les modifier, il est nécessaire de suivre certaines étapes et ainsi éviter quelques problèmes de sécurité.

Tout particulièrement, nous avions décidé que l’identifiant à associer au mot de passe pour l’authentification soit l’adresse email d’un-e utilisateur-ice, et non son nom d’usage (ou *username*). Ce choix est motivé par la possibilité d’enregistrer des prêts à des ami-e-s non inscrit-e-s à Boukin ; dès lors, il est possible d’avoir des noms d’usage similaires en base de données, ce qui rentrerait en conflit avec la contrainte d’unicité pour la colonne `username` de la table `Users` de la base de données. Pour cette raison, il a fallu modifier la configuration par défaut de l’authentification mise en place par **Django**.

Bien heureusement, la documentation de **Django** à ce sujet est fort complète et la suivre pas à pas permet de gérer cette première partie du problème. La page *Customizing authentication in Django* (https://docs.djangoproject.com/en/5.1/topics/auth/customizing/) met à disposition de nombreux éléments pour la mise en place d’une authentification personnalisée. C’est notamment la section *Specifying a custom user model* qui a répondu à notre cas d’usage : comment indiquer au module d’authentification de **Django** que l’identifiant principal est contenu dans la colonne `email`.

Pour cela, il y a plusieurs actions à réaliser : - il faut que le modèle `User` hérite de la classe `AbstractBaseUser`, permettant notamment de ne pas devoir implémenter soi-même une méthode de chiffrement de mot de passe ; - il faut créer une classe `UserManager`, qui permet d’avoir à disposition une méthode `create_user` pour créer l’entrée associée à l’utilisateur-ice en base de données. Cette méthode est notamment utile dans le code de test. - il faut préciser la propriété `USERNAME_FIELD = "email"` dans le modèle `User`.

En implémentant cela, il est ainsi possible d’utiliser l’adresse email et le mot de passe comme identifiants pour l’authentification des requêtes vers l’API REST de *Boukin*. Intervient ensuite la deuxième partie du problème : la génération d’un token valide à partir de ces identifiants.

Django REST Framework implémente une authentification par token par défaut. Toutefois, celle-ci attend l’utilisation d’un *username* et non d’une adresse email pour que ce procédé soit valide. Dès lors, du fait de cette modification dans le modèle `User`, il fallait implémenter une génération de token fonctionnelle à partir de l’adresse email et du mot de passe.

Là aussi, la consultation de documentation de **Django REST Framework** s’est révélée d’une grande aide, détaillant le fonctionnement de l’authentification par token : https://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication. Comme nous voulions disposer d’un

point d'accès par l'API REST pour récupérer le token au moment de l'authentification, la méthode par l'exposition d'un tel point d'accès était la plus pertinente.

De plus, il fallait personnaliser l'authentification pour autoriser la génération d'un token. La section *Custom authentification* de la page précédemment citée donne les informations nécessaires. Toutefois, il me fallait aussi intégrer la nécessite de vérifier qu'un-e utilisateur-ice avait bien activé son compte après consultation du message envoyé sur sa boîte mail après inscription. Une colonne `has_email_confirmed` (de type booléen) a donc été ajoutée, qui permet cette validation.

En allant lire le code source de **Django REST Framework**, j'ai pu consulter le *serializer* permettant la validation des données puis l'inscription en base de celles-ci, et modifier ce qui m'était nécessaire pour permettre d'utiliser l'adresse email comme identifiant.

```
class GenerateTokenSerializer(serializers.Serializer):
    email = serializers.CharField(label=_("Email"), write_only=True)
    password = serializers.CharField(
        label=_("Password"),
        style={"input_type": "password"},
        trim_whitespace=False,
        write_only=True,
    )
    token = serializers.CharField(label=_("Token"), read_only=True)

    def validate(self, attrs):
        email = attrs.get("email")
        password = attrs.get("password")

        if email and password:
            user = authenticate(
                request=self.context.get("request"), email=email,
password=password
            )

            # The authenticate call simply returns None for is_active=False
            # users. (Assuming the default ModelBackend authentication
            # backend.)
            if not user:
                msg = _("Unable to log in with provided credentials.")
                raise serializers.ValidationError(msg, code="authorization")
            if user.has_email_confirmed is False:
                msg = _("Unable to log in with unconfirmed account.")
                raise serializers.ValidationError(msg, code="authorization")
        else:
            msg = _('Must include "email" and "password".')
            raise serializers.ValidationError(msg, code="authorization")

        attrs["user"] = user
        return attrs
```

Grâce à cela, si une personne n'utilise pas les bons identifiants, elle ne peut recevoir de token permettant d'authentifier ces requêtes. De même si la confirmation par email n'a pas été effectuée.

Mes premiers réflexes concernant la recherche d'informations est toujours d'aller consulter la documentation des bibliothèques et *frameworks* utilisés. Pour des outils comme **Django**, développés

depuis plus d’une décennie et utilisés par un grand nombre d’équipes de développement, les problèmes courants sont facilement déductibles après une lecture attentive de cette documentation.

Dans les cas où un cas d’usage plus précis est à implémenter, je consulte Stack Overflow, en favorisant les réponses mises en avant les plus récentes ou mises à jour. En général, cela me permet d’avoir une première idée de comment implémenter le code que je veux rédiger. Si cela ne fonctionne pas, ou pas d’une manière qui me convienne, une lecture plus précise de la documentation liée aux méthodes ou classes mentionnées dans les réponses sur ce site me donne d’avantage de pistes pour progresser dans la rédaction de code.

Intégration continue

Pour faciliter le travail du développement de l'application de manière collective, et s'assurer d'avoir une application toujours fonctionnelle à mesure que de nouvelles portions de code applicatif étaient ajoutées au dépôt, il est rapidement devenu nécessaire d'insérer certaines pratiques d'automatisation. L'intégration continue, dans un cycle de développement logiciel, rassemble ainsi les pratiques permettant de s'assurer que chaque nouvel élément de code ajouté à un projet soit intégrable en maintenant un état fonctionnel de l'application.

En ayant décidé dès le départ d'adopter une méthode de travail comme le *test-driven development*, s'assurer d'avoir un environnement d'exécution neutre des tests unitaires et d'intégration était de première importance. Cet environnement, et non celui d'un ordinateur personnel, sert ainsi de référence pour l'ensemble de l'équipe de développement. Cela permet d'éviter tout effet de bord (des tests valides du fait d'un système d'exploitation différent, ou par la présence d'un fichier spécifique, par exemple), mais aussi de pouvoir tester l'application dans différents contextes.

De plus, certaines pratiques mises en place au cours du projet ont permis de s'assurer que des règles communes de rédaction aient été mises en place. Cela peut se faire dans au moins deux dimensions : - le formatage du code (en termes d'espaces / tabulations, caractères spéciaux utilisés, etc.) ; - le *linting* du code, c'est à dire d'éviter l'incorporation d'erreurs et de *bugs* par une vérification syntaxique, voire sémantique du code statique, c'est-à-dire par l'analyse du code tel qu'il est écrit et non à son exécution. On peut y ajouter aussi la mise en place de bonnes pratiques selon des spécifications externes propres à un langage de programmation utilisé.

Pour automatiser tout cela, nous avons mise en place deux formes de script, qui automatisent l'exécution des tests et l'analyse statique de code.

Pour l'exécution des tests, nous avons utilisé l'environnement d'exécution d'intégration continue propre à **Gitlab**, qui met à disposition des serveurs faisant exécuter des *runners*, c'est-à-dire une solution logicielle permettant de créer un environnement d'exécution neutre pour n'importe quel dépôt de code présent sur **Gitlab**. Cela se fait par le biais d'un fichier `.gitlab-ci.yml`, présent à la racine d'un projet. Selon les conditions décidées à l'avance, ce fichier est lu par un *runner* de **Gitlab** et exécuté. Nous avons ainsi automatisé l'exécution des tests, pour les couches *front-end* et *back-end*, lors de toute action liée à une merge request avant sa validation : lorsqu'une *merge request* est créée, lorsqu'un *commit* est rajouté à cette *merge request* (après correction suite à relecture du code, par exemple).

Pour les deux couches de l'application, un taux de couverture du code applicatif par les tests est présenté. S'il y a lieu, un ratio (positif ou négatif) lié à la *merge request* est précisé, afin de savoir si le nouveau code présente effectivement les tests qui lui sont nécessaires.

Le script suivant permet de créer l'environnement de test puis d'exécuter les tests associés au dépôt de code de la couche *front-end*, lorsqu'une *merge request* est créée pour la branche principale ou la branche de développement. De même lorsqu'un *commit* est ajouté sur la branche de développement :

```
test-for-merge-request:
  stage: build-env-and-test
  image: node:lts-slim
  script:
    - npm i
    - npm run coverage
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml
  coverage: /All files[^\|]*\|[^\|]*\s+([\d\.]+)/
  rules:
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" &&
$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main"
    - if: $CI_PIPELINE_SOURCE == "merge_request_event" &&
$CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "dev"
    - if: $CI_PIPELINE_SOURCE != "merge_request_event" && $CI_COMMIT_BRANCH ==
$CI_DEFAULT_BRANCH
```

Pour le formatage et le *linting* du code, si ces processus sont effectivement automatisables dans le contexte d'un environnement neutre d'intégration continue, nous avons mis en place ces pratiques en amont, par le biais de l'outil *pre-commit* pour le back-end, et *husky* pour le front-end. Fondamentalement, ces outils fonctionnent de manière similaire : par l'utilisation du *hook* `pre-commit` de **Git**, un script rédigé à l'avance est exécuté. Ainsi, à chaque rédaction d'un commit, ce script est exécuté ; s'il réussit, alors le *commit* est effectivement créé et peut être ajouté au dépôt distant de code. Un rapport concernant la couverture du code par les tests est aussi créé, permettant de montrer le pourcentage de couverture en page de présentation du projet.

Pour le back-end, nous utilisons *pre-commit* pour exécuter deux outils **Python** : *black*, qui met en place un formatage du code ; et *flake8*, qui s'occupe de la vérification syntaxique et sémantique (par exemple, en faisant en sorte que toute variable déclarée soit effectivement utilisée, en vérifiant les règles d'indentation, etc.).

Pour le front-end, *husky* nous permet d'automatiser l'exécution de *prettier* pour la correction automatique du formatage du code, et d'*eslint* pour le *linting*.

De plus, nous avons ajouté un script de génération automatique d'un schéma OpenAPI pour la couche back-end de Boukin. Cela permet de documenter de manière claire l'API REST utilisée par l'application, et de ne pas avoir à générer manuellement ce schéma. Cela a lieu à chaque merge request faite vers la branche main du dépôt de code, afin que chaque nouvelle version de l'API effectivement déployable soit documentée.

Déploiement

Pour l'environnement de production des trois couches constituant l'application *Boukin* (base de données, serveur back-end, serveur front-end), nous avons décidé d'utiliser des containers Docker, orchestrés par un fichier `docker-compose`, exécutés sur le serveur d'Ada Tech School.

L'utilisation de containers Docker permet d'isoler l'environnement d'exécution de chacun des trois processus, tout en contrôlant leur communication par le biais d'un réseau, et la sauvegarde de la base de données grâce à un volume. A terme, il sera possible de mettre à jour simplement les images utilisées par les containers en automatisant le processus de création des images et le redémarrage de l'orchestration à partir de ces nouvelles images.

Pour pouvoir accéder à l'application, il nous était possible d'associer un sous-domaine `boukin` au nom de domaine `adaschool` lié au domaine de premier niveau `fr`, ce qui nous permet de disposer de l'URL `https://boukin.adaschool.fr`. De même pour l'API REST, disponible avec `https://api.boukin.adaschool.fr`.

Après création de ces sous-domaines, et leur association avec l'adresse IP du serveur d'Ada Tech School, il est nécessaire de pouvoir les lier aux ports applicatifs utilisés par les containers Docker. Ceci est la responsabilité de **Nginx**, installé sur le serveur, et ici utilisé comme *reverse proxy*. Après création des fichiers de configuration **Nginx** pour `boukin.adaschool.fr` et `api.boukin.adaschool.fr`, toute requête HTTPS reçue par le serveur est traitée par Nginx, qui s'occupe ainsi de l'association entre port et domaine, puis la redirection vers l'application demandée. Un client peut ainsi envoyer des requêtes vers la même adresse IP que d'autres applications en cours d'exécution sur ce serveur, et voir ces requêtes rediriger vers le bon port applicatif. Puis, recevoir les réponses correspondantes.

```
server {

        server_name boukin.adaschool.fr www.boukin.adaschool.fr;

        location / {
                proxy_pass         http://localhost:3001;
                proxy_redirect     off;
                proxy_set_header   Host $host;
        }


    listen 443 ssl; # managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot



    ssl_certificate /etc/letsencrypt/live/boukin.adaschool.fr/fullchain.pem; #
managed by Certbot
    ssl_certificate_key /etc/letsencrypt/live/boukin.adaschool.fr/privkey.pem; #
managed by Certbot
} server {
    if ($host = boukin.adaschool.fr) {
```

```
        return 301 https://$host$request_uri;
    } # managed by Certbot


    if ($host = www.boukin.adaschool.fr) {
        return 301 https://$host$request_uri;
    } # managed by Certbot



        server_name boukin.adaschool.fr;

    listen 80;
    return 404; # managed by Certbot
}
```

Il reste toutefois la question du trafic sécurisé par le protocole *Transport Layer Security*, permettant l'accès à un domaine par le biais du protocole HTTPS. **LetsEncrypt** est aussi installé sur le serveur d'Ada Tech School, et permet donc, à partir du moment que l'on dispose d'entrées DNS liant le nom de domaine à l'IP du serveur, de générer les certificats TLS correspondants permettant d'encrypter le trafic réseau de serveur à client. Il faut ensuite indiquer où se situent les certificats générés sur le serveur dans les fichiers de configuration **Nginx**. Un navigateur peut ainsi reconnaître la fiabilité des certificats, et ainsi permettre une connexion sécurisée vers le site web.

L'application back-end, correctement configurée en acceptant les requêtes provenant de `boukin.adaschool.fr`, par le biais de la variable `CORS_ALLOWED_ORIGINS`, peut ainsi traiter toute requête provenant de ce domaine. Toute requête provenant d'autre domaine, en vertu de la gestion de la sécurité propre au mécanisme Cross-Origin Ressource Sharing, ne peut être traitée.

Évolutions futures possibles

Bien que nous ayons réussi à développer et déployer une première version de l'application *Boukin*, conforme au *Minimum Viable Product* conçu au début du projet, *Boukin* manque de fonctionnalités importantes.

Tout d'abord, la recherche effective de références bibliographiques normalisées, sans à ce qu'un-e utilisateur-ice ait à ajouter manuellement un livre en base de données, devrait être de toute première priorité. Il est peu probable que *Boukin* devienne plaisante à son utilisation sans cela. Par manque de temps, et face à l'ampleur de la tâche pour accomplir quelque chose de satisfaisant, nous avons décidé de mettre ce problème de côté au cours du projet.

Il serait nécessaire de passer par un service externe pour rechercher de nouvelles références bibliographiques et les ajouter dans la base de données de *Boukin*, sans avoir à valider les informations ajoutées. Ainsi, lorsqu'un-e utilisateur-ice rechercherait un livre à ajouter dans sa bibliothèque, il serait possible de retrouver directement l'édition correspondante. A partir du moment où une référence est ajoutée en bibliothèque personnelle, celle-ci serait ajoutée à la table *Books* de la base de données.

Cela permettrait d'ajouter progressivement de nouvelles références à la base de données, sans devoir constamment dépendre et envoyer des requêtes à un service externe. Dans les cas où une référence n'est pas trouvée par ce service externe, alors l'ajout manuel serait possible ; il faut toutefois établir des normes bibliographiques claires afin d'éviter d'ajouter des informations erronées. A minima, l'association entre un ISBN et un titre de livre est à vérifier. Cela ne fonctionnerait toutefois pas dans les cas où une édition d'un livre n'a pas d'ISBN (ce qui est possible pour les éditions d'avant 1970), ou pour les cas où un même ISBN est utilisé pour deux références bibliographiques (rare, mais possible !).

Une autre fonctionnalité à implémenter serait la possibilité de pouvoir rendre un livre dans un espace neutre, dans le cas où deux personnes ne sont plus amies mais que le / la propriétaire d'un livre veuille le récupérer. Cela serait un problème intéressant à résoudre, afin de présenter des lieux de confiance pour entreposer ces livres, sans avoir à contacter une personne avec qui l'on ne veut plus communiquer.

Pour ce qui concerne les pratiques de développement plus proprement dites, il serait utile de pouvoir : - déployer automatiquement les nouvelles versions de production, à mesure que de nouvelles fonctionnalités sont ajoutées ; cela doit pouvoir se faire par l'automatisation de la mise à jour des containers Docker à partir d'images. - incorporer des tests automatisés end-to-end, afin de vérifier le comportement effectif à l'utilisation de l'application, sans avoir à le faire manuellement.