API Projects

- Sanazoo
- Trade Me
- Boissibook

Sanazoo

GitHub repo

package.json versionexpress versionsequelize versiontypescript version

SanaZoo is a very popular zoo !

First created in C with XML files, it is now developed with nodejs and swagger, for your eyes only

Our project

In this school project, we have to realize a complete API to manage a zoo, using Express and Sequelize as a base.

This project has been tested and integrated both on heroku, but also thanks to docker whose image is detailed below

Project Syllabus : Syllabus.pdf

Gantt chart

This project was carried out using a gantt chart :

image-20210424163444370.png

Data model used for DB

Here is our DB model used for this project :

Planode-Zoo.png

Contributions

Noé LARRIEU-LACOSTE	followers
Swann HERRERA	followers

Clément BOSSARD	followers

Information about code

Docker integration

Our docker image is built in 2 step :

- First we build all the project with dev dependencies
- Then we only keep production dependencies with compiled project

This reduce drastictly the size of the image

```
## Stage 1 building the code
FROM node:lts-alpine as builder
WORKDIR /usr/app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

## Stage 2 final stage with builded code
FROM node:lts-alpine
WORKDIR /usr/app
COPY package*.json ./
RUN npm ci --production

COPY --from=builder /usr/app/dist ./dist

ENV PORT=3000 \
    DB_PORT=3306 \
    DB_DRIVER='mysql' \
    DB_HOST='localhost' \
    DB_NAME='zoo' \
    DB_USER='root' \
    DB_PASSWORD=''

CMD node dist/src/index.js
```

Env

Environment variable	Default	Description
PORT	3000	Express listen port
DB_DRIVER	mysql	Driver for sql connection for sequelize
DB_HOST	localhost	Host domain / IP for DB
DB_NAME	zoo	DB Schema name
DB_USER	zoo	DB user
DB_PASSWORD	`empty`	DB password

Main dependencies

Dependency	Version	Description
Express	express version	Web API Framework
Date FNS	date-fns version	Useful librairies to manipulates dates
Dotenv	date fns version	Used to load `.env` file
Argon2	argon2 version	Used to encrupt users password
Mysql2	mysql version	DB driver
Sequelize	sequelize version	Orm librairies to bind class to DB entities
Swagger-jsdoc	version	Used to implements swagger page
Swagger-ui-express	version	Used to implements swagger page
Yup	yup version	Form validation library used to validate data in post body of our requests
Typescript	typescript version	Very useful to use types in JS based framework

API Endpoints

Postman Environment

You can check our endpoints with postman directly on this URL:

https://documenter.getpostman.com/view/11568150/TzJvdwNA

API Description :

- Affluence
  - Daily
  - Daily by enclosure
  - Live enclosure affluence
  - Monthly
  - Monthly by enclosure
  - Total
  - Total by enclosure
  - Weekly
  - Weekly by enclosure
  - 1Yearly
  - 1Yearly by enclosure
- Animal
  - Create
  - Delete
  - Get all
  - Get by id
  - Move Enclosure
  - Update
- Animal Health Book
  - Create entry
  - Delete
  - Get All
  - Get All By Animal
  - Get One
  - Update
- Enclosure
  - Add One
  - Delete One
  - Edit One
  - Get All
  - Get All Animals In Enclosure
  - Get All By Type
  - Get One

- Enclosure Images
  - Add
  - Delete One
  - Edit One
  - Gell All From Enclosure
  - Get All
  - Get One
- Enclosure Service-book
  - Create service-book
  - Delete service-book
  - Edit service-book
  - Gell All From Employee
  - Get All
  - Get All From Enclosure
  - Get One
- Enclosure Type
  - Create
  - Delete
  - Get All
  - Get One
  - Update
- Entry
  - Add entry
  - Get all
  - Get by enclosure
  - Get by pass
  - Get by user
  - Remove entry
- Maintenance
  - Get All by State
  - Get Best Month
  - Update Maintenance State
- Pass
  - Add enclosure access

    - Create
    - Delete pass
    - Get all
    - Get all by user id
    - Get by id
    - Remove enclosure access
    - Update pass
- Pass Night
    - Add availability
    - Delete passnight
    - Get all
    - Get all valid
    - Update passnight
- Pass Type
    - Add pass to pass type
    - Create
    - Delete
    - Get all
    - Get by id
    - Update
- Planning
    - Add
    - Delete
    - Get All
    - Get Calendar
    - Get One
    - Get Open Date
    - Update
- Specie
    - Create
    - Delete
    - Get All
    - Get By ID
    - Update

- Statistics
  - Count all pass
  - Count all pass by types
  - Count animal
  - Count animal by enclosure
  - Count enclosure
  - Count expired pass
  - Count user
  - Count valid pass
  - Count valid pass by types
- User
  - Change Password
  - Create
  - Delete
  - Force Delete
  - Get All
  - Get By ID
  - Login
  - Logout
  - ME
  - Register
  - Restaure User
  - Update by admin
  - Update client only by employee
- User Role
  - Affect User
  - Create
  - Delete
  - Get All
  - Get by ID
  - Update
- Swagger

Swagger

This project contain a complete swagger test environment to use API, you can access it on `https://domain.example/swagger`

It look FABULOUS :

image-20210424170307083.png

Trade Me

GitHub repo

github					
Main status	**Coverage**	**Maintainability Rati**	**Quality Gate Status**	**Reliability Rating**	**Security Rating**
		Code smells		Bugs	Vulnerabilities
		Technical Debt			

Dev status	**Coverage**	**Maintainability Rati**	**Quality Gate Status**	**Reliability Rating**	**Security Rating**
		Code smells		Bugs	Vulnerabilities
		Technical Debt			

Fonctionnalités métiers

Membres

- Ajouter / Modifier / Supprimer un **Contractor**
- Ajouter / Modifier / Supprimer un **Tradesman**
- Faire une demande de paiement pour un **Contractor**
- Faire une demande de paiement pour un **Tradesman**
- Libérer un **Tradesman** d'un **Projet**
- Assigner un **Tradesman** à un **Projet**
- Mettre à jour les attributs professionnel d'un **Tradesman**
- Mettre à jour le statut de paiement d'un **Tradesman**
- Mettre à jour le statut de paiement d'un **Contractor**
- Trouver un **Tradesman** qui match pour un **Projet**
- Récupérer un **Tradesman**
- Récupérer tout les **Tradesman**
- Récupérer les compétences d'un **Tradesman**
- Récupérer un **Contractor**
- Récupérer tout les **Contractor**

Paiement

- Payer un abonnement pour un **Contractor**

- Payer un abonnement pour un **Tradesman**

Factures

- Créer une **facture**
- Supprimer les **factures** d'un **Contractor**
- Supprimer les **factures** d'un **Tradesman**
- Récupérer toutes les **factures**
- Récupérer les **factures** des **Tradesman**
- Récupérer les **factures** des **Contractor**
- Récupérer les **factures** d'un **Tradesman**
- Récupérer les **factures** d'un **Contractor**
- Récupérer une **facture**

Projets

- Ajouter / Modifier un **Projet**
- Terminer un **Projet**
- Ajouter / Retirer un métier au **Projet**
- Ajouter / Modifier / Retirer une compétence requise à un **Projet**
- Assigner un **Tradesman** à un **Projet**
- Enlever un **Tradesman** du **Projet**
- Récupérer les **Projets**
- Récupérer un **Projet**
- Récupérer les **Projets** d'un **Contractor**
- Récupérer les **Projets** d'un **Tradesman**
- Récupérer les compétences requises d'un **Projet**

Architecture choisie

Onion Architecture

Pour mener a bien le projet l'on s'est inspiré de concept qui viennent des architectures dites **clean** et de garder ces 3 objectifs en tête.

1. Être independent du framework et des librairies externe.
2. Testable : Il doit être facile d'ajouté des tests dans la base de code.
3. Être independent de la manière dont on fait persister nos données.

Domain-Driven Design

Lors de la conception du projet on a essayé d'utiliser une approche pilotée par les usecase et le Domaine, en suivant ce que l'on connaissait du **DDD** (**Domain-Driven Design**) et fonctionnant avec des aggregates.

En utilisant ubiquitous langage, pour partager un langage comment au sein de l'équipe, qui est le langage du métier.

On a bien séparé nos features dans des **bounded context** qui partage ce qu'ils ont en commun (les **events** et certain **model interne**) via le **shared kernel**.

Staged event-driven architecture

L'**architecture événementielle par étapes** (**SEDA**) fait référence à une approche de l'architecture logicielle qui décompose le cycle de vie d'un processus en un ensemble d'étapes reliées par des files d'attente.

Il évite la surcharge élevée associée aux threads basés sur les modèles de concurrence et découple la planification des événements et des threads de la logique de l'application.

Chaque fonctionnalité de l'application est gérée par un bus d'évènement principal, qui permet de relier ces évènements à des observateurs présents dans une ou plusieurs autres fonctionnalités afin de pouvoir exécuter certaines actions secondaire.

De cette manière, nous pouvons gérer grâce à un maillage entre événement et observateurs tout le comportement de nos fonctionnalités entre elles sans que ces dernières ne communiquent jamais directement entre elle.

Un des nombreux avantages que cela représente et le découpage de notre application qui devient beaucoup plus simple, et qui se prête naturellement aux **micro services**.

Architectural Decision Record

Voilà quelques décisions d'architecture que nous avons pris pendant le développement du projet :

- Communications externes
- Communications internes
- Service VS Command/Query Handler
- Utilisation des records
- Validation

Implémentation

Dependency Inversion Principle

Le principe d'inversion des dépendances correspond au « **D** » de l'acronyme **SOLID**.

En suivant ce principe, la relation de dépendance conventionnelle que les modules de haut niveau ont, par rapport aux modules de bas niveau, est inversée dans le but de rendre les premiers indépendants des seconds.

Les deux assertions de ce principe sont :

1. Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Ce principe a été respecté pour cette application.

Command Query Separation

La **séparation commande-requête** est un principe de la programmation impérative.

Elle stipule que chaque méthode doit être une ***commande*** qui effectue une action ou une ***requête*** qui renvoie des données à l'appelant, mais pas les deux.

Plus formellement, les méthodes ne devraient retourner une valeur que si elles sont référentiellement transparentes et ne possèdent donc pas d'effets de bord.

Ce principe a été respecté au maximum au sein de l'application, même les observateurs d'événements utilisent ce principe.

Packages

Application

Le package applicatif contient le **traitement dit métier** de notre application.

Ce sont eux qui vont utiliser les différentes ressources de notre application pour **exécuter les traitements de leurs propres domaines**

Les services présents dans ce package se basent principalement sur les **interfaces** de nos autres classes afin de ne pas être dépendants d'une implémentation en particulier. On peut faire ça grâce au **polymorphisme**, la **programmation par interfaces** et le **pattern dependency injection**. Ces mêmes services sont des "micro-services" qui respecte le fameux pattern **CQS** et sont donc des **Query** / **Command** handlers.

Cas d'utilisation, création d'un contractor :

CreateContractorService_handle.png

API

Ce package est utilisé pour pouvoir avoir recours à des services externes en utilisant le **pattern strategy**.

L'interface est présente dans le package **api** et son implémentation dans **l'infrastructure**.

Pour le moment, une seule API est présente, celle du **paiement** qui peut potentiellement lancer une exception ou pas pour indiquer si la transaction s'est bien effectué. Elle est actuellement implémentée par un **stub** qui ne déclenche pas d'exception (paiement effectué).

Configuration

Ce package est celui qui permet le maillage entre toutes nos interfaces et leurs implémentations, c'est lui qui va gérer le contexte et l'injection de dépendances.

Domain

Ce package contient tous les modèles du domaine métier de notre application. C'est également celui qui contient les différentes interfaces qui peuvent être injectés dans nos services applicatifs (ex : Les interfaces des repositories).

Event

Afin de pouvoir prendre en compte **différents traitements**, sans avoir à modifier le service et que ce dernier n'ai qu'**une seule responsabilité**, on utilisera le **pattern event, observable** afin de lancer un événement lors de différentes actions menés à l'intérieur d'un service.

Ces mêmes événements seront alors pris en charge par des observables (listener) un peu partout dans le programme, qui feront eux même appel à une command / requête pour déclencher une action secondaire au traitement initial.

Les différentes tâches à exécuter suite à cet enregistrement n'auront alors qu'à **s'inscrire à cet événement** pour lancer leur propre traitement.

Cas d'utilisation, création d'une facture suite à un paiement effectué :

ContractorSubscriptionPaymentService_handle.png

NewContractorSubscriptionPaymentListener_accept.png

Model

Ce package contient toutes les entités utilisées dans l'application, elles suivent le **pattern value object** ainsi qu'**entity**. L'objet est donc immutable et possède un identifiant pour son utilisation à

travers un repository.

Exception

Contient les exceptions d'exécution du domaine métiers tel que **PaymentException** si le paiement a échoué, ** UserInvalidException** si l'utilisateur n'est pas valide et **UserNotFoundException** si l'utilisateur n'est pas présent dans le repository implémenté.

Features

Chaque fonctionnalité de notre application est séparée dans différents packages à l'intérieur du package feature.

Ces dernières ne peuvent utiliser les ressources uniquement de l'application principale, mais jamais directement entre elles. Une feature ne dépend jamais d'une autre feature.

Ces dernières reprennent chacune les différents packages (domain, infrastructure, kernel, ...) selon leurs besoins.

Actuellement, il existe 4 features :

- **Invoices** qui gère la génération et récupération des factures lors d'un paiement d'un utilisateur
- **Member** qui gère les utilisateurs de TradeMe (CRUD)
- **Payment** qui s'occupe d'effectuer les paiements lorsque cela est nécessaire.
- **Projects** qui gère les projets de l'application

Infrastructure

Repositories

Pour cela on utilise le **pattern repository et strategy** afin de **séparer son interface**, qui restera dans le ** domaine**, de son implémentation dans **l'infrastructure**.

Actuellement, il existe deux implémentations de cette interface :

- Une en mémoire : les entités en mémoire et se vide quand l'application s'arrête.
- Une grâce à un fichier JSON, qui persiste à arrêt de l'application et se charge au démarrage.

Il est possible de switcher entre l'un et l'autre grâce à la propriété `repository.in-memory=true|false` dans le fichier `application.properties`

Kernel

Ce package contient les différentes interfaces et leurs implémentations de fonctions "utilitaires" qui pourront être exploité par nos services applicatifs et autre afin d'assurer un fonctionnement correct de notre application.

IO

Ce package nous permet d'avoir les interfaces **Reader** et **Writer** qui vont nous permettre d'interagir avec différents contenus, tel que des fichiers par exemple.

Il y a actuellement deux implémentations pour chacune de ces interfaces. Les deux permettent accéder à des fichiers.

Marshaller

Le package marshaller met à disposition une interface de **Sérialisation** et de **Désérialisation** de nos objets vers une chaine de caractères.

Actuellement, il existe une implémentation de chaque pour le format **JSON**.

Exception handler

Ce package contient également des intercepteurs permettant d'intercepter les exceptions métiers lancés dans le programme afin d'en avoir un traitement centralisé pour logger l'erreur et faire un retour adapté pour l'utilisateur.

RuntimeExceptionHandler_toResponse.png

Validators

Apporte des fonctions utilitaires de validation de nos différentes entités du domaine.

Exemple : Lors de la vérification des champs, le programme jouera le diagramme de séquence suivant :

5c7cc1b9735a6d60726007287ea148f0f0509299.png

Command

Ce package contient la logique d'exécution d'une commande et le bus qui y est associé. Ce dernier possède une implémentation globale à chaque features et est injecté grâce au package configuration qui aura configuré le maillage correctement entre les commandes et les services associés.

Le comportement suivant est observé :

b03d3454f6c7f3dbeb0336faaf94ee45eb074b77.png

Création d'un contractor :

ContractorController_register-16417374095051.png

Query

Ce package contient la logique d'exécution d'une requête et le bus qui y est associé. Ce dernier possède une implémentation globale à chaque features et est injecté grâce au package configuration qui aura configuré le maillage correctement entre les requêtes et les services associés. Le comportement suivant est observé :

226848c6adb9c7b64e310525ca2fe3cc8b8c5654.png

Récupération d'un contractor :

ContractorController_getById.png

Event

Ce package est essentiel au bon déroulement de notre architecture SEDA !

Il contient la logique du bus d'événement qui permet à tous les observables de s'enregistrer à un événement. Ainsi, lorsque qu'un événement est publié, il est ensuite distribué à tous ses observateurs.

Comportement du bus d'événement par défaut :

DefaultEventBus_publish.png

Logger

En utilisant le **pattern strategy** ainsi que **factory**, ce package permet à une classe d'obtenir un logger qui lui est propre grâce au **LoggerFactory**. Les interfaces font parties du **domaine** et leurs implémentations de ** l'infrastructure**. Actuellement l'implémentation présente réutilise le la classe **Logger de Java**.

Une deuxième implémentation utilise le logger **JBoss** qui permet d'avoir des logs formatés autrement en console, en plus de les écrire dans un fichier de logs en temps réel afin de garder une trace du comportement de l'application et des éventuelles erreurs.

Shared kernel

Ce package est présent sur le niveau le plus haut de l'application, avant les **features**. Ce dernier contient les entités qui sont partagées entre les différents usecases.

Web

Ce package fournit une interface pour l'utilisateur afin qu'il puisse utiliser l'application à l'aide de requêtes REST. Ce dernier utilise uniquement les command et les requêtes à travers leurs bus associé, et n'a pas connaissance de quoi que ce soit d'autre dans l'application, ce qui lui permet d'avoir très peu de dépendance sur le fonctionnement global, hormis les entités du domaine.

Cas d'utilisation, récupération des factures :

InvoiceController_getAll.png

Quarkus

Pour gagner en puissance dans notre application et avoir des controller web ainsi qu'une **injection de dépendance** puissante, l'application est soutenue par le framework **Quarkus**.

Ce dernier sert à :

- Gérer les controller web ainsi que le sérialiseur / désérialiser JSON grâce à **Jackson**
- Configurer les différentes **bean** pour l'injection de dépendances au sein des différents services (package configuration)
- Gérer le **Scheduler** qui permet de lancer les paiements mensuels.
- Intégrer le logger **JBoss** plus facilement.
- Intégrer **Swagger** plus facilement grâce à des annotations sur les controller
- Gérer certain paramètre de l'application à la volée sans devoir recompiler le code tout le temps grâce à un fichier de configuration **application.properties** (configuration swagger, gestion du prix des abonnements, du jour de paiement, du formatage et du stockage des logs).

image-20220305155344800.png

Le découpage de l'application en amont a permis une intégration très simple de Quarkus. L'application ne dépend pas de quarkus mais utilise simplement le framework comme une implémentation de la solution.

Dependency Injection

Dans le package configuration, les beans sont réparties dans différentes classes :

GlobalConfiguration

C'est lui qui va injecter les bean dites "globales" tel que le logger ou encore la classe contenant les montants pour les abonnements.

image-20220305153432641.png

APIConfiguration

Comme son nom l'indique, inject les différents API nécessaires au bon fonctionnement de l'application, actuellement il n'y a que l'API de paiement qui est injecté, mais d'autres peuvent être amené à être créés...

APIConfiguration.png

CommandConfiguration

Injecte les différents bus de commandes selon la feature, la configuration aura fait au préalable le maillage nécessaire entre les commandes et les services applicatifs.

image-20220305153609910.png

QueryConfiguration

Injecte les différents bus de requêtes selon la feature, la configuration aura fait au préalable le maillage nécessaire entre les requêtes et les services applicatifs.

image-20220305153653788.png

EventConfiguration

Injecte les différents bus d'événements selon le type d'événement (actuellement uniquement ceux du type ** ApplicationEvent**), la configuration aura fait au préalable le maillage nécessaire entre les événements et les observateurs.

image-20220305153731926.png	EventConfiguration.png

RepositoryConfiguration

Injecte les différents repositories au sein des services qui en ont besoin :

image-20220305153814991.png

IOConfiguration

Injecte l'implémentation choisie pour nos Reader et Writer :

image-20220305154003138.png

MarshallerConfiguration

Injecte l'implémentation choisie pour notre Sérialiser et Désérialiser :

image-20220305154035698.png

Tests unitaires

Afin d'assurer le bon fonctionnement et grâce au découpage de nos composants, l'application est couverte par des tests unitaires (qui se lancent d'ailleurs automatiquement à chaque push sur github grâce à des **actions de CI**). Nous utilisons CodeCov pour analyser les rapports de tests unitaires au moment de push ou de pull requests

image-20220109160441842.png

image-20220305134206216.png

Branche DEV	Branch MAIN
codecov	codecov
grid coverage	grid coverage

Swagger

Lorsque l'application est lancée avec le profil **dev** ou **test** cette dernière rend accessible une page web swagger-ui afin de pouvoir tester directement nos interfaces REST et visualiser nos entités pour les requêtes, ainsi que ceux en réponse.

Le Swagger est configurables grâce à quelques propriétés dans le fichier `application.properties` :

image-20220305155045674.png

Le Swagger est accessible à l'adresse suivante lorsque l'application est lancée : http://localhost:8080/q/swagger-ui/#/

image-20220305154723783.png

SonarQube

Pour assurer un code propre, le code de l'application est analysé à chaque mise à jour par un serveur Sonarqube autohébergé (https://sonar.nospy.fr) pour assurer que le code rempli bien les critères de maintenabilité, sécurité, fiabilité, ...

image-20220109155624909.png

image-20220109155708591.png

image-20220109155835217.png

image-20220109155804810.png

Boissibook

Dépôts GitHub:

- API : https://github.com/Nouuu/Boissibook
- Scrapper : https://github.com/RemyMach/Boissibook-scraper
- Swift App : https://github.com/RemyMach/boissibook-swift

uality Gate Stat	Reliability Ratin	Security Rating	Technical Debt	Bugs	Code Smells	Coverage

Concept

C'est un utilitaire pour gérer sa collection de livres, à la manière d’un myanimelist, book collector.

On peut gérer sa liste de livre, ses statuts de lecture, son avancement…

Petit aspect social où l’on peut noter un livre et voir la moyenne de ce dernier donné par les différents utilisateurs. Il sera aussi possible de laisser un commentaire (publique ou pas).

Petite fonctionnalité pour pouvoir télécharger l’ebook, et l'ajouter, si on le possède, pour le partager aux autres utilisateurs (tout à fait légal, oui oui.). On pourrait également scraper quelques sites pour essayer de le trouver si on ne le possède pas grâce à un utilitaire intégré (de mieux en mieux !).

Idées de nom

- Boissibook
- Kindle surprise, Kindle bueno, maxi ... □□
- ...

Api de recherche de livres

Google Books APIs

Open Library

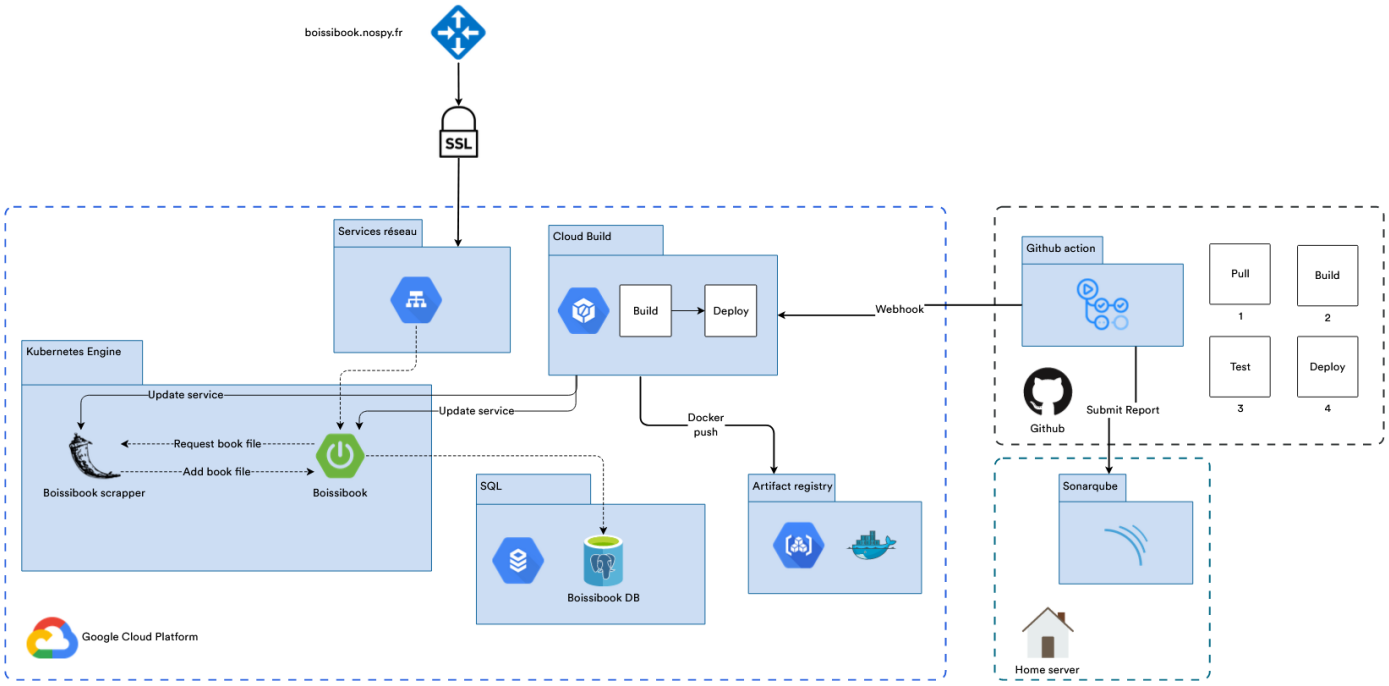
Dépôts Github

- Boissibook
- Application Swift

- Scrapper Zlib

Architecture Google Cloud Platform & CI/CD

L'application est entièrement déployée sur Google Cloud Platform, avec une infrastructure de déploiement automatique.



Features

Gestion des utilisateurs

Ce usecase est assez classique, elle permet de gérer les utilisateurs.

Un utilisateur est défini par les propriétés suivantes :

```
{
  "userId": {
    "type": "string",
    "description": "The user's id"
  },
  "email": {
```

```
    "type": "string",
    "description": "The user's email",
    "example": "gregory@mail.com"
  },
  "name": {
    "type": "string",
    "description": "The user's name",
    "example": "Gregory"
  }
}
```

Fonctionnalités

Les différentes fonctions sont les suivantes :

- Créer un utilisateur
- Modifier un utilisateur
- Supprimer un utilisateur
- Supprimer tous les utilisateurs
- Récupérer un utilisateur
- Récupérer un utilisateur par son email
- Récupérer la liste des utilisateurs
- Compter le nombre d'utilisateurs

Gestion des livres

Feature permettant de chercher un livre, l'ajouter à la base s'il n'existe pas encore et récupérer les informations de ce dernier (y compris sa note et les commentaires publics laissés par les utilisateurs).

Un livre est défini par les propriétés suivantes :

```
{
  "id": {
    "type": "string"
  },
  "title": {
    "type": "string"
  },
  "authors": {
    "type": "array",
    "items": {
```

```
      "type": "string"
    }
  },
  "publisher": {
    "type": "string"
  },
  "publishedDate": {
    "type": "string"
  },
  "description": {
    "type": "string"
  },
  "isbn13": {
    "type": "string"
  },
  "language": {
    "type": "string"
  },
  "imgUrl": {
    "type": "string"
  },
  "pages": {
    "type": "integer",
    "format": "int32"
  }
}
```

Fonctionnalités

Les différentes fonctions sont les suivantes :

- Chercher un livre en une ligne (qui pourra prendre aussi bien le nom d'un livre que celui d'un auteur, d'un genre) .
- Chercher en ligne par ISBN
- Enregistrer un livre en base (par ISBN)
- Chercher un livre en base en une ligne (qui pourra prendre aussi bien le nom d'un livre que celui d'un auteur, d'un genre).
- Supprimer un livre en base
- Récupérer les informations d'un livre en base (par ISBN)
- Récupérer les commentaires (public) d'un livre en base (par ISBN)

Readlist

Feature permettant à un utilisateur de gérer sa bibliothèque et ses livres en cours de lecture.

Une review est définie par les propriétés suivantes :


```
{
  "bookProgressionId": {
    "type": "string",
    "description": "The id of the readlist item",
    "example": "7bd1b206-833d-4378-8064-05b162d80764"
  },
  "bookId": {
    "type": "string",
    "description": "The id of the book",
    "example": "7bd1b206-833d-4378-8064-05b162d80764"
  },
  "userId": {
    "type": "string",
    "description": "The id of the user",
    "example": "7bd1b206-833d-4378-8064-05b162d80764"
  },
  "readingStatus": {
    "type": "string",
    "description": "The reading status of the book",
    "example": "READING"
  },
  "visibility": {
    "type": "string",
    "description": "The visibility of the review",
    "example": "PUBLIC"
  },
  "currentPage": {
    "type": "integer",
    "description": "The number of the current page",
    "format": "int32",
    "example": 12
  },
  "note": {
    "type": "integer",
    "description": "The note given to the book",
    "format": "int32",
```

```
    "example": 5
  },
  "comment": {
    "type": "string",
    "description": "The comment of the review",
    "example": "This book is awesome"
  }
}
```

Fonctionnalités

Les différentes fonctions sont les suivantes :

- Récupérer une review par son id
- Mettre à jour sa review sur un livre
- Supprimer sa review sur un livre
- Ajouter une review sur un livre
- Mettre à jour son statut de lecture sur un livre
- Mettre à jour sa note sur un livre
- Mettre à jour son commentaire sur un livre
- Mettre à jour sa progression sur un livre
- Récupérer toutes les reviews d’un utilisateur
- Récupérer toutes les reviews d’un livre

Téléchargement et envoie du livre

La fonctionnalité phare et tout à fait légale (□□) de Boissibook. Il est possible d'ajouter sa propre version numérique d'un livre.
Si vous ne possédez pas le livre, pas de problème ! Un autre utilisateur l'a peut être déjà ajouté à votre place. Sinon, vous pouvez demander à Boissibook de tenter de le télécharger pour vous (dans la limite du quota de 5 par jours).

Un fichier de livre est défini par les propriétés suivantes :

```
{
  "id": {
    "type": "string",
    "description": "Book file id"
  },
  "name": {
    "type": "string",
    "description": "Book file name"
  },
```

```
  "type": {
    "type": "string",
    "description": "Book file type"
  },
  "bookId": {
    "type": "string",
    "description": "Book id"
  },
  "userId": {
    "type": "string",
    "description": "User who uploaded id"
  },
  "downloadCount": {
    "type": "integer",
    "description": "File download count",
    "format": "int32"
  }
}
```

Fonctionnalités

- Ajouter sa version numérique d'un livre
- Trouver un fichier via Zlib → Scrapper Zlib
- Récupérer la liste des liens de téléchargement (ordonnée par nombre de téléchargements) d'un livre
- Récupérer le nombre de fichiers de livres disponibles pour un livre
- Supprimer un fichier livre
- Télécharger un livre

Achievements

Pour un peu plus de FUN, Boissibook propose des achievements. Ces derniers s'obtiennent lorsque vous avez terminé un certain nombre de livres ou qu'un de vos livres ajouté à la bibliothèque a été téléchargé plusieurs fois.

Scrapper Zlib

Scrapper python, utilisé par l'application Spring pour parcourir Zlib et télécharger le bouquin grâce à son nom, ISBN.

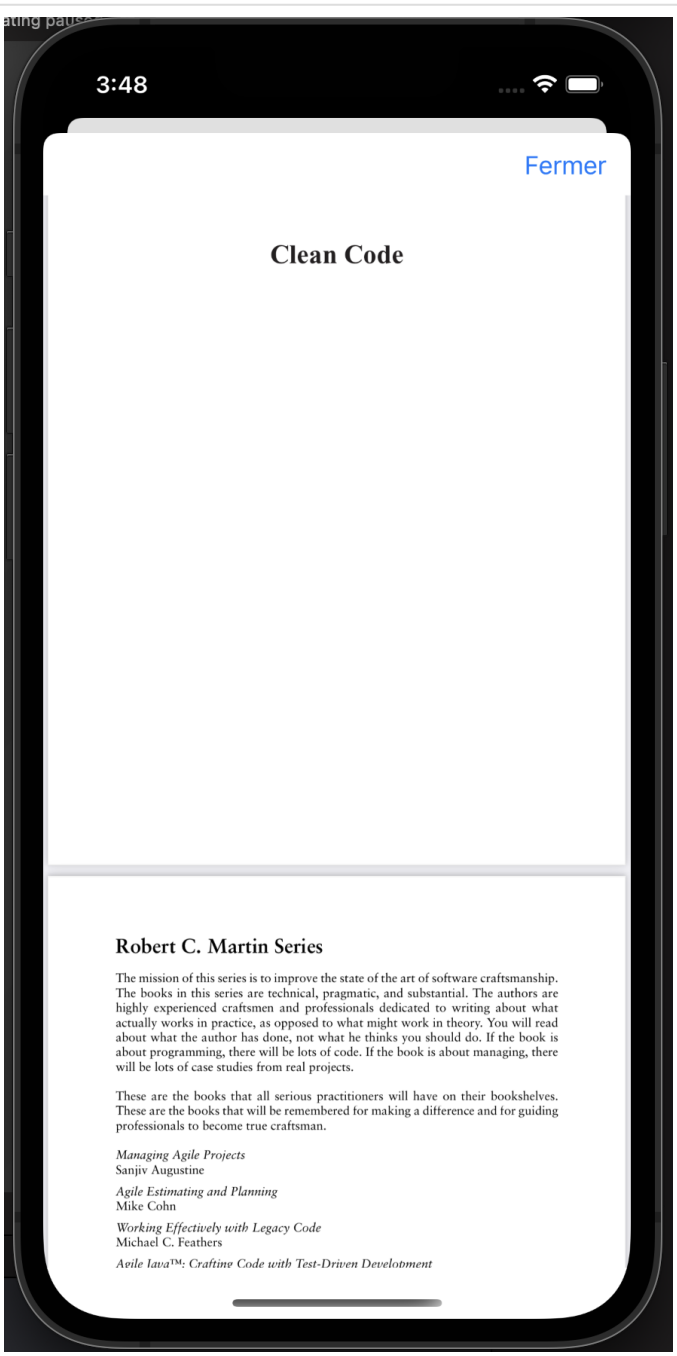
- FastAPI

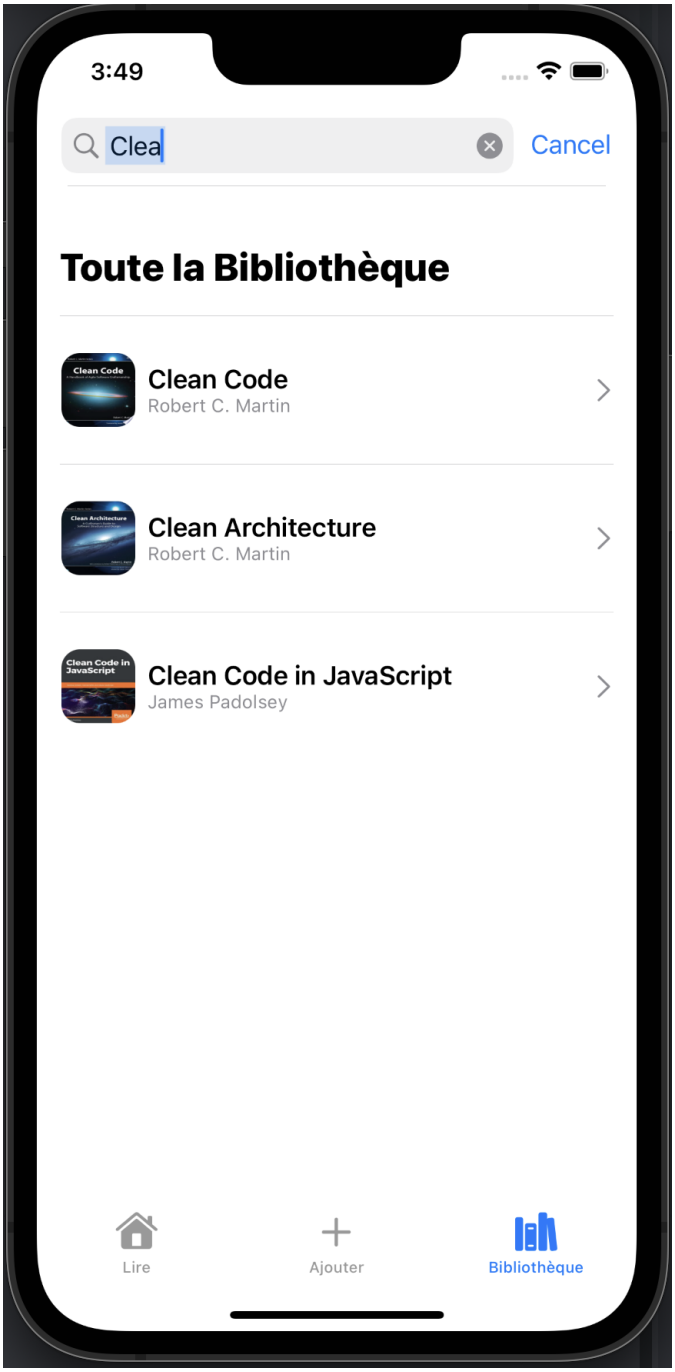
- Selenium
- Ou beautifulsoup

Une fois le fichier du livre récupéré → Téléchargement / Envoie du livre

Application frontend

Afin de pouvoir exploiter la puissance de Boissibook, nous avons développé une application IOS en Swift.





Choix d'implémentations

Hexagonal architecture

L’objectif principal de l’architecture hexagonale est de découpler la partie métier d’une application de ses services techniques. Ceci dans le but de préserver la partie métier pour qu’elle ne contienne que des éléments liés aux traitements fonctionnels. Cette architecture est aussi appelée “Ports et Adaptateurs” car l’interface entre la partie métier et l’extérieur se fait, d’une part, en utilisant les ports qui sont des interfaces définissant les entrées ou sorties et d’autre part, les adaptateurs qui sont des objets adaptant le monde extérieur à la partie métier.

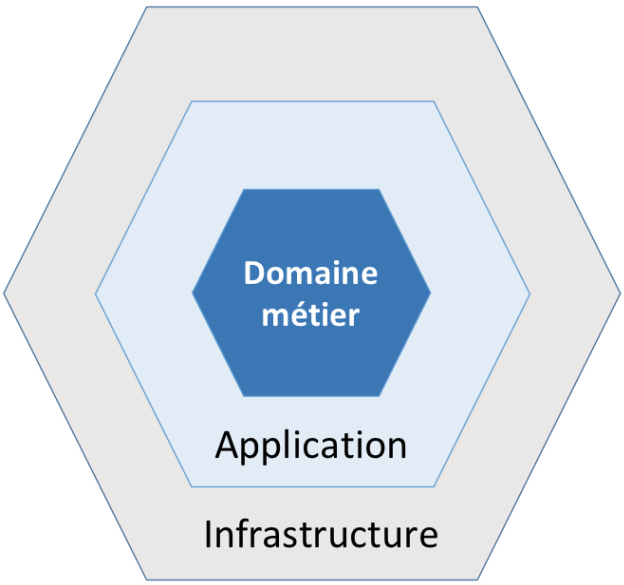
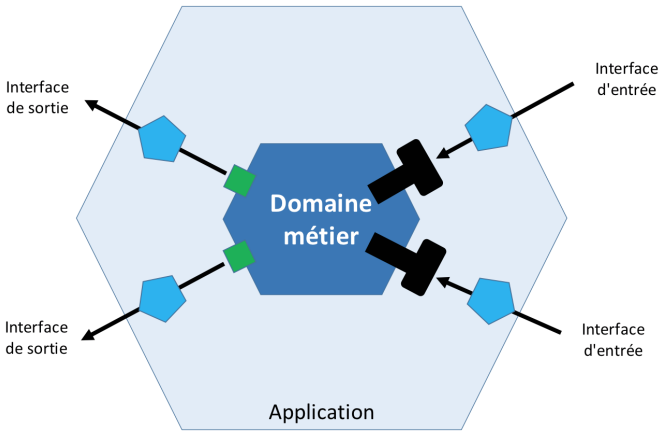
Architecture en couche

L’architecture hexagonale préconise une version simplifiée de l’architecture en couches pour séparer la logique métier des processus techniques.

La logique métier doit se trouver à l’intérieur de l’hexagone. Nous prenons plusieurs concepts en compte pour affiner cette architecture tel que :

- Inversion de dépendances
- Couche applicative
- Couche infrastructure
- ...

La couche applicative ne doit contenir que le métier de notre application, toutes ses dépendances doivent ainsi être des interfaces métiers, qui seront ensuite injectées et implémentées par la couche infrastructure.

Couches	Couche Applicative
	

Diagrammes de séquence

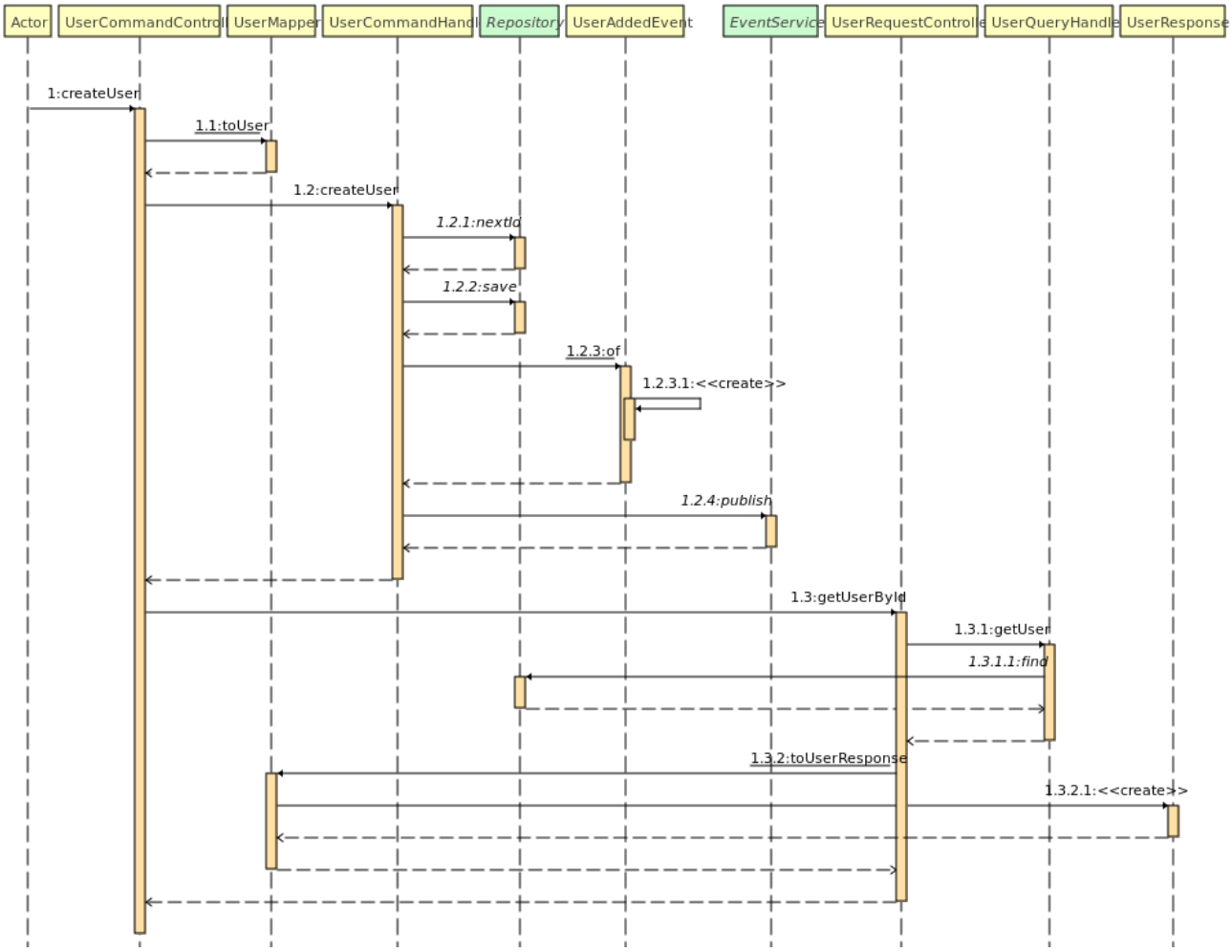
Voici quelques diagrammes de séquence montrant un workflow "Classique" de nos cas d'utilisations.

Ajout d'un utilisateur

Lors de l'ajout d'un utilisateur, plusieurs choses se déroulent :

1. Transformation de l'objet utilisateur provenant de la requête en un objet utilisateur métier.

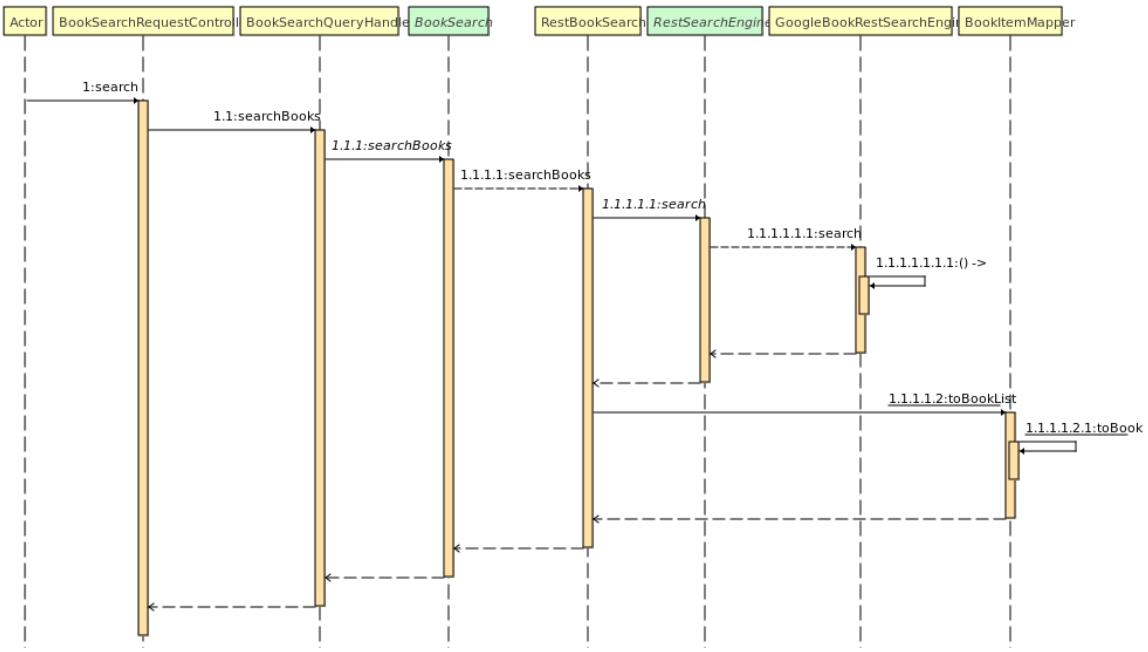
2. Appel du service métier d'enregistrement de l'utilisateur.
3. Récupération d'un nouvel ID pour l'enregistrement de l'utilisateur.
4. Enregistrement de l'utilisateur dans la base de données.
5. Envoie d'un événement de création d'utilisateur.
6. Retour au client de confirmation de l'enregistrement de l'utilisateur, avec en en-tête le lien pour consulter l'utilisateur créé.



Recherche d'un livre

Lorsque l'on cherche un livre à travers l'API (Recherche google), plusieurs choses se déroulent :

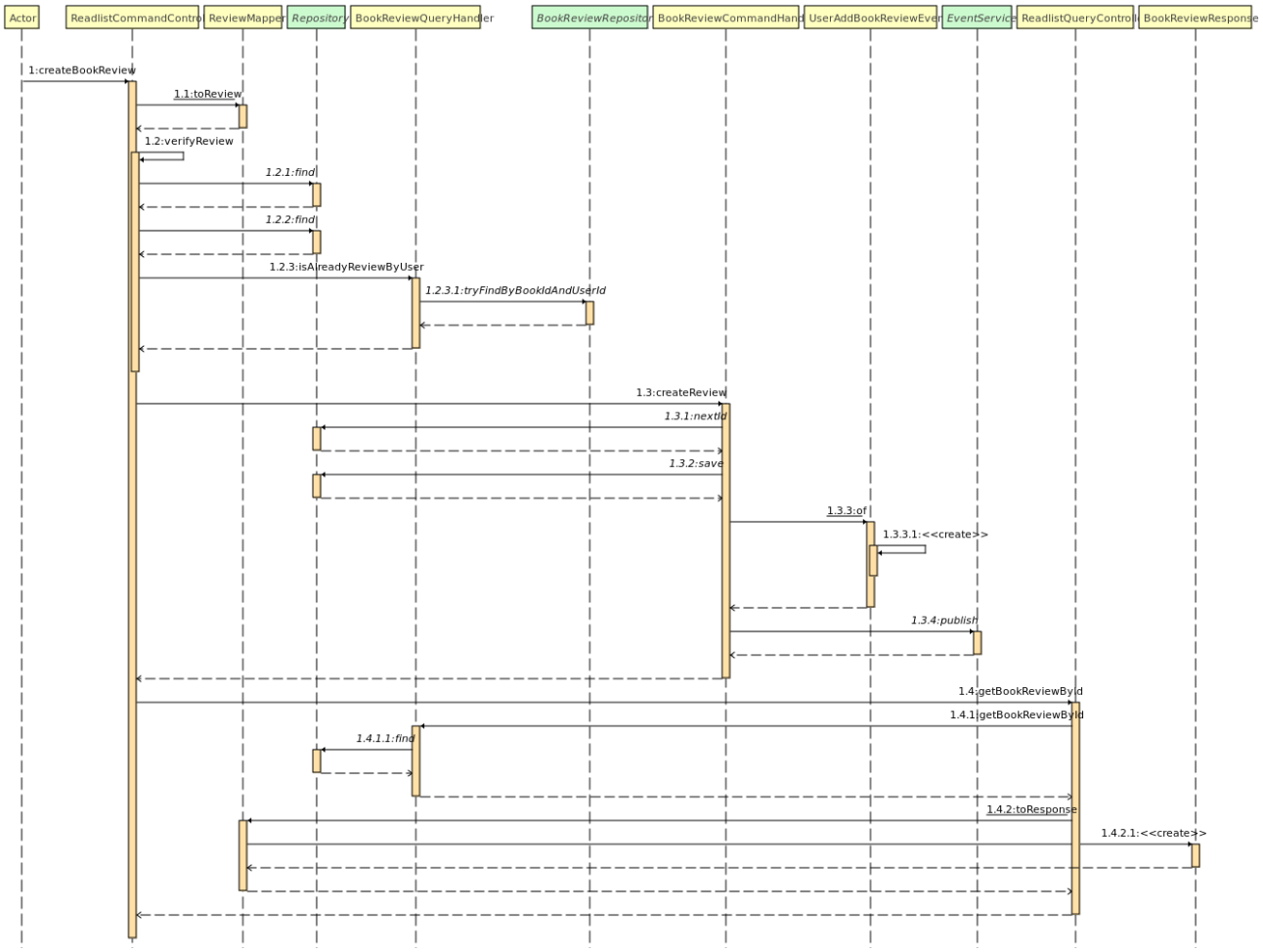
1. Récupération du terme de la recherche
2. Appel du service métier de recherche de livre.
3. Appel du moteur de recherche (en l'occurrence, celui de Google)
4. Récupération des résultats de la recherche.
5. Transformation de l'objet de résultat de la recherche en un objet de résultat de recherche métier.
6. Renvoie des résultats de la recherche au client.



Ajout d'une review sur un livre

L'ajout d'une review sur un livre se passe comme suit :

1. Transformation de l'objet review provenant de la requête en un objet review métier.
2. Vérification de l'existence du livre
3. Vérification de l'existence de l'utilisateur
4. Vérification de l'existence d'une précédente review pour cet utilisateur sur ce livre (Si c'est le cas on déclenche une erreur).
5. Appel du service métier d'enregistrement de la review.
6. Récupération d'un nouvel ID pour l'enregistrement de la review.
7. Enregistrement de la review dans la base de données.
8. Envoie d'un événement de création de review.
9. Retour au client de confirmation de l'enregistrement de la review, avec en en-tête le lien pour consulter la review créée.



Tests

Afin de garantir que notre application fonctionne correctement, nous avons mis en place plusieurs types de tests. Ces derniers sont automatiquement exécutés lorsque nous faisons un nouveau déploiement et peut interrompre ce dernier s'ils ne se valident pas tous.

Tests d'architecture

Grâce à la librairie **Arch Unit**, nous vérifions que notre application respecte les spécifications de l'architecture hexagonale.

Pour ce faire, nous allons valider trois choses :

- Les éléments du domaine ne doivent jamais importer d'éléments du framework **Spring** ou **Javax**
- Les éléments du kernel ne doivent jamais importer d'éléments du framework **Spring** ou **Javax**
- L'infra peut importer des éléments du framework **Spring** ou **Javax** ou du domaine, mais pas l'inverse.

> Exemple d'un test avec **Arch Unit**

```
class ArchitectureTest {
    @Test
    void should_domain_never_be_linked_with_frameworks() {
        var ruleNoFramework = noClasses().that().resideInAPackage("..domain..")
                .should().dependOnClassesThat().resideInAPackage("..springframework..")
                .orShould().dependOnClassesThat().resideInAPackage("javax..");

        ruleNoFramework.check(projectClasses);
    }
}
```

Tests unitaires

Nous avons décidé de mettre en place des tests unitaires pour nos classes de domaine. Nos tests unitaires sont complètement séparés du framework **Spring** et **Javax**, ce dernier n'est absolument pas présent.

> Exemple de tests unitaires sur la partie utilisateur

```
class UserCommandHandlerTest {
    // ...
    @BeforeEach
    void setUp() {
        userRepository = new InMemoryUserRepository();
        userCommandHandler = new UserCommandHandler(userRepository, new VoidEventService());
        // ...
    }

    @Test
    void createUser() {
        var userId = userCommandHandler.createUser(user1);

        assertThat(userId).isNotNull();
        assertThat(userRepository.find(userId))
                .isNotNull()
                .isEqualTo(user1.setId(userId));
```

```
    }

    @Test
    void updateUser() {
        userRepository.save(user1.setId(userRepository.nextId()));

        user1.setName("newName")
                .setPassword(null);

        userCommandHandler.updateUser(user1);
        assertThat(userRepository.find(user1.id()))
                .isEqualTo(user1.setPassword("password"));
    }
    // ...
}
```

Tests de contrat avec test container

Dans le cas de nos implémentations de nos interfaces de **Repository**, nous souhaitons tester le bon fonctionnement de nos méthodes faisant appel à la base de donnée. Pour se mettre en situation réelle, il nous faut donc une vraie base de donnée pour effectuer nos tests.

De plus, nous avons également une implémentation de nos **Repositories** en mémoire et nous devons nous assurer que cette dernière a le même comportement que la base de donnée. Ainsi nous nous assurons de ne pas avoir de comportements inattendus en changeant d'une implémentation à l'autre.

Cela nous permet également pour les autres tests unitaires de n'utiliser que la base en mémoire pour nous affranchir totalement de Spring, sans prendre le risque de passer à côté de quelque chose.

Pour ce faire, nous allons utiliser la librairie Testcontainers pour pouvoir monter à la volée un conteneur Docker d'une base de donnée entièrement dédiée aux tests.

Lorsqu'une classe de tests nécessite une base de donnée, nous allons lui faire implémenter l'interface suivante afin de lui faire monter un conteneur docker.

> PostgresIntegrationTest

```
public abstract class PostgresIntegrationTest {
    private static final PostgreSQLContainer POSTGRES_SQL_CONTAINER;
```

```
    static {
        POSTGRES_SQL_CONTAINER = new PostgreSQLContainer<>(DockerImageName.parse("postgres:14-
alpine"));
        POSTGRES_SQL_CONTAINER.start();
    }

    @DynamicPropertySource
    static void overrideTestProperties(DynamicPropertyRegistry registry) {
        registry.add("spring.datasource.url", POSTGRES_SQL_CONTAINER::getJdbcUrl);
        registry.add("spring.datasource.username", POSTGRES_SQL_CONTAINER::getUsername);
        registry.add("spring.datasource.password", POSTGRES_SQL_CONTAINER::getPassword);
    }
}
```

Cette dernière va venir surcharger les paramètres Spring pour lui faire se connecter à la base de donnée automatiquement.

> UserRepositoryTest

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)
@Testcontainers
@SpringBootTest
@ActiveProfiles("test")
class UserRepositoryTest extends PostgresIntegrationTest {

    @Autowired
    JPAUserRepository jpaUserRepository;

    private final static String springDataUserRepositoryKey = "SpringDataUserRepository";

    private final static String inMemoryUserRepositoryKey = "InMemoryUserRepository";

    private HashMap<String, UserRepository> userRepositories;

    // ...

    @BeforeEach
```

```
    void setUp() {
        SpringDataUserRepository userRepository = new
SpringDataUserRepository(jpaUserRepository);
        InMemoryUserRepository inMemoryUserRepository = new InMemoryUserRepository();

        userRepositories = new HashMap<>();
        userRepositories.put(springDataUserRepositoryKey, userRepository);
        userRepositories.put(inMemoryUserRepositoryKey, inMemoryUserRepository);

        // ...
    }

    private static Stream<String> provideRepositories() {
        return Stream.of(
                springDataUserRepositoryKey,
                inMemoryUserRepositoryKey
        );
    }

    @ParameterizedTest
    @MethodSource("provideRepositories")
    void save(String userRepositoryKey) {
        UserRepository userRepository = userRepositories.get(userRepositoryKey);

        userRepository.save(user1);

        assertThat(userRepository.find(user1.id()))
                .isEqualTo(user1);
    }
}
```

Grâce aux `ParameterizedTest`, nous allons jouer les mêmes tests aussi bien sur la base de donnée réelle que celle en mémoire, afin de nous assurer que chacune valide exactement les mêmes tests.

Tests E2E

Afin de pouvoir tester les fonctionnalités de notre application, nous devons tester que l'application fonctionne de bout en bout avec un cas d'utilisation réel. Il faut alors lancer l'application avec tout le context **Spring**, ainsi qu'avec **Testcontainers** pour avoir un comportement réel. Nous nous servons ensuite de la librairie RestAssured pour faire des requêtes sur l'API afin de s'assurer que le comportement est bien celui attendu.

> UserCommandsAPITest

```
@DirtiesContext(classMode = BEFORE_EACH_TEST_METHOD)
@Testcontainers
@SpringBootTest(webEnvironment = RANDOM_PORT)
@ActiveProfiles("test")
class UserCommandsAPITest extends PostgresIntegrationTest {
    //...
    @LocalServerPort
    int port;

    @BeforeEach
    void setUp() {
        RestAssured.port = port;
        RestAssured.filters(new RequestLoggingFilter(), new ResponseLoggingFilter());
    }

    @Test
    void createUser() {

        var getUserUri = given()
                .contentType(JSON)
                .body(validUser1)
                .when()
                .post("/users")
                .then()
                .statusCode(201)
                .extract()
                .header("location");

        var user = given()
                .baseUri(getUserUri)
                .when()
                .get()
                .then().statusCode(200)
                .extract()
                .body().as(UserResponse.class);
```

```
        assertThat(user.userId()).isNotNull();
        assertThat(user.email()).isEqualTo(validUser1.email());
        assertThat(user.name()).isEqualTo(validUser1.name());
    }
    //...
}
```