VALKYLIT

Projet Ecom - M2 GI

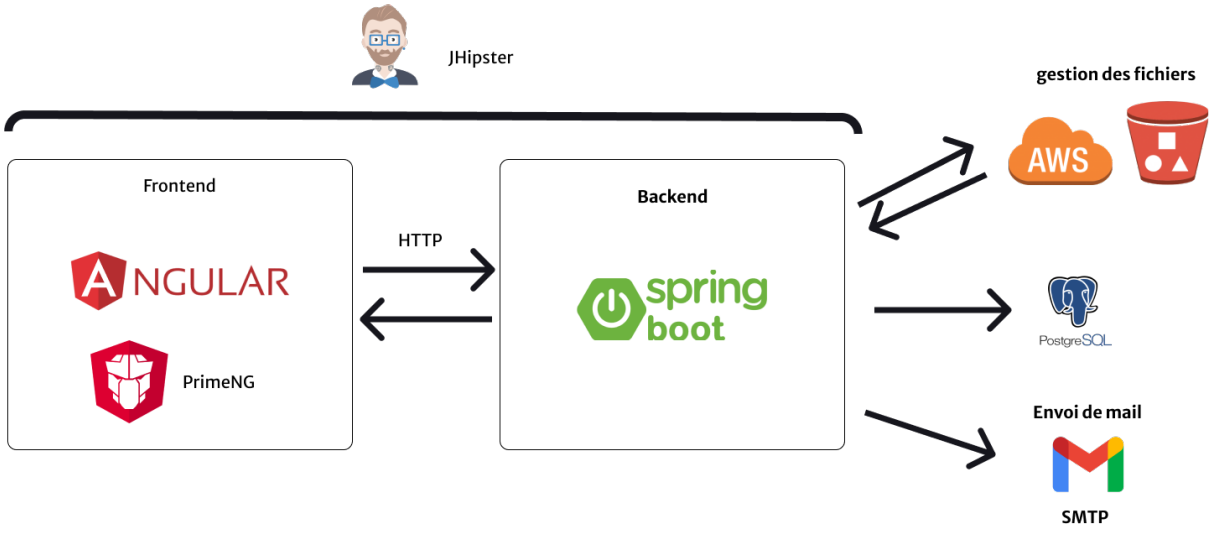
Rapport

Tom DOLIDON, Florent POUZOL, Yazid CHERITI, Hugo TRIOLET

Présentation du document

Vous trouverez au sein de ce document une présentation des problématiques rencontrées ainsi que les solutions mises en place dans le développement de notre plateforme de commerce en ligne ValkyLit. Cette application a été développée dans le cadre du projet E-COM du Master 2 Génie Informatique de l'IM²AG à l'Université Grenoble Alpes. L'objectif de ce projet était de fournir une plateforme viable, avec des fonctionnalités essentielles selon différents profils utilisateurs ciblés, la mise en place de processus de tests et de déploiement continus ainsi que la montée en compétences sur des technologies orientées web.

Solution Technique



Le diagramme ci-dessous représente l'architecture de notre projet en détaillant les choix technologiques associés. Nous avons décidé d'utiliser le framework JHiPSTER afin de nous aider à concevoir une application monolithique composée :

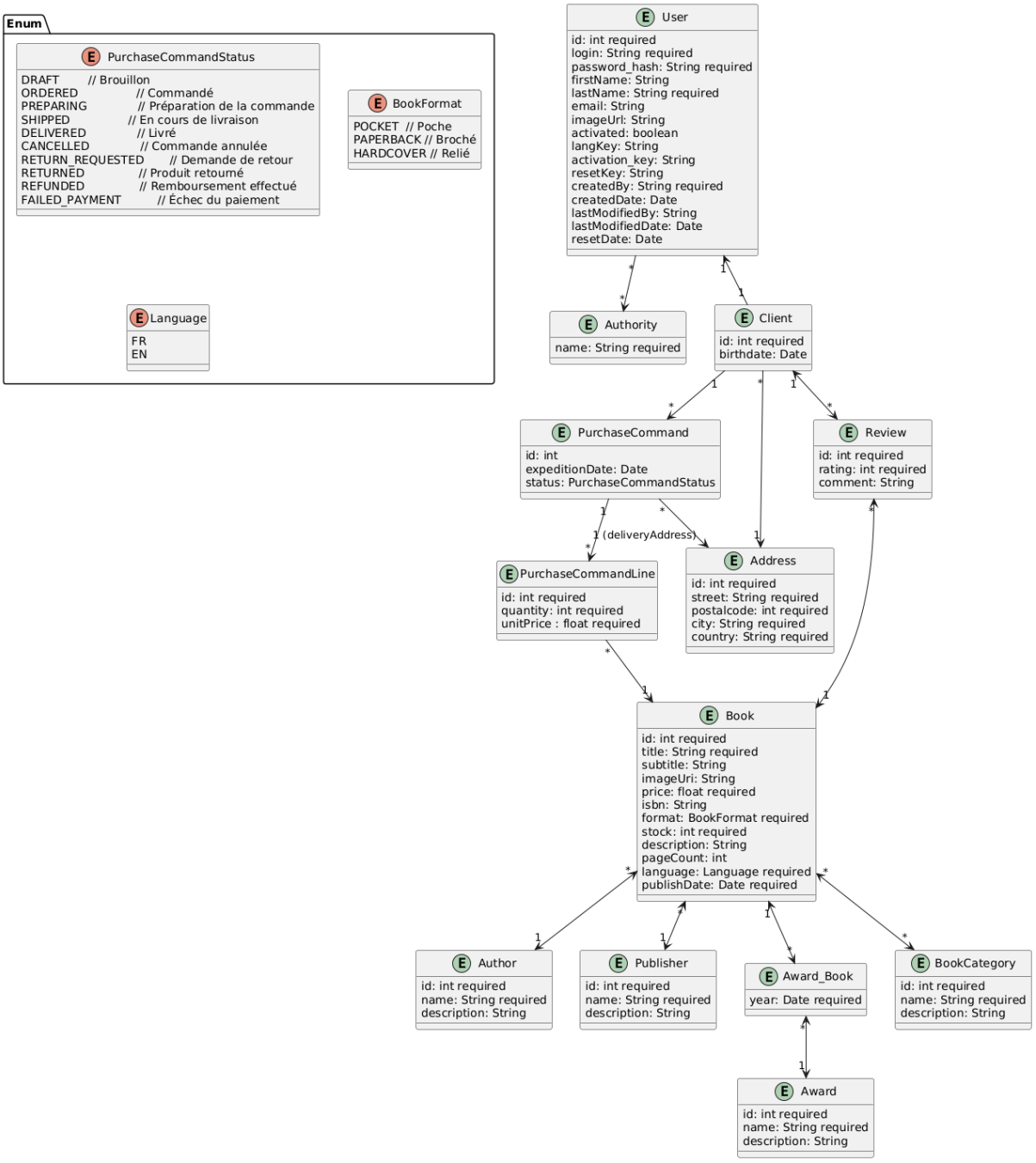
- d'un frontend en Angular TypeScript auquel nous avons ajouté la suite de composants PrimeNG
- d'un backend Java Spring Boot

JHiPSTER nous a permis de nous concentrer sur les fonctionnalités métiers du projet en nous proposant la conception d'un modèle de données à partir d'un fichier de configuration, la création d'un back-office pour les administrateurs, la gestion des rôles utilisateurs et authentification, la création d'une API Rest liée à nos entités.

A cela viennent s'ajouter les éléments suivants :

- Une base de données Postgres
- Un bucket AWS S3 : nous avons mis en place un bucket pour le stockage d'images utilisé sur le site, en particulier pour le stockage des couvertures de livres. A l'ajout d'un nouveau livre, l'image de la couverture est directement stockée sur le bucket, le chemin d'accès étant enregistré sur la base de données.
- Un Serveur SMTP gmail permettant l'envoi de mail de confirmation d'inscription, ainsi qu'un mail de récapitulatif à l'achat d'une commande**.**

Modèle de données



Ce modèle de données nous a permis d'élaborer notre modèle JDL dans le but de générer les entités en base de données par JHiPSTER et de consolider des données de base à injecter en base de données lors du déploiement initial du projet.

Plusieurs choix ont été faits afin de déterminer les relations et la répartition des données entre tables afin de retrouver diverses informations d'un livre, telles que les auteurs, les éditeurs, les catégories, les récompenses, etc.

Ainsi, ce modèle conceptuel décrit les tables suivantes :

- ***Book* :** table des livres, un livre ayant un éditeur, un/des award(s) ou non, une ou des catégories, un auteur, et associé à un ou des reviews et pouvant se trouver dans plusieurs PurchaseCommandLine.
- **Author :** table des auteurs, un auteur étant associé à plusieurs livres
- **Publisher :** table des éditeurs, un éditeur étant associé à plusieurs livres
- **Award_Book :** table décrivant l'année/l'édition d'une récompense, pouvant donc être associée à plusieurs récompenses.
- **BookCategory :** table des catégories (genres) de Book, une catégorie étant associé à de multiples livres
- **Award :** table des fiches de description des awards, un award étant associé à un Award_Book (une année de l'award).
- **PurchaseCommand :** table des commandes, une commande n'étant associée qu'à un unique Client et à une unique adresse, mais à une ou de multiples lignes de commandes.
- **PurchaseCommandLine :** table des lignes d'une commande, chaque ligne étant associée à une unique commande et un unique livre (dans une quantité possiblement multiple si désiré).
- **Review :** table des review, une review étant associée à un unique Book et un Client unique.
- **Address :** table d'adresses, une adresse se retrouve associée avec plusieurs commandes et avec un seul Client.
- **User :** table d'utilisateur classique, possédant un nombre d'autorités variant. Générée par JHipster par défaut. Extension des champs avec la table Client ci-dessous par contrainte.
- **Client :** table "extension" de User, où des informations supplémentaires sont (ou peuvent être) ajoutées. Un Client possède une adresse, peut être associé à plusieurs récompenses et est aussi associé à plusieurs commandes.
- **Authority :** table sur les différents rôles disponibles pour un User.
- **User :** Table créée par défaut par JHipster, qualifiant les utilisateurs lambdas NON CLIENT de l'utilisation.
- **Language, PurchaseCommandStatus, BookFormat :** énumérations de status sur différents champs (liste des langages disponibles, liste des status différent d'une commande, formats des livres).

Une fois la génération du modèle via JDL avec JHipster, des tables de relations se rajoutent sur livre-catégorie et livre-auteur (les IDs de auteur et catégorie n'étant pas présentes car potentiellement multiples dans la table Book, contrairement à l'ID de l'éditeur).
A partir de ce modèle conceptuel, nous avons pû générer un fichier .jdl dans JDL Studio, afin de faire générer les entités par JHipster dans le dossier projet.

Évolution de l'API Rest

L'API REST a été enrichie par de nouveaux points de terminaison afin de répondre à des besoins spécifiques. Ci-dessous sont détaillés les nouveaux points de terminaison (déclarés dans les ressources) ainsi que d'autres classes.

BookRessource.java :

- **public ResponseEntity<Book> createBook(@RequestParam MultipartFile file, @RequestPart BookCreateDTO bookCreate)**
    - Permet l'ajout d'une image dans le formulaire de création d'un livre, ainsi que son téléchargement sur un service externe en l'occurrence sur un bucket S3 AWS.
- **public ResponseEntity<List<Object[]>> getTopBooksByAverageRating(@RequestParam(value = "limit", defaultValue = "2") int limit)**
    - Permet de récupérer les livres les mieux notés pour les suggérer aux clients en page d'accueil.

ClientRessource.java :

- **public ResponseEntity<Client> getSelfClient()**
    - Permet de récupérer le client à partir d'un utilisateur authentifié pour l'affichage et, ou la modification de ses données personnelles

PurchaseCommandLine.java :

- **public ResponseEntity<PurchaseCommandLine> createPurchaseCommandLine(@Valid @RequestBody PurchaseCommandLine purchaseCommandLine)**
    - Permet l'ajout d'une ligne de commande, modifiée pour les besoins de la persistance du panier

PurchaseCommandRessource.java :

- **public List<PurchaseCommand> getAllPurchaseCommands(@RequestParam(name = "eagerload", required = false, defaultValue = "true") boolean eagerload)**
    - Permet de récupérer l'ensemble des commandes d'un client pour afficher l'historique de celles ci
- **public ResponseEntity<PurchaseCommand> getSelfCurrentPurchaseCommand()**
    - Permet de récupérer le panier d'un client. La "commande d'achat actuelle" d'un client est la "purchaseCommand" enregistrée avec le statut "DRAFT" et avec le client_id du client.
- **public ResponseEntity<PurchaseCommand> clearCart()**
    - Permet de supprimer les lignes de commandes associées à la commande.
- **public ResponseEntity<PurchaseCommand> decrementItemInCart(@PathVariable UUID bookId)**
    - Permet de décrémenter de 1 le champ "quantity" d'une ligne de commande de l'utilisateur actuellement authentifié

- **public ResponseEntity<PurchaseCommand> removeItemFromCart(@PathVariable UUID bookId)**
    - Permet de supprimer une ligne de commande associée à un livre pour l'utilisateur actuellement authentifié
- **public ResponseEntity<PurchaseCommand> addCommandLineToCart(@Valid @RequestBody PurchaseCommandLine purchaseCommandLine)**
    - Permet d'ajouter une ligne de commande associée à un livre à la commande "DRAFT" de l'utilisateur actuellement authentifié
- **public ResponseEntity<PurchaseCommand> updateCart(@Valid @RequestBody List<PurchaseCommandLine> purchaseCommandLines)**
    - Permet de réinitialiser et redéfinir la commande de l'utilisateur actuellement authentifié.

ReviewRessource.java :

- **public ResponseEntity<Review> createReview(@PathVariable UUID bookId, @RequestBody ReviewDTO reviewDTO)**
    - Permet à un client d'écrire un avis sur un livre et de le stocker en BDD
- **public List<Review> getReviewForBookAndClient(@PathVariable UUID bookId)**
    - Permet de récupérer les avis sur un livre pour les afficher

Déploiement et CI/CD

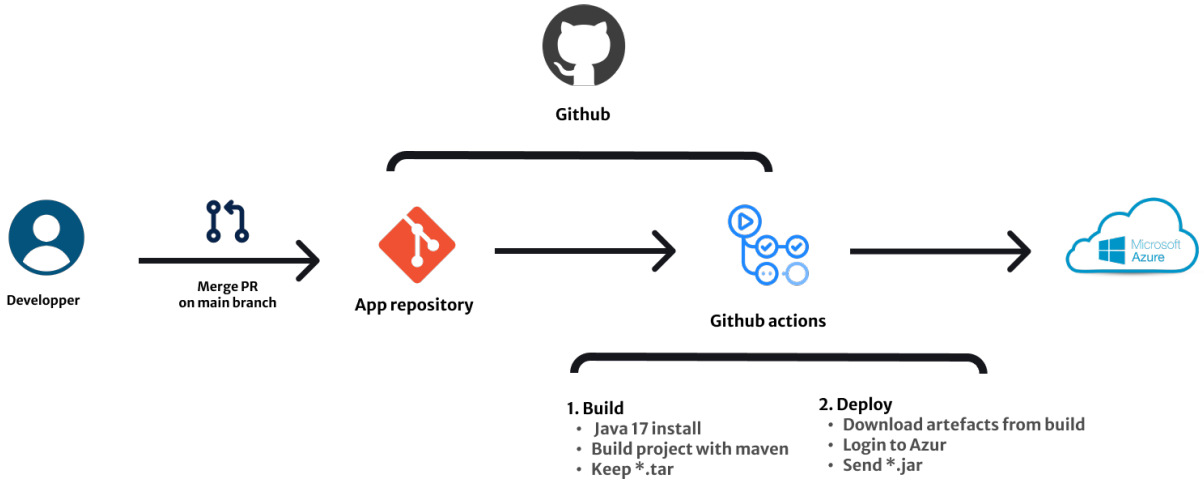


Schéma de fonctionnement du CI/CD

Pour le déploiement et l'intégration continue nous avons choisi Azure pour l'hébergement de la base de données et de notre application ainsi que les actions GitHub pour l'intégration et le déploiement en continu. Dans un premier temps nous avons utilisé Terraform pour gérer l'infrastructure sur Azure (Infrastructure as Code).

Une fois l’infrastructure Azure d’hébergement mise en place, nous avons utilisé les GitHubs Actions pour pouvoir intégrer et déployer de manière continue notre application.

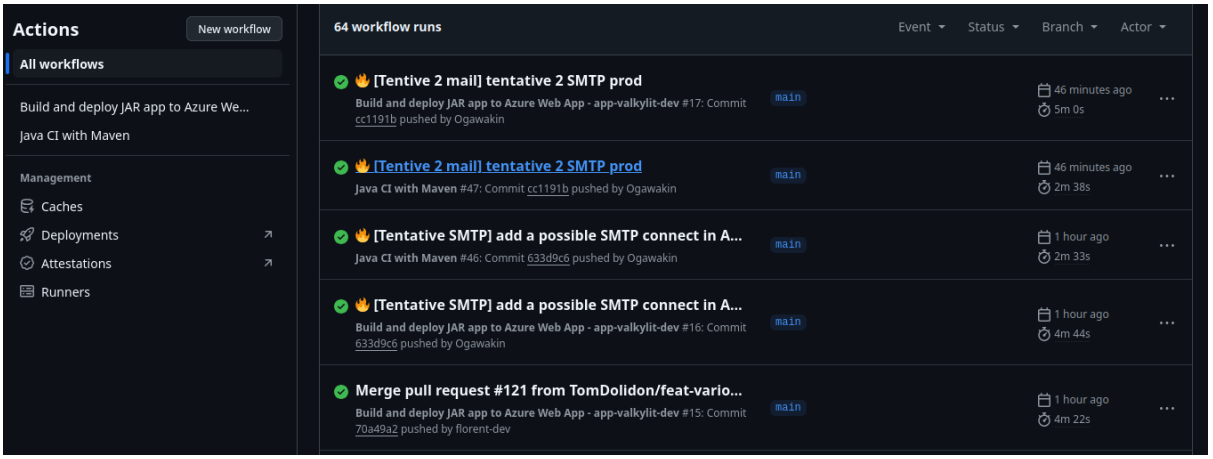
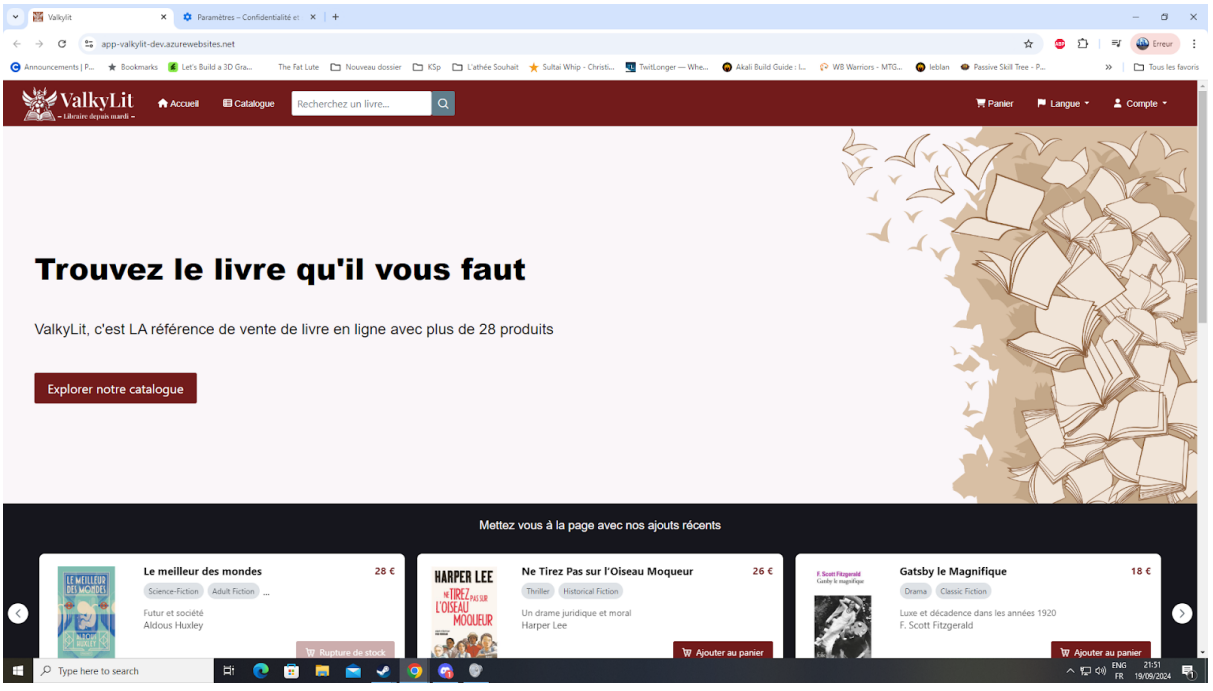
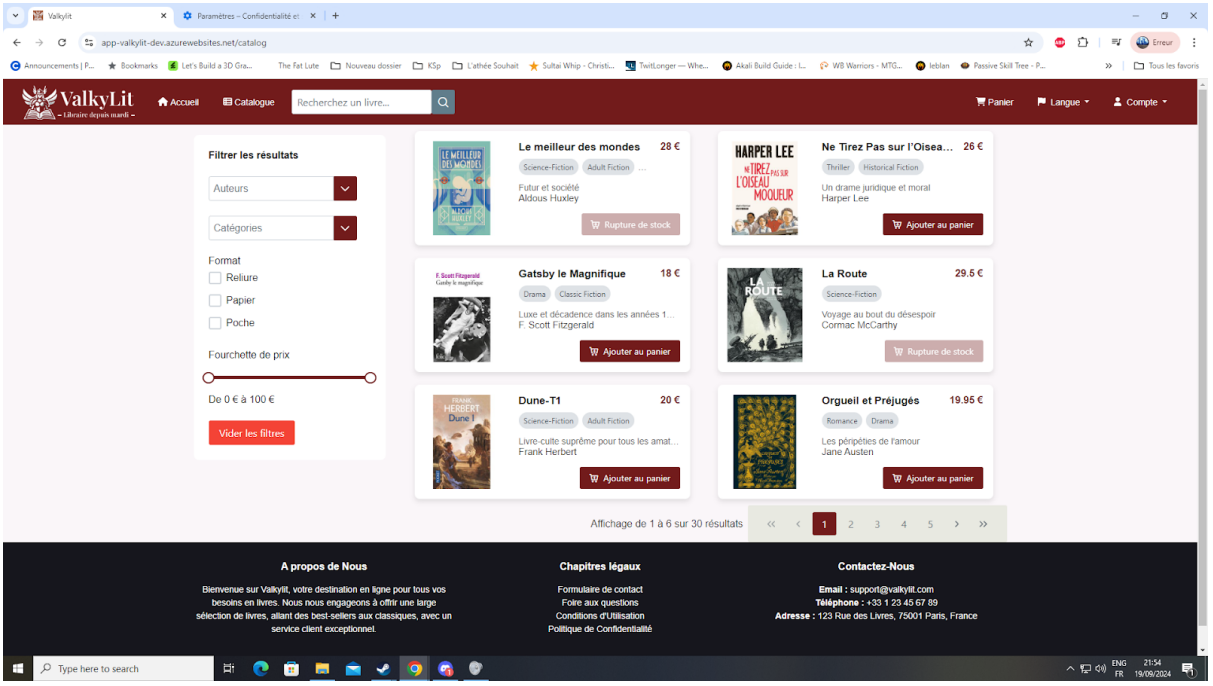


Illustration des GitHub Actions sur notre projet

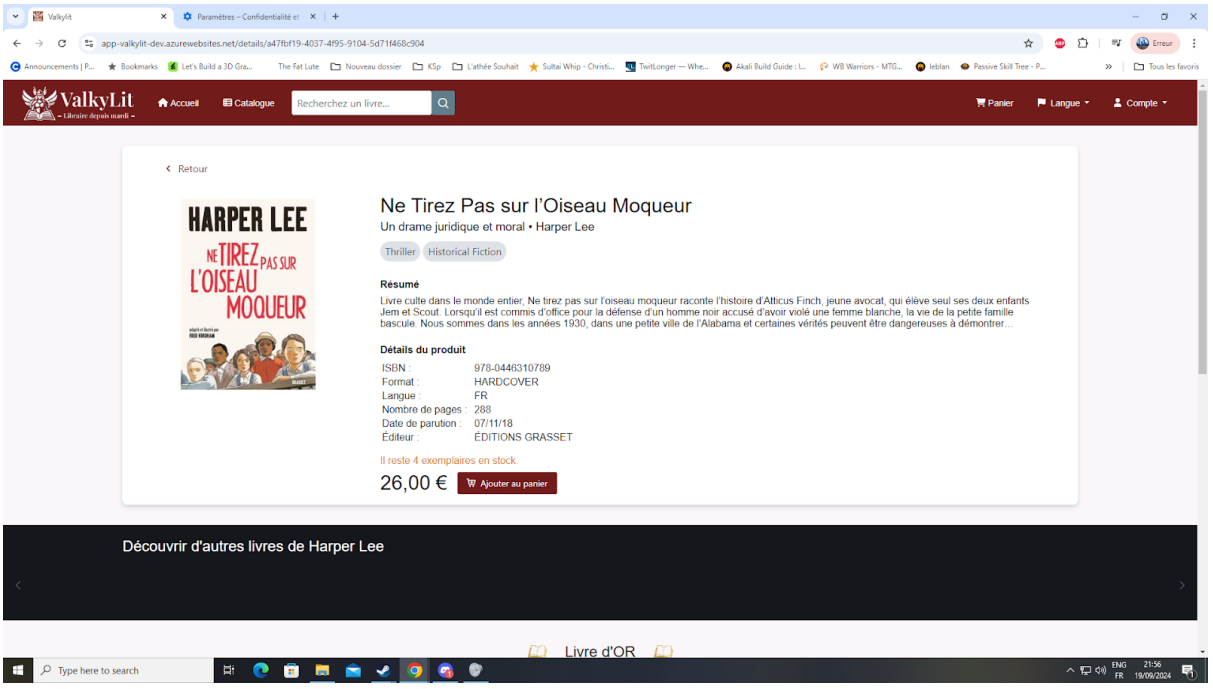
Aperçu de l’application et des fonctionnalités



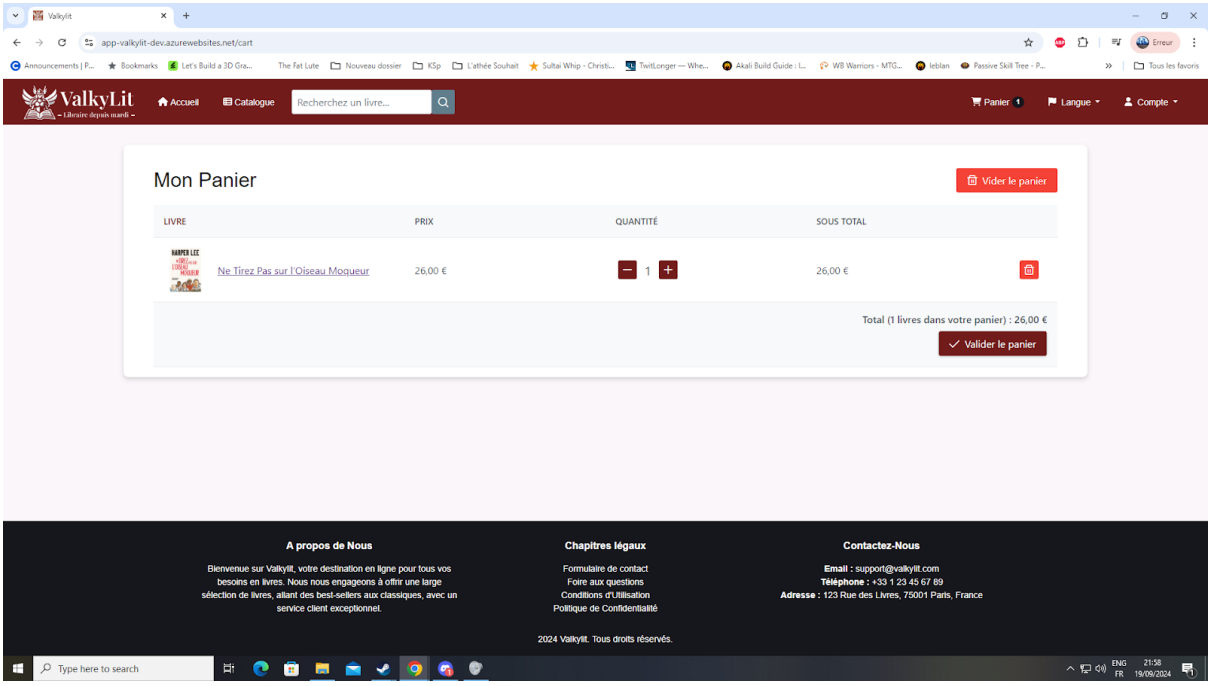
Page d'accueil de ValkyLit : *La page d’accueil regroupe l’ensemble des outils de navigation en barre de navigation, les ajouts récents, une catégorie qui met en avant les livres les mieux notés*



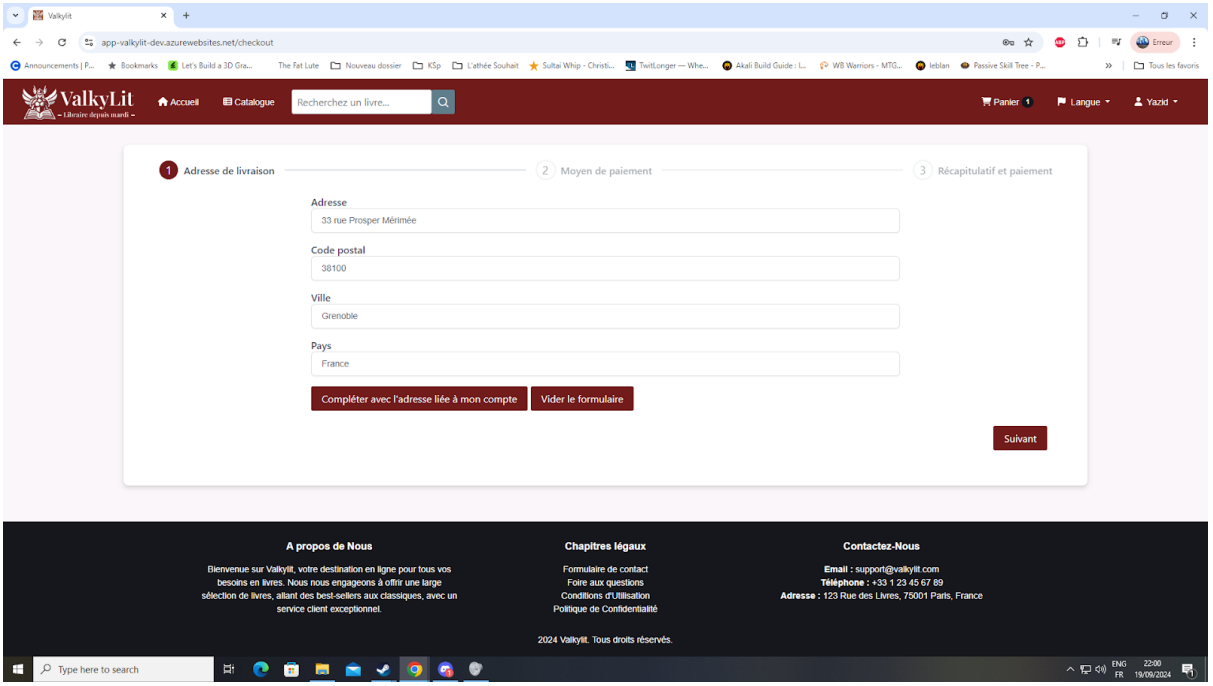
Catalogue : *Ici on retrouve sur la gauche l'ensemble des filtres pouvant faciliter la recherche, on remarque également l'impossibilité de commander des livres en rupture de stocks*



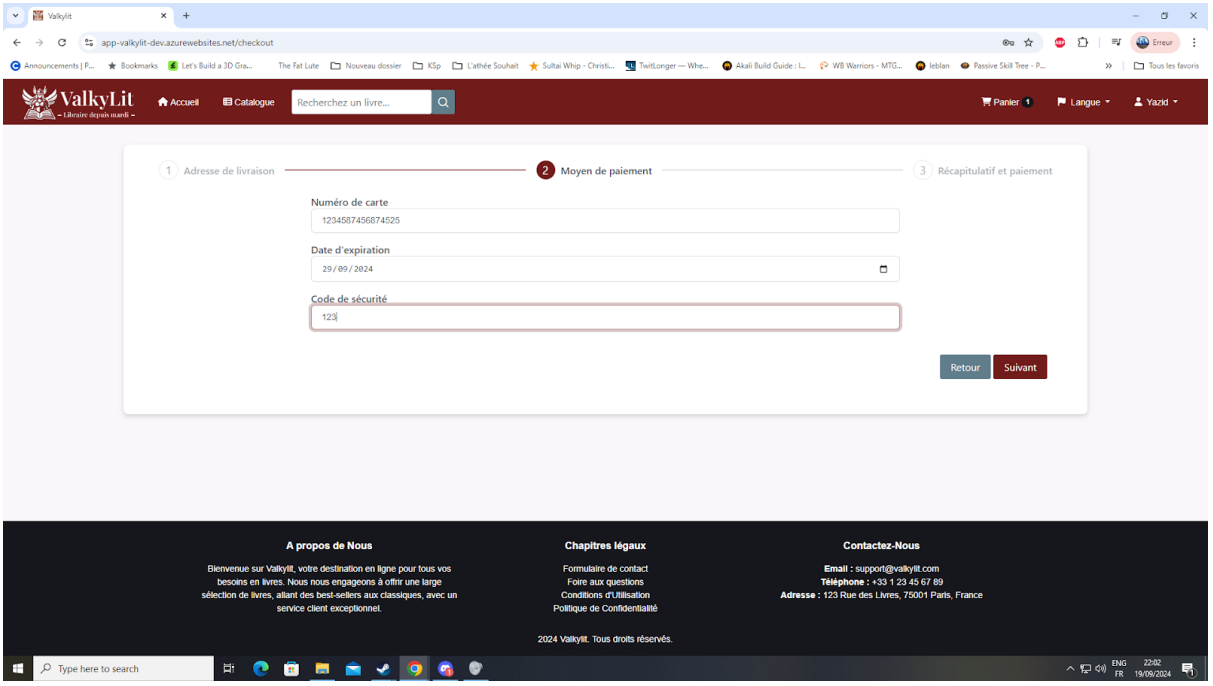
Fiche produit : *Après avoir cliqué sur un livre vous pouvez lire les détails et avis le concernant, découvrir d'autres livres du même auteur.e et cliquer sur les catégories pour poursuivre une recherche spécifique. De plus les stocks restants s'affichent avec un code couleur compréhensible*



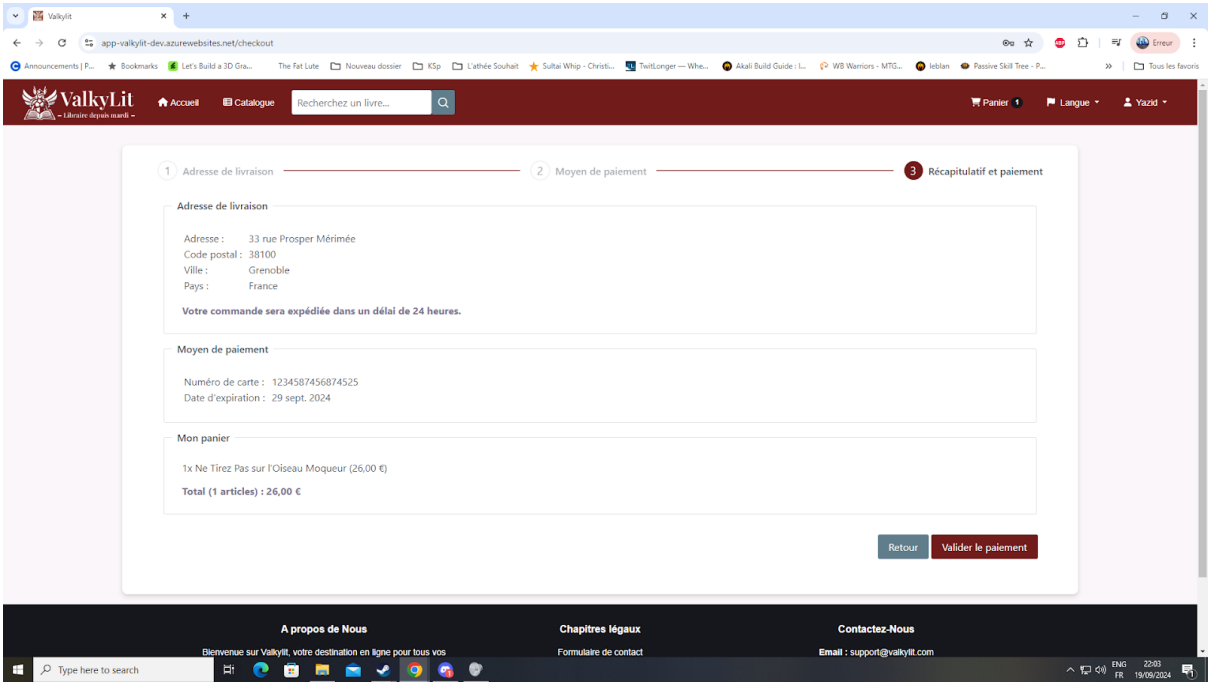
Panier : *Le panier permet à un utilisateur d'ajouter et retirer des livres à sa guise. Un popup de confirmation s'affichera pour les boutons Vider le panier, supprimer le livre ou lorsqu'un livre n'est commandé qu'une seule fois et que l'utilisateur clique sur le bouton "-". Valider le panier proposera dans un popup de s'authentifier sans changer d'interfaces afin de finaliser sa commande.*



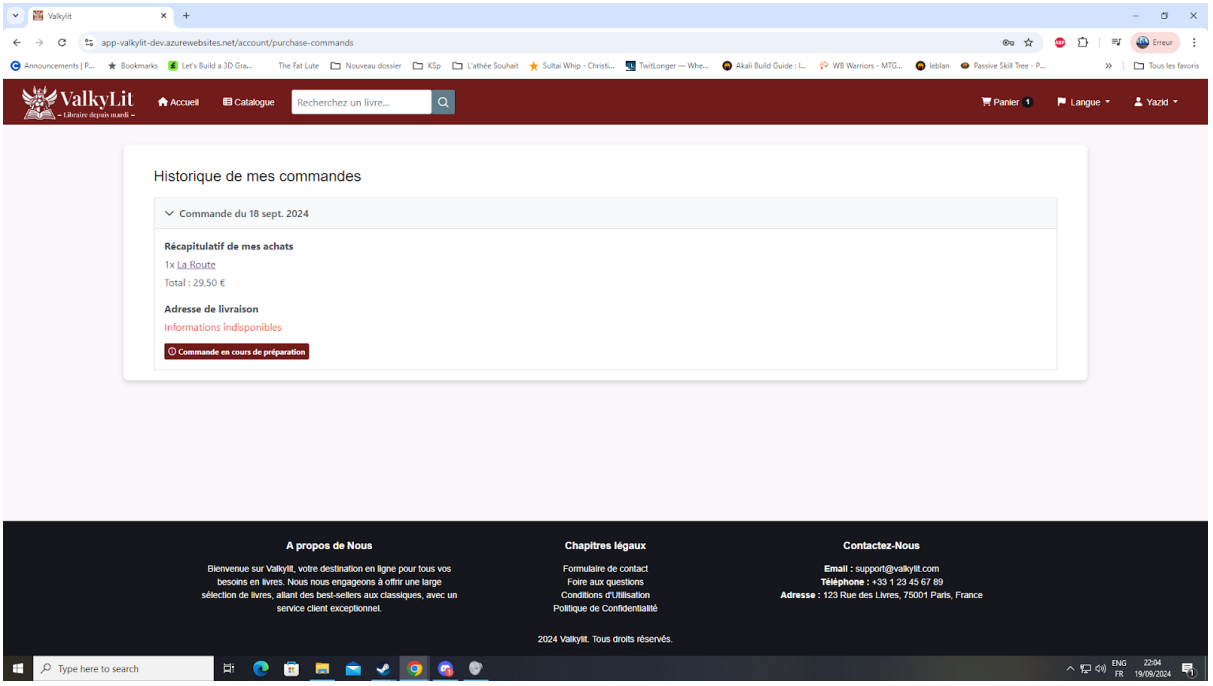
Validation 1 - Adresse : *Ici on peut remplir son adresse soit en utilisant l'adresse d'inscription soit en fournir une autre*



Validation 2 - Moyen de paiement: *Ici le client devra fournir un numéro de carte bleu à 16 chiffres avec la date d'expiration de sa carte ainsi que le code de sécurité.*



Validation 3 - Récapitulatif: *Ici nous permettons au client de visualiser une dernière fois son panier ainsi que les détails renseignés lors des précédentes étapes avant de valider la commande et le paiement.*



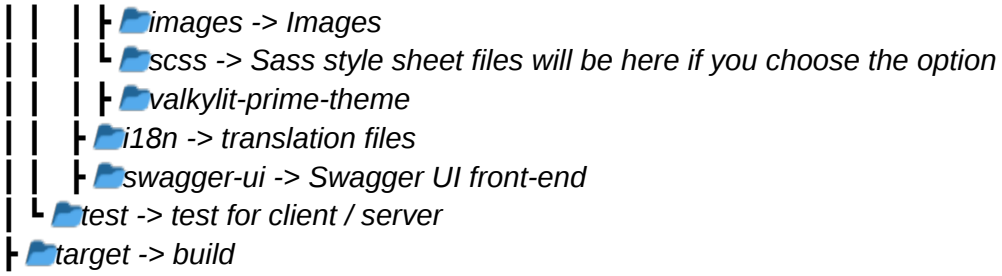
Historique des commandes : *Le client a accès à l'historique et à l'état de ses commandes, pour pouvoir les suivre et laisser des avis sur les livres lus*

Structure du projet

Ci dessous l'arborescence de la structure du projet :

```
├ .devcontainer
│ ├ devcontainer.json
│ └ Dockerfile
├ .husky -> git commit enhancement
├ .jhipster
├ .mvn
├ docs -> Document et livrables du projet
├ src
│ ├ main -> all SERVER ressources
│ │ ├ docker -> docker config for tierce services (postgress, sonar, ...)
│ │ ├ java
│ │ │ └ com
│ │ │   └ valkylit
│ │ │     └ myapp
│ │ │       ├ aop
│ │ │       │ └ logging
│ │ │       ├ config
│ │ │       ├ domain -> here are stored entities
│ │ │       ├ management
│ │ │       ├ repository -> entities repositories
│ │ │       ├ security -> security aspects (roles, jwt handling)
```

```
│ │ │     ├ service -> all backend services
│ │ │     │ ├ dto -> DTO
│ │ │     │ ├ mapper
│ │ │     ├ web
│ │ │     │ ├ filter
│ │ │     │ │ ├ package-info.java
│ │ │     │ │ └ SpaWebFilter.java
│ │ │     │ └ rest -> REST controllers
│ │ │     │   ├ errors -> custom errors
│ │ │     │   ├ vm -> view models
│ │ │     ├ ApplicationWebXml.java
│ │ │     ├ GeneratedByJHipster.java
│ │ │     ├ package-info.java
│ │ │     └ ValkylitApp.java -> entry point
│ │ ├ resources
│ │ │ ├ config -> server config
│ │ │ ├ i18n -> translations
│ │ │ ├ templates
│ │ │ ├ banner.txt
│ │ │ └ logback-spring.xml
│ │ └ webapp -> all CLIENT ressources
│ │   ├ app
│ │   │ ├ account -> User account management UI
│ │   │ ├ admin -> Administration UI
│ │   │ ├ cart -> Cart components
│ │   │ ├ catalog-> catalog components
│ │   │ ├ checkout -> checkout components
│ │   │ ├ config -> Some utilities files
│ │   │ ├ core -> Common building blocks like configuration and interceptors
│ │   │ ├ details
│ │   │ ├ entities -> Generated entities (more information below)
│ │   │ ├ home -> Home page
│ │   │ ├ layouts -> Common page layouts like navigation bar and error pages
│ │   │ │ └ main -> Main page
│ │   │ │ │ └ main.component.ts -> Main application class
│ │   │ ├ login -> Login page
│ │   │ ├ model
│ │   │ ├ pages
│ │   │ ├ shared -> Common services like authentication and internationalization
│ │   │ ├ app-page-title-strategy.ts
│ │   │ ├ app.component.ts
│ │   │ ├ app.config.ts
│ │   │ ├ app.constants.ts
│ │   │ └ app.routes.ts
│ │   ├ content -> Static content
│ │   │ ├ css -> CSS stylesheets
│ │   │ ├ favicon
│ │   │ ├ fonts/Merriweather-Sans
```


images -> Images
scss -> Sass style sheet files will be here if you choose the option
valkylit-prime-theme
i18n -> translation files
swagger-ui -> Swagger UI front-end
test -> test for client / server
target -> build