Dauphine | PSL
UNIVERSITÉ PARIS

Rapport

Applications web oritentées services

Auteurs:
Hocine HAMI, Mohamed LEBIB.

31 Mai 2020

Table des matières

1 Diagramme des classes

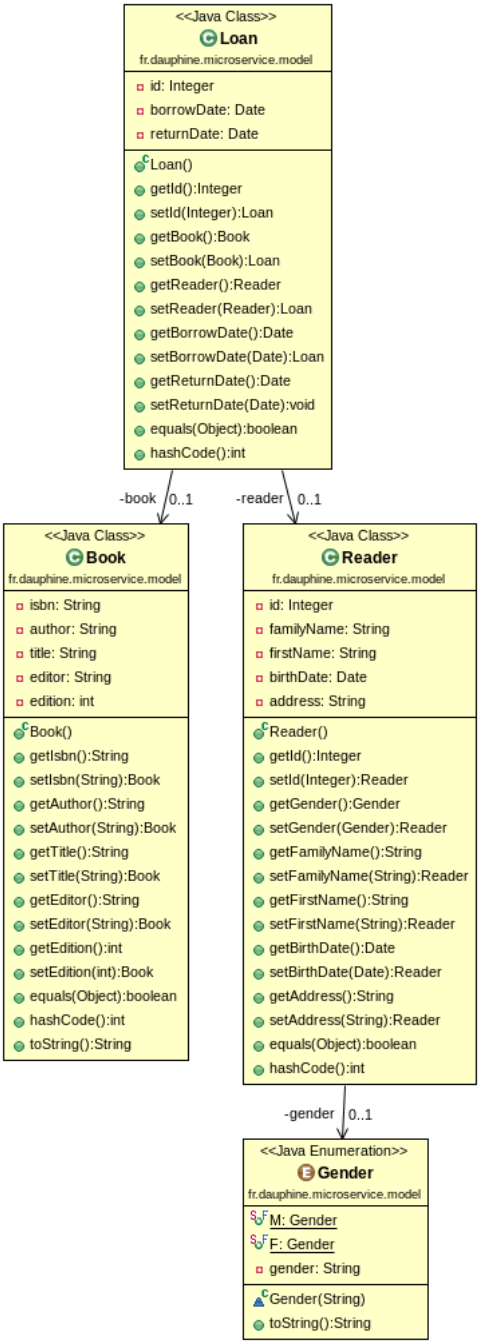

Figure 1: Diagramme de classes "métier" de l'application obtenu à partir d'objectAid sur eclipse.

- **Classe « Reader » :** Cette classe modélise un lecteur. Il est identifié de façon unique par un entier naturel (qui s'auto-incrémente à chaque création d'un nouveau lecteur).
- **Énumération « Gender » :** Qui modélise le genre (sexe) du lecteur.
- **Classe « Book » :** Cette classe modélise un livre. Celui-ci est identifié de façon unique par son ISBN, qui est renseigné lors de sa création.
- **Classe « Loan » :** Cette classe modélise un prêt. Chaque prêt est identifié de façon unique par un entier naturel (qui s'auto-incrémente à chaque création d'un nouveau prêt).
  Une date de retour à *null* signifie que le livre n'a pas encore été retourné. Toutes les dates sont au format yyyy-MM-dd.
  Cette classe possède une association avec les deux classes Book et Reader. En effet, un prêt concerne un livre et un lecteur.

2 Architecture de l'application

2.1 Découpage sous forme de microservices

Afin de découper le projet en microservices, nous nous sommes basés sur un critère, qui est celui de l'interdépendance des différents services. La création d'un prêt a besoin, au préalable, de connaître le lecteur et le livre associés. Le service des prêts devra donc interroger le service des lecteurs ainsi que celui des livres afin de récupérer ces informations. Par ailleurs, nous savons que :

- Un lecteur peut exister même s'il n'effectue aucun prêt. Prenons comme exemple une bibliothèque universitaire. Un étudiant existe, et est identifié de façon unique par son numéro étudiant, même s'il n'emprunte jamais de livres. Le service de prêt de la bibliothèque récupère les informations concernant un étudiant auprès d'un serveur d'authentification centralisé, par exemple, et ce dernier est indépendant des autres services.
- Une bibliothèque permet à ses étudiants de consulter les livres disponibles en rayon, même si le système de prêts est hors service. Nous devons donc être en mesure de consulter le catalogue des livres, indépendamment de tout autre service.

En prenant en compte ces détails, il devient clair que nous aurons trois microservices à mettre en place :

- Microservice « Reader » : Ce microservice se charge de la création, de la récupération, de la modification et de la suppression d'un lecteur.
- Microservice « Book » : Microservice des livres. Il s'occupe de créer, récupérer, modifier et supprimer des livres.
- Microservice « Loan » : Microservice des prêts, il se charge de la création, de la modification et de la récupération des prêts suivants certains critères. Il interroge les deux microservices « Reader » et « Book » afin de récupérer le lecteur et le livre dont il est question pour le prêt.

Le schéma ci-dessous illustre nos trois microservices et leur interaction.

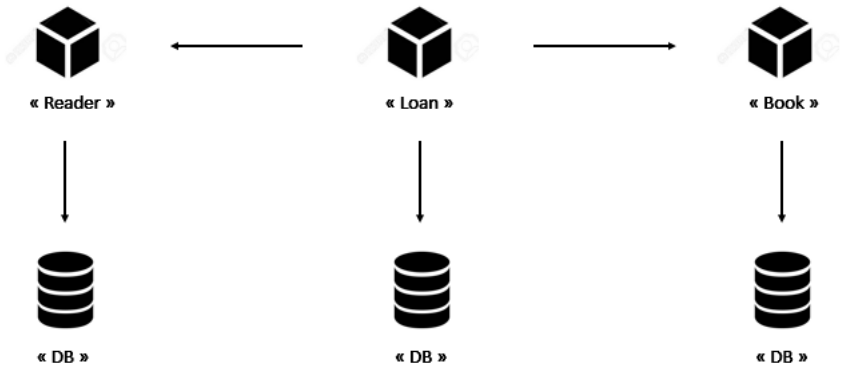

Figure 2: Schéma d'interaction entre les trois microservices.

2.2 Définition des APIs REST

Chaque microservice possèdera une API REST. Celle-ci contiendra les différents endpoints permettant d'effectuer nos actions de création, de récupération, de modification et de suppression. Ces APIs sont de niveau 3 dans le modèle de Maturité de Richardson (HATEOAS). Chacune des APIs est préfixée par « /api » (Context-Path).

1. **Reader API**
    - HealthCheck « GET /healthcheck » : Endpoint permettant de vérifier « la santé » de l'API, c'est-à-dire qu'elle est bien déployée.
    - Création d'un lecteur « POST /readers » : Les informations sur le lecteur sont renseignées dans le corps de la requête HTTP (RequestBody).

- Récupération d'un lecteur « GET /readers/id » : Permet de récupérer un lecteur en faisant une sélection par identifiant.
- Mise à jour d'un lecteur « PUT /readers » : Permet de mettre à jour partiellement les informations d'un lecteur (sauf son identifiant, bien entendu).
- Suppression d'un lecteur « DELETE /readers/id »

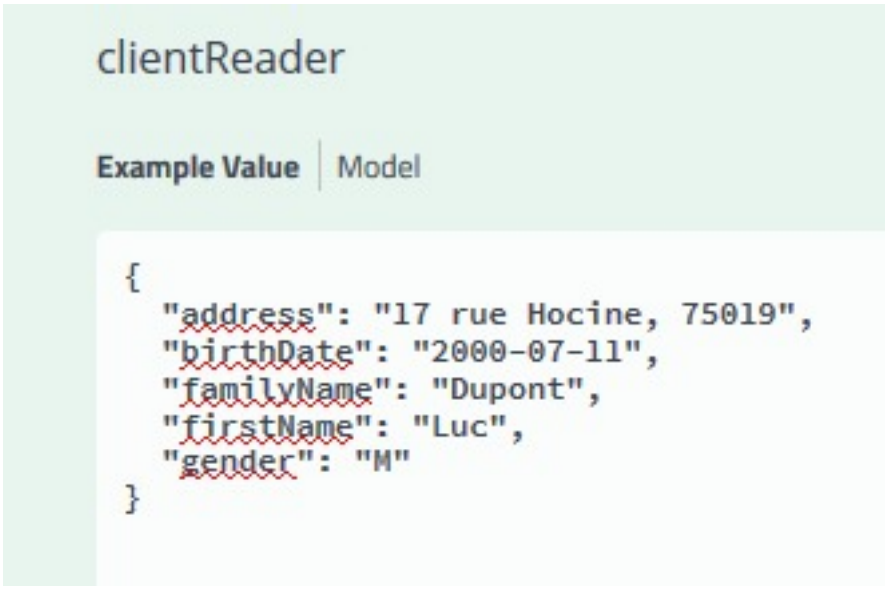

Figure 3: Exemple de création d'un lecteur.

2. **Book API**

- HealthCheck « GET /healthcheck »
- Création d'un livre « POST /books »
- Recherche par ISBN « GET /books/isbn »
- Recherche par attribut (auteur, éditeur, édition ou titre) « GET /books?attribut=valeur ». Si aucun de ces deux attributs n'est spécifié, la requête récupère l'ensemble de tous les livres. Pour des raisons de simplicité, la recherche ne se fait que sur un seul attribut à la fois.
- Mise à jour d'un livre « PUT /books » : Mise à jour partielle d'un livre
- Suppression d'un livre « DELETE /books/isbn »

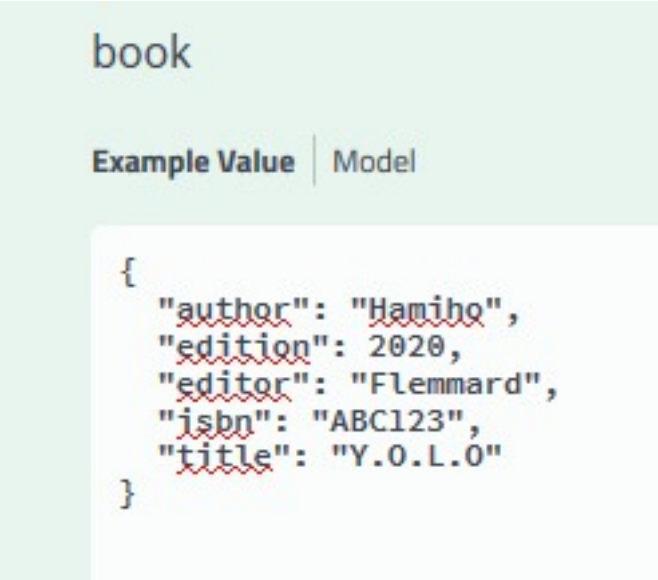

Figure 4: Exemple de création d'un livre.

3. **Loan API**

   - HealthCheck « GET /healthcheck »
   - Création d'un prêt « POST /loans » : Le corps de la requête HTTP contient, en plus des informations sur le prêt, l'identifiant du lecteur et l'ISBN du livre. Des appels à leurs services respectifs sont alors effectués afin de vérifier leur existence et de récupérer leurs informations respectives.
   - Retour d'un livre « PUT /loans/id » : Le retour d'un livre s'effectue en positionnant la date de retour à celle du jour.
   - Récupération par identifiant « GET /loans/id »
   - Recherche par attributs (date d'emprunt ou lecteur) « GET /loans?attribut=valeur » : Si aucun de ces deux attributs n'est spécifié, la requête récupère l'ensemble de tous les prêts. Pour des raisons de simplicité, la recherche ne se fait que sur un seul attribut à la fois.

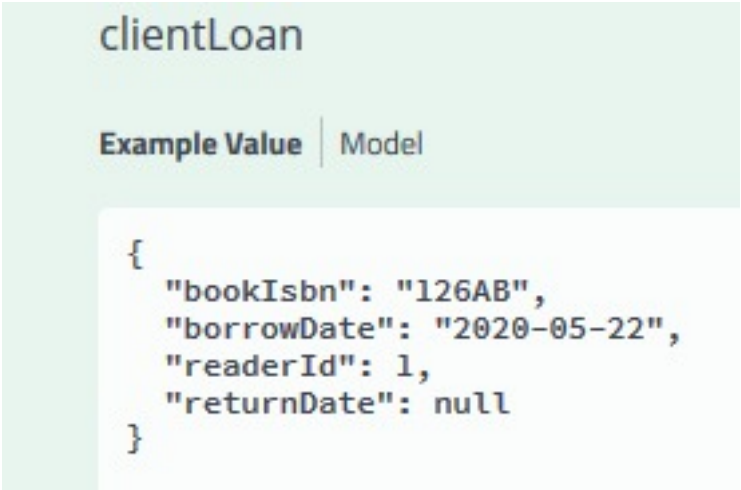

Figure 5: Exemple de création d'un prêt.

2.3 Choix techniques

- **Spring Boot :** Nous utiliserons le Framework Spring, à travers son Starter Spring Boot, afin de réaliser nos microservices.

- **Spring Data JPA :** La persistance des données sera assurée par le Framework JPA (Java Persistence API), qui fournit une couche ORM (Object Relational Mapping) à travers son implémentation par défaut, Hibernate.
- **Spring HATEOAS :** Afin de fournir des APIs de niveau 3 dans le modèle de maturité de Richardson, le Starter HATEOAS sera utilisé.
- **H2 Database :** Une base de données H2 (In Memory) sera utilisée pour la persistance des données.
- **Swagger :** Le Framework Swagger sera utilisé pour fournir une interface graphique qui permettra de tester nos APIs.
- **Junit et Mockito :** Afin d'effectuer nos tests unitaires et de simuler nos injections de dépendances (Mock).
- **Maven :** Les dépendances du projet seront gérées par Maven.
- **Git :** L'outil de versionning Git sera utilisé pour gérer les versions de nos microservices et les sauvegarder dans des dépôts sur Github.
- **Docker :** Afin de lancer nos microservices dans des conteneurs.
- **Minikube :** Afin d'automatiser nos déploiements, gérer la montée en charge et mettre en œuvre nos conteneurs Docker.
- **Postman :** Afin de tester nos APIs REST en envoyant des requêtes HTTP.

2.4 Schéma d'architecture des APIs

Chacune des APIs suit le schéma suivant:

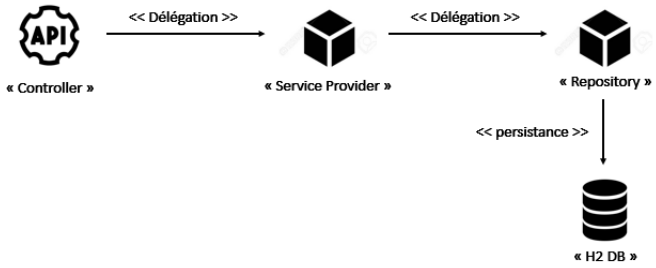

Figure 6: Architecture d'une API.

- Le contrôleur (Controller) constitue le point d'entrée des requêtes HTTP. Il s'agit d'une API REST qui expose les différents endpoints.
- Le Controller ne s'occupe que de l'aiguillage des requêtes. Il délègue alors les actions à réaliser au fournisseur de service (Service Provider), dont la définition est assurée grâce au mécanisme d'injection de dépendances.
- Le Service Provider contient la logique métier (Business Logic) du Microservice. Sous forme d'interface, il définit le contrat de la logique métier, qui sera ensuite implémenté dans une classe concrète que l'on appellera ServiceProviderImpl
- La classe ServiceProviderImpl déléguera alors les accès à la base de données à une autre couche appelée « Repository »
- La couche Repository est la couche d'accès à la base de données (DAO, Data Access Object). Il s'agit d'une interface contenant le même contrat que le Service Provider et dont l'implémentation dépendra du choix du SGBD (H2, dans notre cas).

Cette architecture respecte les principes SOLID de la conception orientée objet. Évolutive et facilement maintenable, c'est l'architecture par défaut des projets Spring Boot.

3 Exécution et compilation du projet

3.1 Liens repository Github

- Repository Github du microservice reader.
- Repository Github du microservice book.
- Repository Github du microservice loan.

3.2 Compilation

Après avoir récupéré les dépôts Git depuis Github, il suffit de lancer la commande :

```
$ mvn clean install
```

afin de récupérer les dépendances du projet, les compiler mais également lancer les tests unitaires. (Ces étapes sont précisées dans le fichier README.md de chaque dépôt Git). Une fois le serveur lancé, nous pouvons tester nos APIs de deux façons différentes :

Utilisation de Swagger : Le Swagger de chaque API se trouve à l'url http://localhost:**port**/api/swagger-ui.html/. Les ports sont les suivants

- Loan API : 8000
- Book API : 8001

- Reader API : 8002

Utilisation de Postman : La collection .json de chaque API se trouve à la racine du projet, et porte le nom XXX $API.postman_collection.json$ où XXX est le nom du projet(Loan, Book ou Reader) Ce fichier est à importer dans Postman.

3.3 Conteneurisation et déploiement avec Docker et Minikube

Afin de dockeriser nos trois applications, après avoir écrit nos "DockerFile", il nous suffit d'exécuter les commandes suivantes, par exemple pour le microservice Loan:

- `$ docker build -t microserivce-loan .`
- `$ docker run --name microservice-loan -p 8000:8000 microserivce-loan`

En ce qui concerne le déploiement avec Minikube nos devons exécuter dans l'ordre les commandes suivantes:

- `$ minikube start`
- `$ minikube start --vm-driver docker`
- `$ eval $(minikube docker-env)`
- `$ docker build -t microservice-loan`
- `$ docker run -p 8000:8000 microservice-loan:latest`
- `$ kubectl create deployment loan-service --image=microservice-loan \ --dry-run=client -o=yaml > deployment-loan.yaml`
- `$ echo --- >>deployment-loan.yaml`
- `$ kubectl create service clusterip loan-service --tcp=8083:8083 --dry-run=client \ -o=yaml >> deployment-loan.yaml`

Puis dans le fichier deployment-loan.yaml, nous ajoutons ces lignes :

```yaml
spec:
  containers:
  - image: microservice-loan:latest
    name: microservice-loan
    imagePullPolicy: Never
    ports:
    - containerPort: 8000
    resources: {}
```

Figure 7: Mises à jour a effectuer dans le fichier .yaml

- `$ kubectl apply -f deployment-loan.yaml`
- `$ kubectl get all`

N.B : Les ports à utiliser pour les différents microservices lors d'une commande *docker run* sont :

- 8000 pour loan.
- 8001 pour book.
- 8002 pour reader.

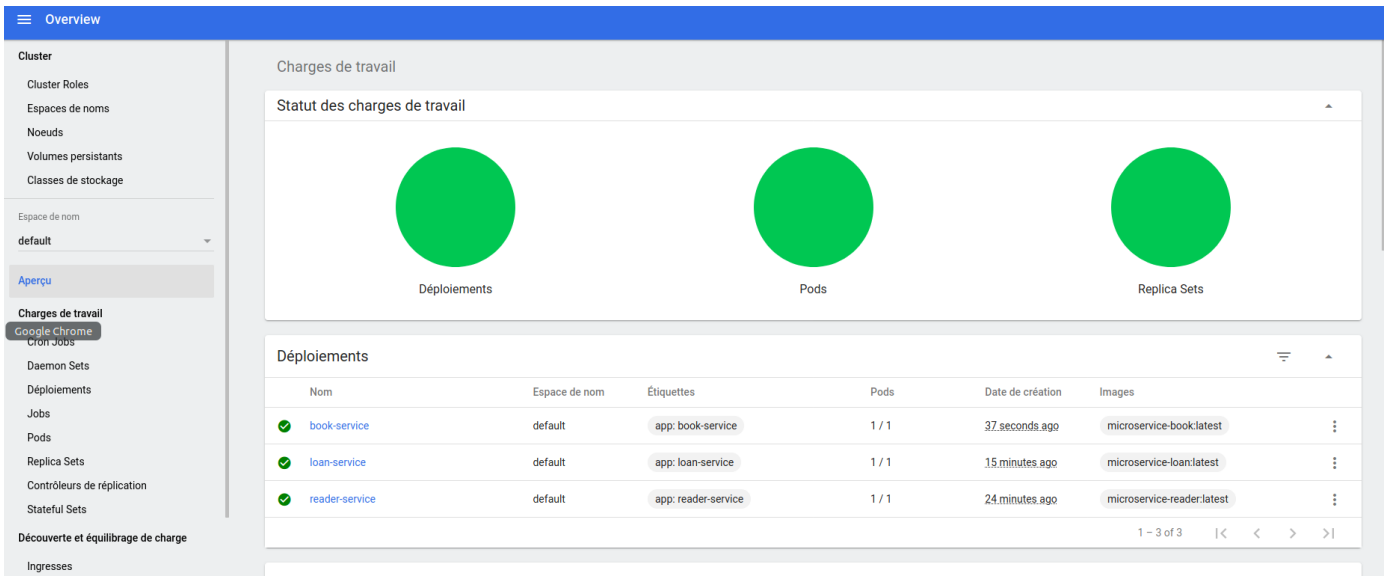

Figure 8: Résultat du déploiement sur le dashboard de Minikube.

4 Bilan du projet

- Travailler sur un projet reflétant ce qui se fait dans le monde professionnel. ✔
- Prise en main de différentes technologies telles que Spring Boot, Docker. ✔
- Difficulté sur l'utilisation de Minikube. ✗