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
  - Idées de nom
  - Api de recherche de livres
- Dépôts Github
- Architecture Google Cloud Platform & CI/CD
- Features
  - Gestion des utilisateurs
    - Fonctionnalités
  - Gestion des livres
    - Fonctionnalités
  - Readlist
    - Fonctionnalités
  - Téléchargement et envoie du livre
    - Fonctionnalités
  - Achievements
  - Scrapper Zlib
- Application frontend
- Choix d'implémentations
  - Hexagonal architecture
    - Architecture en couche
  - Diagrammes de séquence
    - Ajout d'un utilisateur
    - Recherche d'un livre

    - Ajout d'une review sur un livre
  - Tests
    - Tests d'architecture
    - Tests unitaires
    - Tests de contrat avec test container
    - Tests E2E

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

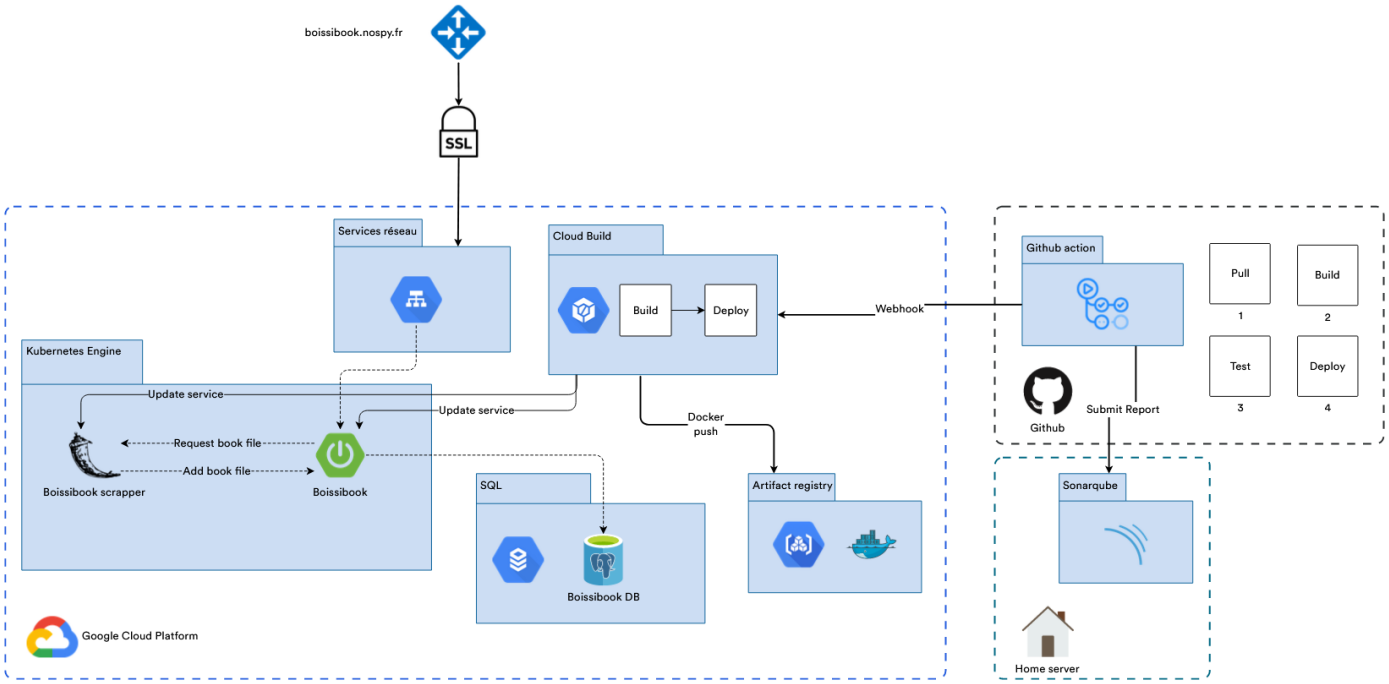
Dépôts Github

- 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

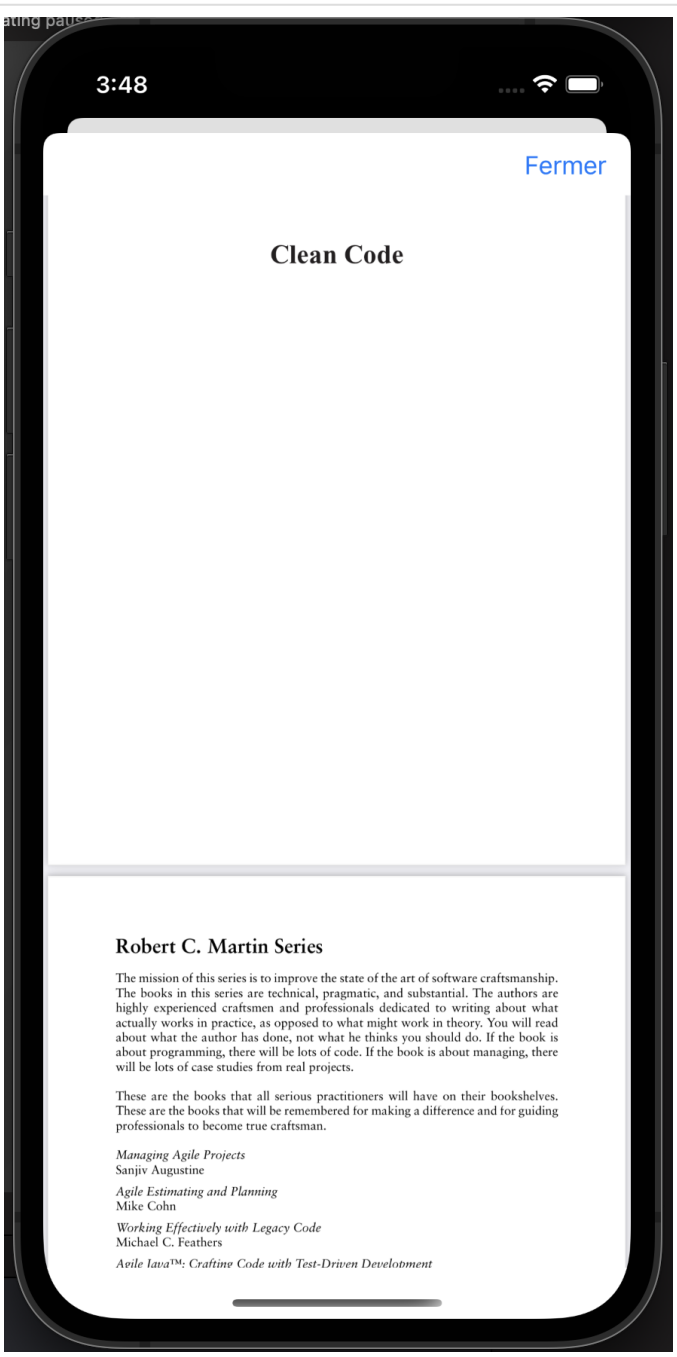
Une fois le fichier du livre récupéré → Téléchargement / Envoie du livre

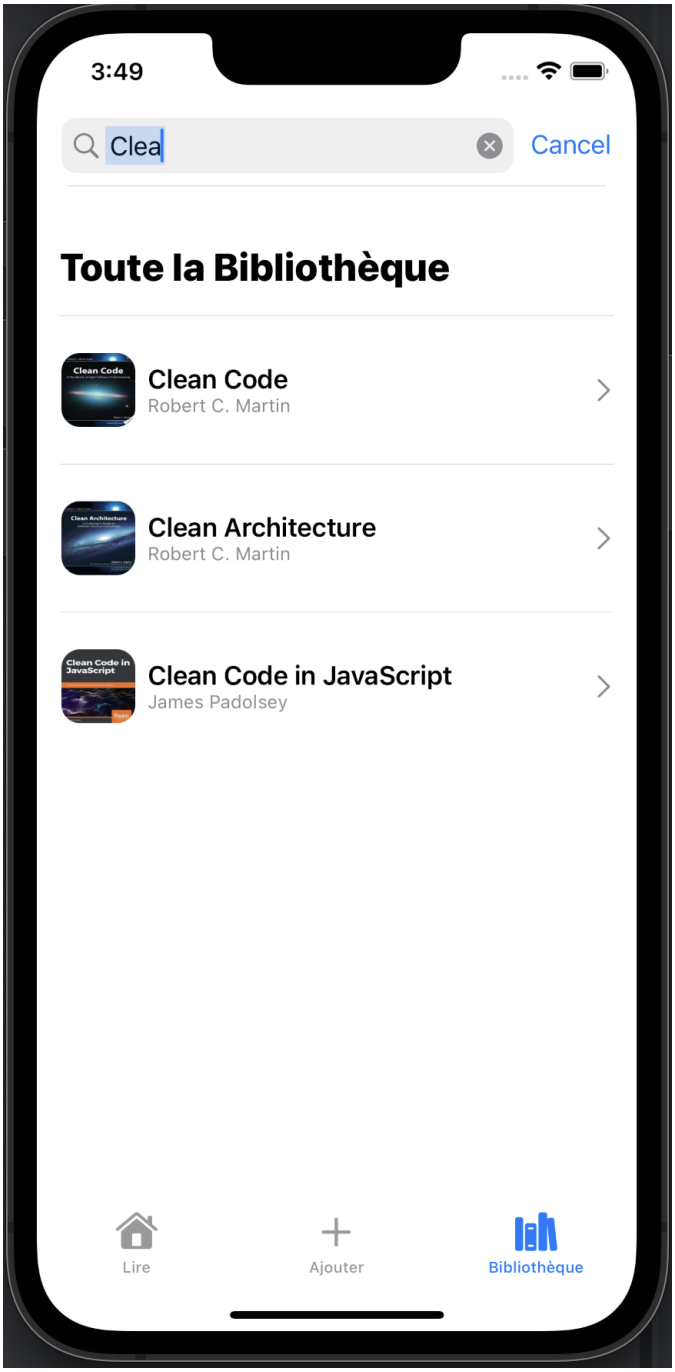
Application frontend

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


3:33
Lire.
Lire, c'est boire et manger. L'esprit qui ne lit pas maigrit comme le corps qui ne mange pas.
Ouvrir


3:47
Mes livres
Récents
Clean Code
A Handbook of Agile Software Craftsmanship
Clean Code
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Clean Architecture
Livres disponibles
Clean Code
Clean Architecture
Lire
Ajouter
Bibliothèque





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.

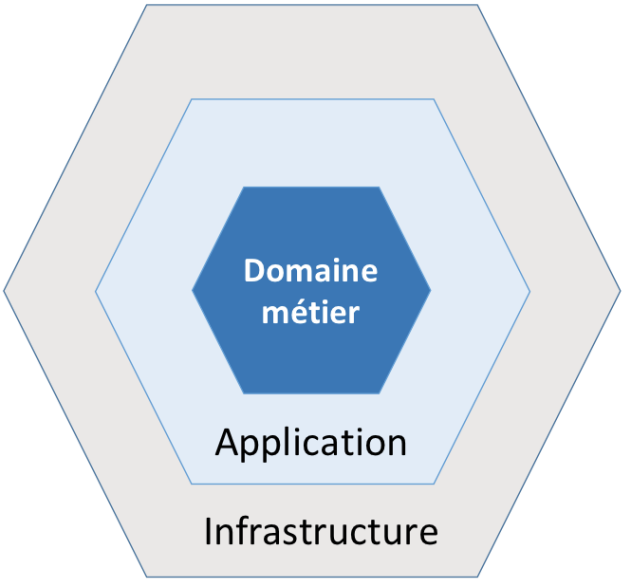
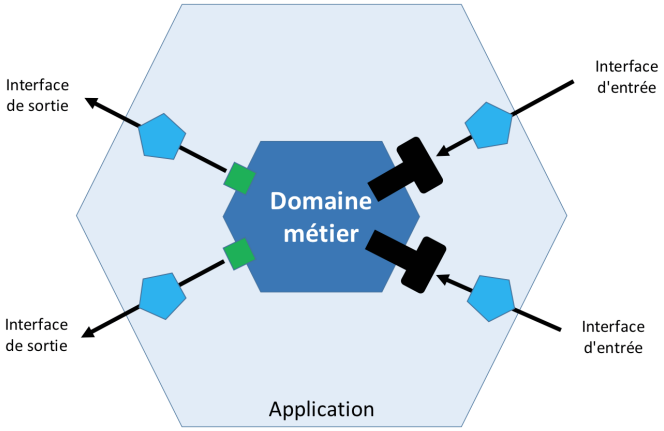
Architecture en couche

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
	

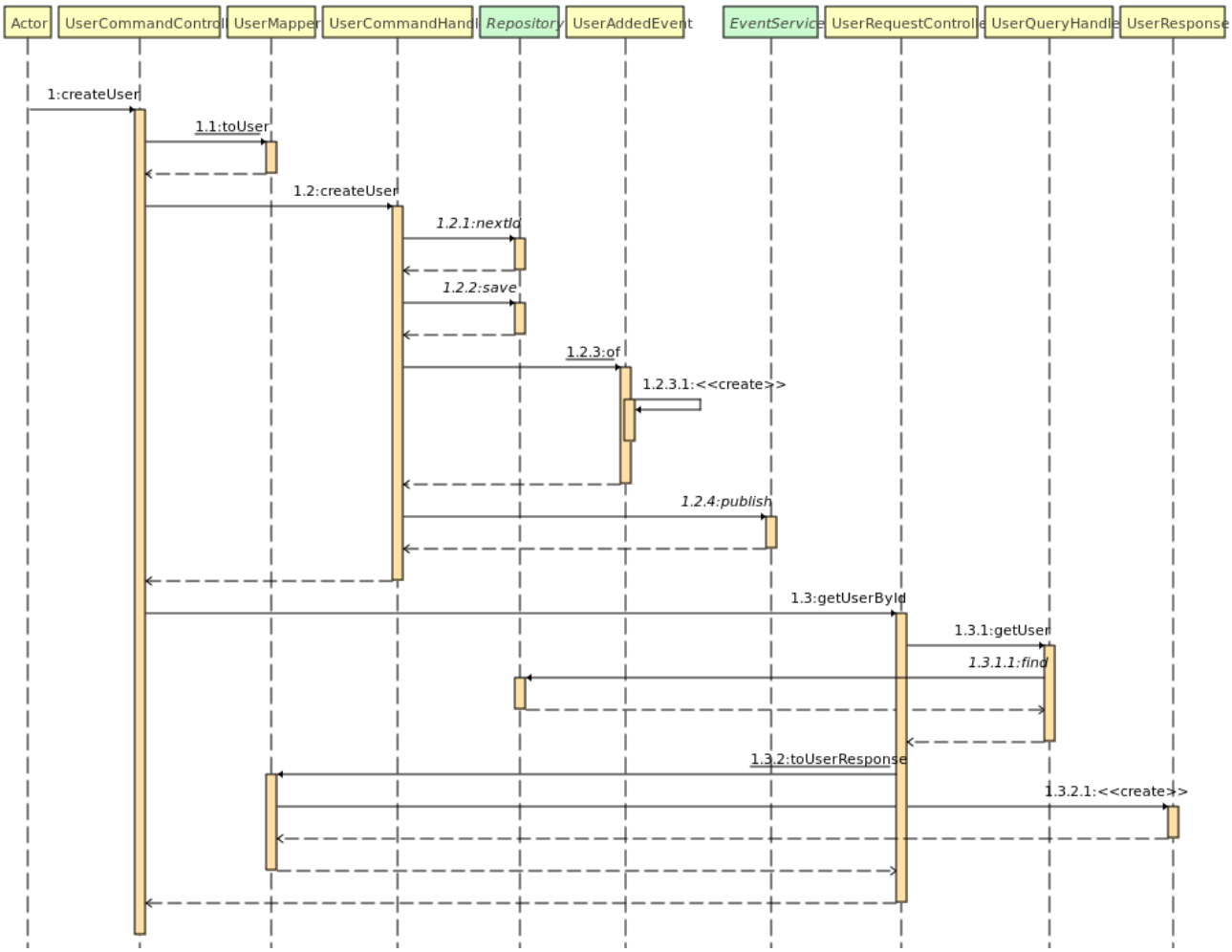
Diagrammes de séquence

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 :

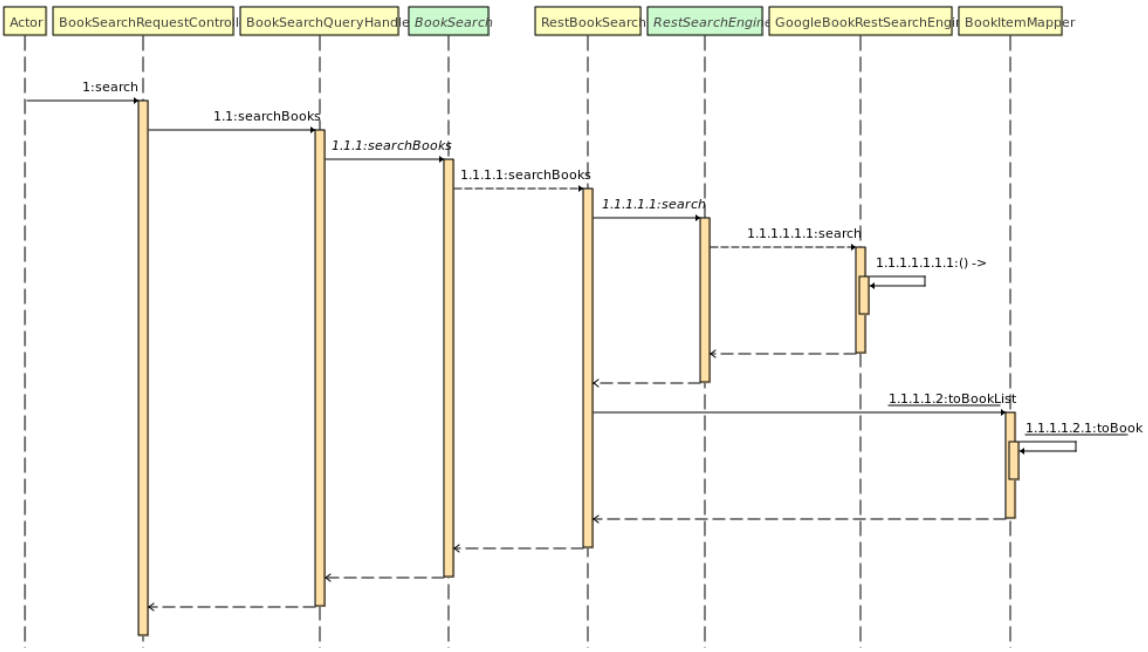
1. Transformation de l'objet utilisateur provenant de la requête en un objet utilisateur métier.
2. Appel du service métier d'enregistrement de l'utilisateur.
3. Récupération d'un nouvel ID pour l'enregistrement de l'utilisateur.
4. Enregistrement de l'utilisateur dans la base de données.
5. Envoie d'un 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 :

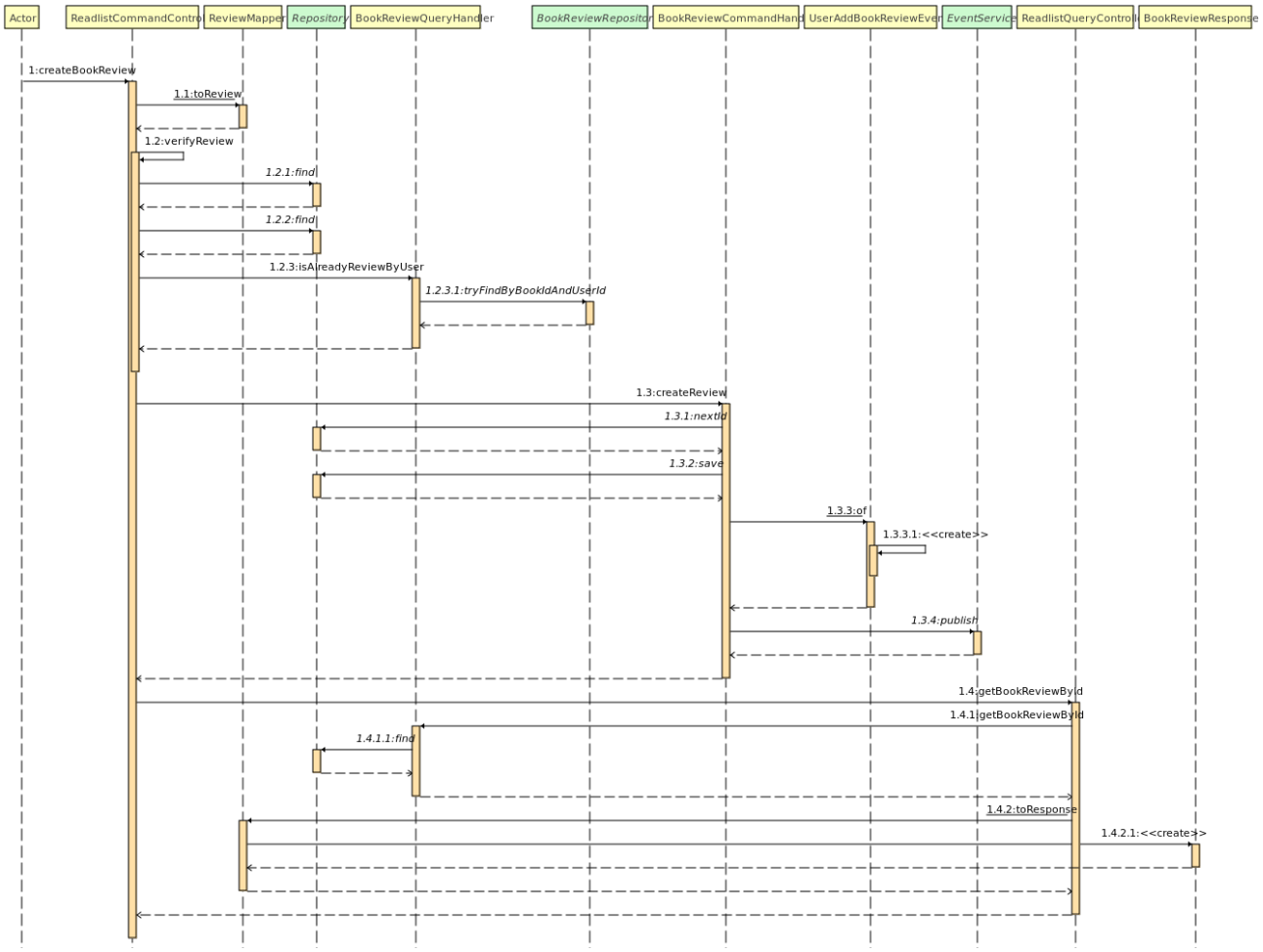
1. Récupération du terme de la recherche
2. Appel du service métier de recherche de livre.
3. Appel du moteur de recherche (en l'occurrence, celui de Google)
4. Récupération des résultats de la recherche.
5. Transformation de l'objet de résultat de la recherche en un objet de résultat de recherche métier.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

Revision #1
Created 2 October 2022 15:10:50 by Noé Larrieu-Lacoste
Updated 2 October 2022 15:20:24 by Noé Larrieu-Lacoste