Tutoriel sur le développement full stack d'une application Web avec Angular 7 et Spring Boot 2

Par Georges KEMAYO

Date de publication : 14 novembre 2019

Dernière mise à jour : 6 avril 2020

TOUT PUBLIC

Une application web riche est un logiciel informatique dont la vocation est de fonctionner dans un navigateur web, type Firefox, Chrome, Edge, Safari, etc. Elle est communément appelée *application client léger* et est constituée de deux parties qui intercommuniquent : le *front-end* (partie visible correspondant aux pages web construites et affichées dans votre navigateur) et le *back-end* (partie invisible construite et déployée sur un serveur d'application tel que Tomcat et réalisant les traitements lourds en vue de répondre aux requêtes soumises par le front-end).

Le but de cet article est de vous présenter la construction complète d'une application web générée via le framework **Spring Boot** 2 chargé de la gestion de sa partie back-end. Nous montrerons comment enrichir cette application d'une couche cliente (ou front-end) à l'aide du framework **Angular** dans sa version 7. Nous profiterons pour introduire quelques notions de **Swagger 2** qui est un excellent outil de test et de documentation d'API REST.

Réagissez à cet article en faisant vos remarques ici : **Commentez**

I - Expression du besoin de l'application

Pour mieux exposer les notions techniques et technologiques de cet article, nous allons nous faire guider par un exemple d'application web. Il convient donc de fixer ses besoins et ses objectifs ainsi que les détails métier à implémenter.

I-A - Expression du besoin

L'application web que nous voulons construire est un outil **d'administration de bibliothèque** ou de **gestion de livres**. L'objectif est de proposer un outil qui permet à un bibliothécaire la possibilité de gérer l'insertion dans son système, de nouveaux livres, de nouveaux clients et de réaliser des prêts de livres à ces derniers.

D'après le bibliothécaire, les informations qui l'intéressent sur un livre pour la gestion de sa bibliothèque sont les suivantes : ISBN, titre, date de sortie, nom de son auteur et nombre d'exemplaires qu'il gère. Il souhaite ranger ses livres par catégories (Poésie, Roman, Sciences, etc.) afin de connaitre rapidement dans quel domaine se situe un livre qu'il consulte. Pour les clients, potentiels emprunteurs, qu'il souhaite enregistrer, il lui faut leur nom, leurs prénoms, leur profession, leurs adresses e-mail et postale, et la date à laquelle il les a enregistrés dans son système. Ensuite, lorsque le bibliothécaire réalise une opération d'emprunt, il souhaiterait enregistrer la date de début et de fin prévue de l'emprunt. Pour un client donné, il ne souhaite pas qu'il soit restreint dans l'emprunt de plusieurs livres, mais inversement, comme un livre a plusieurs exemplaires, il souhaite l'emprunter à plusieurs clients autant qu'il y en a d'exemplaires. En revanche, un client ne peut disposer de plusieurs exemplaires du même livre tant qu'il a un emprunt en cours avec ce livre. Par ailleurs, il faudra qu'il puisse être capable d'envoyer un mail à ses emprunteurs. Enfin, le bibliothécaire souhaite une organisation simple de son application pour pouvoir réaliser de façon efficace ses besoins.

I-B - Découpage du besoin en User Stories

Dans un esprit agile et par application de la méthodologie *Scrum*, nous allons constituer, ci-dessous, le *Backlog* de notre application exprimé sous forme d'une liste de *User Stories* (US). Cette approche permet de mettre en place un développement logiciel par petit bout. Chaque bout correspond à une User Story et représente une expression fine et élémentaire du besoin métier concourant à la réalisation de l'objectif global attendu de l'application (cf. **section I-A**).

Pour info, un *Backlog* peut être défini comme un bac à sable contenant une liste ordonnée des fonctionnalités à mettre en place pour atteindre la réalisation d'un produit final qui pour le cas d'espèce est notre l'application web. Une *User Story* dans un backlog représente donc une et une seule fonctionnalité exprimée généralement sous forme de scénario **.....**« En tant que… je veux que… afin de… »

User Story 1 : en tant qu'administrateur de la bibliothèque, je dois pouvoir ajouter un nouveau livre, afin d'augmenter l'effectif des livres du système.

User Story 2 : en tant qu'administrateur de la bibliothèque, je dois pouvoir rechercher un livre par son ISBN ou par une partie ou totalité de son titre, afin de visualiser l'ensemble de ses informations.

User Story 3 : en tant qu'administrateur de la bibliothèque, suite à une recherche de livres, je dois pouvoir avoir la possibilité de modifier ou de supprimer chacun des livres sur la liste qui s'affiche.

User Story 4 : en tant qu'administrateur de la bibliothèque, je dois pouvoir ajouter un nouveau client, afin d'augmenter l'effectif des clients du système.

User Story 5 : en tant qu'administrateur de la bibliothèque, je dois pouvoir rechercher un client par son e-mail ou par une partie ou totalité de son nom de famille, afin de visualiser l'ensemble de ses informations.

User Story 6 : en tant qu'administrateur de la bibliothèque, suite à une recherche de clients, je dois pouvoir avoir la possibilité de modifier ou de supprimer chacun des clients sur la liste qui s'affiche.

User Story 7 : en tant qu'administrateur de la bibliothèque, je dois pouvoir ajouter un nouveau prêt, afin de réaliser la connexion entre un livre sorti de la bibliothèque et son emprunteur (client enregistré).

User Story 8 : en tant qu'administrateur de la bibliothèque, je dois pouvoir rechercher la liste des prêts d'un client via son e-mail et la liste des prêts réalisés jusqu'à une certaine date, afin de visualiser l'historique.

User Story 9 : en tant qu'administrateur de la bibliothèque, suite à une recherche de prêts, je dois pouvoir avoir la possibilité de modifier ou de clôturer chacun des prêts sur la liste qui s'affiche.

User Story 10 : en tant qu'administrateur de la bibliothèque, suite à une recherche de prêts, je dois pouvoir avoir la possibilité d'envoyer un mail à un emprunteur, afin de lui passer une information utile.

User Story 11 : en tant qu'administrateur de la bibliothèque, je dois pouvoir avoir sur mon application, un menu général qui me permet d'accéder à la page de gestion des livres, celle des clients et celle des prêts ; afin d'être plus efficace dans l'utilisation de l'application.

Cette liste de User stories n'est pas exhaustive. Nous ferons abstraction dans le cadre de cet article de l'aspect sécurisation de l'application décliné par l'authentification de l'administrateur pour avoir accès aux différentes possibilités citées dans les US ci-dessus.

II - Conception et choix architectural

II-A - Modèle de données UML

Lorsque nous analysons attentivement l'expression des besoins exprimés en **section I-A**, il en ressort quatre entités pertinentes permettant de modéliser le système d'informations de la bibliothèque : ***Livre***, ***Catégorie***, ***Client*** et ***Prêt***.

Une lecture encore minutieuse permet ensuite de déceler une *association unidirectionnelle* entre le Livre et la Catégorie et une *Classe association* entre le Livre et le Client qui peuvent se décliner en ces deux points :

- **association unidirectionnelle :** un Livre appartient à une et une seule Catégorie. Autrement dit, une Catégorie peut être vue comme un identifiant ensembliste de plusieurs Livres traitant le même thème. L'inverse n'est pas vrai, d'où le caractère unidirectionnel de l'association ;
- **association :** un Client peut effectuer un prêt de plusieurs Livres ; et comme un Livre a plusieurs exemplaires, il peut être emprunté par plusieurs Clients. Chaque prêt ayant une date de début et de fin à prendre en compte, on se retrouve dans une association porteuse de propriétés.

Les éléments exposés ci-dessus nous permettent, en résumé, de concevoir le diagramme de classes UML représenté par la figure ci-dessous. Pour la suite de cet article et pour donner une dimension internationale à notre application, tous les termes de la bibliothèque sont utilisés dans leur expression anglaise.

Diagramme de classes UML pour la gestion de la bibliothèque

- Client = **Customer**
- Livre = **Book**
- Catégorie = **Category**
- Prêt = **Loan**
- Bibliothèque = **Library**

II-B - Choix et proposition d'architecture

Dans cet article, nous allons mettre en œuvre notre application en faisant les choix suivants :

- le front-end correspondant au côté client, sera mis en place à l'aide des langages descriptifs HTML/CSS. Nous gérerons la dynamisation des pages web construites, ainsi que la communication entre le front-end et le back-end à l'aide du framework Angular dans sa version 7 ;
- le back-end correspondant au côté serveur, sera mis en place à l'aide du framework Spring Boot 2 qui nous permettra de générer un projet Java préconstruit de type ***.war***. Spring Boot est un framework qui offre entre autres, la possibilité de construire une application Java intégrant par défaut un écosystème des frameworks Spring. De plus, il donne la possibilité à l'application une fois construite d'être déployée aisément et de fonctionner de façon autonome (mode stand-alone). Un de ses atouts majeurs est qu'il s'occupe de l'inclusion et de la gestion des versions de multiples dépendances aux frameworks dont peut nécessiter l'application lors de son développement.

Par ailleurs dans cet article, nous appliquerons le ***package by feature architecture*** qui est une architecture basée sur une organisation des packages d'un projet Java, par domaine fonctionnel. Cette approche est en quelque sorte une forme de découpage et regroupement des traitements par domaine fonctionnel, mais tous au sein du même projet. Il précède ainsi une architecture de type *Microservice*.

Ce qui nous motive dans ce choix, est le fait qu'en analysant le cahier des charges de l'application à construire, il en ressort quatre domaines fonctionnels qui peuvent être traités séparément : le ***domaine fonctionnel des clients***, le ***domaine fonctionnel des livres***, le ***domaine fonctionnel des catégories de livres*** et le ***domaine fonctionnel des prêts***.

Concrètement qu'est-ce qu'une architecture *package by feature* ?

L'architecture *Package by feature* est un modèle d'organisation des Classes d'un projet Java par domaine fonctionnel. C'est-à-dire que contrairement à l'architecture *n-tiers* très répandue, qui organise les packages d'un projet par couches (couche présentation/service, couche métier et couche Dao), elle regroupe au sein d'un même package toutes les Classes Java courantes à répondre aux traitements d'un domaine fonctionnel de l'application à construire (Classe Controller, Classe Service, Classe Dao, etc.). Pour mieux vous documenter sur cette architecture et comprendre ses avantages et ses inconvénients, nous vous proposons de lire cet **article**.

Frameworks et IDE utilisés pour cette application

- **HTML/CSS et Angular 7**, pour les traitements front-end.
- **Java 8**, pour les développements applicatifs back-end.
- **Spring Boot 2.1.2**, pour la construction de notre projet applicatif *.war*.
- **Spring ioc**, pour l'injection de dépendances.
- **Spring webmvc**, pour le traitement des requêtes HTTP Restful venant du front-end.
- **Springfox 2.9.2**, pour la documentation **Swagger 2** des API REST développées ;
- **Spring Mail**, pour la gestion des mails.
- **Spring Data JPA / Hibernate**, pour la persistance des données.
- **H2**, la base de données de notre application. Nous faisons le choix d'utiliser une base embarquée pour cet article, afin d'avoir une application complètement portable et sans configuration à faire une fois développée. Ceci pour le grand bonheur des lecteurs qui voudraient tester l'application. À noter quand même que dans la réalité, une base embarquée est généralement utilisée uniquement pour les tests d'intégration, mais sa configuration reste identique à toutes autres bases de données telles que MySql, MariaDB, Oracle, etc.
- **Eclipse 4.10 (SimRel 2018-12)**, comme IDE pour le développement de notre application côté serveur.
- **Visual Studio Code 1.35.1**, comme IDE pour le développement de notre application côté client.
- **Tomcat 8** pour le déploiement de l'application.

La figure ci-dessous résume de façon graphique l'architecture de notre application.

Architecture de l'application de gestion de la bibliothèque

Dans la suite, nous allons entrer dans le vif du sujet. La page suivante présentera le processus de création et de configuration du projet via Spring Boot.

III - Création du projet Spring Boot

Spring Boot offre la possibilité de créer deux types d'application :

- Java archive, correspondant à une application dont le paquetage est d'extension *.jar* ;
- Web archive, correspondant à une application dont le paquetage est d'extension *.war.*

Dans cet article et pour notre exemple, nous allons créer une application web et donc de type *.war*. Spring Boot nous donne la possibilité de créer un projet Java déjà préconstruit et disposant des configurations Spring de base permettant à l'application d'être immédiatement exploitable. Il fonctionne à la manière d'un Magasin à outils qui nous donne la possibilité de faire notre marché et de choisir les différents outils nécessaires à la réalisation de notre application. Les avantages de Spring Boot sont nombreux :

- offre une interface simple de création d'un projet organisé et préconfiguré, prêt à exploitation ;
- permet l'ajout des dépendances et la gestion de leurs versions via des *Starters* ;
- permet l'autoconfiguration d'une application moyennant des annotations simples ;
- donne la possibilité de **déployer/exécuter** l'application sans aucune autre ressource externe (mode standalone pour le cas des applications .jar) ;
- simplifie la configuration du projet pour permettre de se concentrer sur le développement des règles métiers ;
- ajouter de nouvelles dépendances reste simple pendant la phase de développement ;
- etc.

Nous vous présentons ci-dessous les deux étapes nécessaires à la création d'un projet Spring Boot que nous guidons par les besoins de notre application de gestion de bibliothèque (appelée **Library**, en anglais).

Étape 1 : aller sur le site web de Spring Boot **https://start.spring.io/** pour la création et la préconfiguration du projet **Library**.

Création d'un projet Spring Boot

Dans cette capture d'écran, nous avons préconfiguré quatre éléments :

- nous créons un projet de type **Maven** pour la gestion des dépendances, du cycle de vie et l'organisation de *packages*. L'autre option étant *Gradle* ;
- nous créons un projet dont le langage applicatif est **Java.** Les autres options étant *Kotlin* et *Groovy* ;
- nous choisissons la version **2.1.7** de Spring Boot qui est celle effectivement éprouvée lors de l'écriture de cet article ;
- ensuite, nous paramétrons le *goup id* de l'application (**com.gkemayo**), le nom du projet (**library**) et le type d'artefact à générer par Maven (**war**) ;
- et finalement, nous choisissons la version du langage Java utilisée pour la compilation des *sources*, **Java 8**.

Étape 2 : nous allons maintenant choisir les différentes dépendances que Spring Boot inclura dans notre projet. Pour rappel, dans la **section II-B** (Choix et proposition d'architecture), nous avons listé les besoins techniques de l'application. Ici, nous choisissons déjà ce que Spring Boot nous propose, le reste sera ajouté manuellement plus tard une fois le projet généré.

Création d'un projet Spring Boot - le choix des dépendances

La figure ci-dessus présente une capture d'écran montrant entre autres le choix des dépendances *H2* et *Spring Data JPA*. Le petit onglet en haut à droite nous indique que nous avons inclus quatre dépendances dans notre projet. Les deux autres étant *Spring mail* et *Spring web*.

En cliquant enfin sur le bouton **Generate the project**, nous téléchargeons le projet ainsi créé (sous forme de zip) sur notre machine pour utilisation. En le dézippant et en l'ouvrant à l'aide d'un IDE comme Eclipse, il est prêt pour contenir du code métier. Mais avant d'y arriver, nous pouvons visualiser la structure de ce projet en cliquant sur le bouton **Explore the project** :

Structure d'un projet Spring Boot généré

Nous pouvons aisément remarquer sur la partie gauche de cette image, le rendu structurel d'un projet maven avec les packages *src/main/java*, *src/main/resources* et *src/main/test*.

IV - Exploration du projet Spring Boot

IV-A - Le pom.xml

Une fois notre projet maven créé via Spring Boot, le pom.xml généré est le suivant :

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
3.     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
maven-4.0.0.xsd">
4.     <modelVersion>4.0.0</modelVersion>
5.     <parent>
6.         <groupId>org.springframework.boot</groupId>
7.         <artifactId>spring-boot-starter-parent</artifactId>
8.         <version>2.1.7.RELEASE</version>
9.         <relativePath/> <!-- lookup parent from repository -->
10.     </parent>
11.     <groupId>com.gkemayo</groupId>
12.     <artifactId>library</artifactId>
13.     <version>0.0.1-SNAPSHOT</version>
14.     <packaging>war</packaging>
15.     <name>library</name>
16.     <description>Project for managing books in a library</description>
17.
18.     <properties>
19.         <java.version>1.8</java.version>
20.     </properties>
21.
```

```
    <dependencies>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-mail</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
                <groupId>com.h2database</groupId>
                <artifactId>h2</artifactId>
                <scope>runtime</scope>
            </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

Pour expliquer ce pom.xml, nous le découpons en quatre blocs :

Bloc 1 :

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Ce bloc décrit le *projet parent* du projet *library*. En effet, tout projet Spring Boot créé est toujours en quelque sorte un module d'un projet parent nommé *spring-boot-starter-parent* dont le group id est *org.springframework.boot*. Pour cet exemple, ce parent est à la version 2.1.7 (celle que nous avons choisie à la **section III**). C'est au sein de ce projet parent que les différentes dépendances que nous invoquerons dans le bloc 3 ci-dessous sont gérées. Le pom.xml du parent est consultable Source **ici**.

Bloc 2 :

```
<groupId>com.gkemayo</groupId>
<artifactId>library</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>library</name>
<description>Project for managing books in a library</description>
```

```
7.
8. <properties>
9.     <java.version>1.8</java.version>
10. </properties>
```

Ce bloc correspond essentiellement aux différents paramètres que nous avons renseignés lors de la création du projet (cf. figure 1 de la **section II**).

Bloc 3 :

```
1. <dependencies>
2.     <dependency>
3.         <groupId>org.springframework.boot</groupId>
4.         <artifactId>spring-boot-starter-data-jpa</artifactId>
5.     </dependency>
6.     <dependency>
7.         <groupId>org.springframework.boot</groupId>
8.         <artifactId>spring-boot-starter-mail</artifactId>
9.     </dependency>
10.     <dependency>
11.         <groupId>org.springframework.boot</groupId>
12.         <artifactId>spring-boot-starter-web</artifactId>
13.     </dependency>
14.     <dependency>
15.           <groupId>com.h2database</groupId>
16.           <artifactId>h2</artifactId>
17.           <scope>runtime</scope>
18.     </dependency>
19.     <dependency>
20.         <groupId>org.springframework.boot</groupId>
21.         <artifactId>spring-boot-starter-tomcat</artifactId>
22.         <scope>provided</scope>
23.     </dependency>
24.     <dependency>
25.         <groupId>org.springframework.boot</groupId>
26.         <artifactId>spring-boot-starter-test</artifactId>
27.         <scope>test</scope>
28.     </dependency>
29. </dependencies>
```

Dans ce bloc, nous demandons à Spring Boot d'injecter les dépendances concernant **Spring Mail** (*spring-boot-starter-mail*), **Spring Data** et **Hibernate** (*spring-boot-starter-data-jpa*), **Spring Web** (*spring-boot-starter-web*), **la base de données embarquée H2** (*h2*) et un Tomcat **embarqué** (*spring-boot-starter-tomcat*). Par défaut, Spring Boot injecte le starter *spring-boot-starter-test* qui permet d'embarquer les dépendances telles que **JUnit** permettant de réaliser des tests unitaires dans notre application, mais cela n'est pas le point focal de cet article.

Enfin, nous pouvons remarquer (pour ceux qui connaissent nommément la dépendance maven de certains frameworks) que la plupart du temps Spring Boot n'inclut pas directement la dépendance du framework souhaité, mais utilise plutôt la notion de dépendance ***starter***. Cette notion est introduite pour déléguer la possibilité au projet parent (*spring-boot-starter-parent*) de gérer lui-même les dépendances maven effectives ainsi que les versions à inclure dans notre projet. Ceci permet au développeur de s'affranchir de la gestion propre des conflits entre dépendances. Une définition complète des starters Spring Boot est disponible **ici**.

Bloc 4 :

```
1. <build>
2.     <plugins>
3.         <plugin>
4.             <groupId>org.springframework.boot</groupId>
5.             <artifactId>spring-boot-maven-plugin</artifactId>
6.         </plugin>
7.     </plugins>
8. </build>
```

Cette partie du pom.xml signifie simplement que l'on délègue la construction automatique des artefacts du projet *library* à Spring Boot qui utilisera maven pour le faire.

IV-B - application.properties

Si vous avez déjà eu à faire de la configuration Spring, il va sans dire que vous connaissez la difficulté à gérer et maintenir les différents fichiers ressources *.properties* contenant les données dont les beans injectés dans votre application ont besoin pour fonctionner correctement. Ceci est encore plus difficile lorsque vous gérer la configuration Spring effectuée dans des fichiers xml.

Pour pallier ce dispersement dans la configuration des fichiers ressources, Spring Boot propose l'unique et central fichier *application.properties* dans lequel il faut mettre toutes les données nécessaires aux beans injectés dans l'application. *application.properties* comme tout fichier ressource Spring, contient une liste de données ***clef=valeur***.

Enfin, Spring Boot a déjà recensé pour nous, une liste de ressources standards qui peuvent être ajoutées à l'application.properties de notre application en fonction du besoin. Nous pouvons trouver la liste **ici**.

IV-C - LibraryApplication.java et ServletInitializer.java

Après avoir généré le projet web Spring Boot *library*, en le dezippant et en l'ouvrant, on constate la présence de deux classes java dans le *package com.gkemayo.library* : **LibraryApplication.java** et **ServletInitializer.java**.

LibraryApplication.java :

```
1. package com.gkemayo.library;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5.
6. @SpringBootApplication
7. public class LibraryApplication {
8.
9.     public static void main(String[] args) {
10.         SpringApplication.run(LibraryApplication.class, args);
11.     }
12.
13. }
```

Cette classe est le point d'entrée de notre application. C'est-à-dire, la classe utilisée par Spring Boot pour démarrer (*booter*) notre application. En effet, Spring Boot marque toute classe comme point de démarrage d'une application par l'annotation **@SpringBootApplication**. Cette annotation propre à Spring Boot est en réalité l'agrégation de trois annotations Spring qui sont :

- **@configuration :** cette annotation indique à Spring de considérer la classe sur laquelle elle est apposée comme étant un *bean* dans lequel d'autres *beans* peuvent être déclarés ;
- **@CompoenentScan(basePackages = "com.gkemayo.library"**) : cette annotation indique à Spring de scanner le package et les sous-packages passés en paramètre pour rechercher des classes qu'elle pourrait injecter comme *beans* dans l'application. Ces classes peuvent être marquées par des annotations telles *@Component*, *@RestController*, *@Service*, *@Repository*, *@Configuration*, etc. Lorsque l'annotation @ComponentScan ne prend pas de paramètre, alors c'est le package de la classe sur laquelle elle est apposée qui est considéré ;
- **@EnableAutoConfiguration :** cette annotation permet de détecter et d'injecter comme *bean*, certaines classes contenues dans les dépendances du classpath de votre application.

ServletInitializer.java :

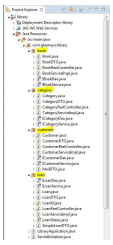
```
1. package com.gkemayo.library;
```

```
2.
3. import org.springframework.boot.builder.SpringApplicationBuilder;
4. import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
5.
6. public class ServletInitializer extends SpringBootServletInitializer {
7.
8.     @Override
9.     protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
10.         return application.sources(LibraryApplication.class);
11.     }
12.
13. }
```

Toute application web doit se voir configurer une classe Servlet dont le but est d'intercepter toute requête HTTP -- venant d'un client web en direction d'un serveur d'application -- dans l'optique de la contrôler et de gérer l'intercommunication. Spring Boot crée automatiquement pour tout projet web la classe *ServletInitializer* héritant de *SpringBootServletInitializer* correspondant à la servlet de notre application et utilisant toutes les ressources de la classe de démarrage, dans le cas d'espèce *LibraryApplication*.

V - Objectifs du back-end

Dans la section **II-B** de notre article, nous avions identifié quatre domaines fonctionnels pour notre application de gestion de la bibliothèque : *Book*, *Customer*, *Loan* et *Category*. Nous avons aussi fait le choix d'utiliser une architecture d'organisation du projet de type *package by feauture*. Pour rappel, cette architecture se distingue de l'architecture *n-tiers* (très répandue) par le fait qu'elle prescrit la mise de toutes classes Java d'un domaine fonctionnel au même niveau dans le package dédié, contrastant ainsi avec une organisation en couches. Au terme de notre développement, notre projet *library* devra ressembler à la figure ci-dessous et cela représente notre objectif.



Objectif du projet
Spring Boot Library

Dans chacun de ces domaines coloriés en jaune sur la figure, nous devons créer quatre classes pertinentes :

- une classe *entité* (exemple, de la classe *Book.java*) correspondant à l'ORM (Object Relationnal Mapping) hibernate/Jpa ;
- une classe *Dao* (exemple, *IBookDao.java*) correspondant à la classe d'accès à la base de données et de traitement des requêtes sur les entités;
- une classe *Service* (exemple, *BookServiceImpl.java*) correspondant à la classe de traitement des règles métier ;
- une classe *Rest Controller* (exemple, *BookRestController.java*) correspondant à la classe d'exposition des services REST (ou web services) de notre application en direction des composants consommateurs tels que le front-end.

Si nous passons en revue, la liste des Users Stories de notre application édictée en section **I-B**, en dehors du besoin d'envoi de mail et des IHM, le reste se réduit à de simples opérations communément appelées **CRUD** (*Create*, *Read*, *Update*, *Delete*) pour les domaines *Book*, *Customer*, *Category* et *Loan*. Pour ce faire, nous ne présenterons que les concepts pertinents et non redondants de chacun de ces domaines. Une présentation vidéo complète de notre application sera faite à la fin de cet article. Nous vous donnerons aussi le lien d'accès aux *codes sources*.

VI - Configuration des ressources de l'application

Notre application *Library* a besoin de trois ressources pour fonctionner correctement et comme attendu :

- la base de données H2 et sa *DataSource*, pour permettre à l'application de s'y connecter ;
- une ressource Spring mail, afin de permettre à l'application de pouvoir envoyer des mails (cf. User Story 10) ;
- un fichier de données (*categories.sql*), correspondant aux différentes catégories de livres, à charger au démarrage de l'application. Il s'agit de données de référence qui ne sont pas susceptibles de changer. Dans notre application nous n'avons listé qu'un exemple non exhaustif.

La figure ci-dessous présente en jaune ces différentes ressources et le contenu du fichier de données *categories.sql* qu'il faut absolument configurer dans le package *java/main/resources* de l'application. Spring Boot sait détecter les ressources à placer dans ce package.

Ressources de l'application library

La configuration de ces ressources est effectuée dans le fichier *application.properties* qui se présente ainsi :

```
1. ############## DataSource Config #################
2. spring.datasource.name=library-db
3. spring.datasource.username=sa
4. spring.datasource.password=
5. spring.datasource.url = jdbc:h2:file:./src/main/resources/database/library-db
6. spring.datasource.driver-class-name=org.h2.Driver
7. spring.datasource.sql-script-encoding= UTF-8
8. spring.datasource.data=classpath:data/categories.sql
9.
10. ############## Hibernate properties #################
11. spring.jpa.show-sql=true
12. spring.jpa.hibernate.ddl-auto=create-drop
13. spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
14.
15. ############## Enable H2 Database browser console #################
16. #http://localhost:port/library/h2-console/
17. spring.h2.console.enabled=true
18.
19.
20. ############## Email Config #################
21. spring.mail.default-encoding=UTF-8
22. spring.mail.protocol=smtp
23. spring.mail.host=smtp.gmail.com
24. spring.mail.username=noreply.library.test@gmail.com
25. spring.mail.password=password1Test
26. spring.mail.port= 587
27. spring.mail.properties.mail.smtp.auth=true
28. spring.mail.properties.mail.smtp.starttls.enable=true
29. spring.mail.test-connection=false
30. #https://www.google.com/settings/security/lesssecureapps
```

Bien que vous puissiez ajouter vos propriétés personnelles dans l'*application.properties*, pour cette application, nous n'en avons pas eu besoin. Celles que nous avons utilisées sont les **propriétés standards** proposées par Spring Boot. Vous pouvez remarquer que les propriétés :

- ***spring.datasource.****, permettent de configurer l'accès à la base de données et sa localisation. Pour le cas d'espèce, nous avons configuré la base embarquée H2 qui persistera les données non en mémoire, mais dans un fichier */src/main/resources/database/library-db*. La propriété *spring.datasource.data* permet à Spring Boot d'exécuter un script sql au démarrage de l'application. On pourra donc charger nos différentes catégories de livres. En affectant la valeur *create-drop* à la propriété *spring.jpa.hibernate.ddl-auto*, nous

demandons à hibernate, à chaque fois que l'application démarre, de supprimer et de recréer le schéma de la base de données ;
- ***spring.jpa.****, permettent de configurer les paramètres Hibernate/JPA. Dans le cas d'espèce : toute requête sql exécutée sera tracée dans la console ; hibernate supprimera et recréera la base de données au démarrage de l'application. Enfin, le dialecte qu'il utilisera pour communiquer avec la base est celui d'H2 naturellement ;
- ***spring.h2.console.enable***, active la possibilité de visualiser dans un navigateur les tables de notre base de données H2 (cf. lien en commentaire) ;
- ***spring.mail.****, permettent de configurer les ressources d'envoi de mails. Pour le cas d'espèce, nous avons choisi *gmail.com* comme le fournisseur du compte d'envoi de mails (on aurait pu choisir Y*ahoo*, ou H*otmail*, etc.). Nous détaillerons un peu plus cette partie dans la suite de cet article.

VII - Les entités hibernate/JPA

Dans la section **II-A**, nous vous avons présenté le modèle de données UML de notre application. De ce modèle de données, nous déduisons le modèle relationnel ci-dessous. Cela résulte de l'application des **règles de passage UML** :

- *Category*(code, label) ;
- *Book*(id, isbn, title, creation_date, total_examplaries, author) ;
- *Customer*(id, first_name, last_name, job, address, email, creation_date) ;
- *Loan*(book_id, customer_id, begin_date, end_date, status).

La propriété soulignée dans chaque *relation* ci-dessus correspond à sa clef primaire. La relation *Loan* se voit ainsi migrer les deux clefs primaires des relations *Book* et *Customer* correspondant par conséquent à sa clef primaire composée.

Cette modélisation permet ainsi de créer les classes entités hibernate/JPA suivantes :

- Classe entité *Category* :

```
1. package com.gkemayo.library.category;
2.
3. import javax.persistence.Column;
4. import javax.persistence.Entity;
5. import javax.persistence.Id;
6. import javax.persistence.Table;
7.
8. @Entity
9. @Table(name = "CATEGORY")
10. public class Category {
11.
12.     public Category() {
13.     }
14.
15.     public Category(String code, String label) {
16.         super();
17.         this.code = code;
18.         this.label = label;
19.     }
20.
21.     private String code;
22.
23.     private String label;
24.
25.     @Id
26.     @Column(name = "CODE")
27.     public String getCode() {
28.         return code;
29.     }
30.
31.     public void setCode(String code) {
```

```
32.         this.code = code;
33.     }
34.
35.     @Column(name = "LABEL", nullable = false)
36.     public String getLabel() {
37.         return label;
38.     }
39.
40.     public void setLabel(String label) {
41.         this.label = label;
42.     }
43.
44.     // + les méthodes hashCode() et equals() que vous retrouverez dans les sources de cet
 article
45. }
```

- Classe entité *Book* :

```
1. package com.gkemayo.library.book;
2.
3. import java.time.LocalDate;
4. import java.util.HashSet;
5. import java.util.Set;
6.
7. import javax.persistence.CascadeType;
8. import javax.persistence.Column;
9. import javax.persistence.Entity;
10. import javax.persistence.FetchType;
11. import javax.persistence.GeneratedValue;
12. import javax.persistence.GenerationType;
13. import javax.persistence.Id;
14. import javax.persistence.JoinColumn;
15. import javax.persistence.ManyToOne;
16. import javax.persistence.OneToMany;
17. import javax.persistence.Table;
18.
19. import com.gkemayo.library.category.Category;
20. import com.gkemayo.library.loan.Loan;
21.
22. @Entity
23. @Table(name = "BOOK")
24. public class Book {
25.
26.     private Integer id;
27.
28.     private String title;
29.
30.     private String isbn;
31.
32.     private LocalDate releaseDate;
33.
34.     private LocalDate registerDate;
35.
36.     private Integer totalExamplaries;
37.
38.     private String author;
39.
40.     private Category category;
41.
42.     Set<Loan> loans = new HashSet<Loan>();
43.
44.     @Id
45.     @GeneratedValue(strategy = GenerationType.AUTO)
46.     @Column(name = "BOOK_ID")
47.     public Integer getId() {
48.         return id;
49.     }
50.
51.     public void setId(Integer id) {
52.         this.id = id;
53.     }
```

```
54.
55.     @Column(name = "TITLE", nullable = false)
56.     public String getTitle() {
57.         return title;
58.     }
59.
60.     public void setTitle(String title) {
61.         this.title = title;
62.     }
63.
64.     @Column(name = "ISBN", nullable = false, unique = true)
65.     public String getIsbn() {
66.         return isbn;
67.     }
68.
69.     public void setIsbn(String isbn) {
70.         this.isbn = isbn;
71.     }
72.
73.     @Column(name = "RELEASE_DATE", nullable = false)
74.     public LocalDate getReleaseDate() {
75.         return releaseDate;
76.     }
77.
78.     public void setReleaseDate(LocalDate releaseDate) {
79.         this.releaseDate = releaseDate;
80.     }
81.
82.     @Column(name = "REGISTER_DATE", nullable = false)
83.     public LocalDate getRegisterDate() {
84.         return registerDate;
85.     }
86.
87.     public void setRegisterDate(LocalDate registerDate) {
88.         this.registerDate = registerDate;
89.     }
90.
91.     @Column(name = "TOTAL_EXAMPLARIES")
92.     public Integer getTotalExamplaries() {
93.         return totalExamplaries;
94.     }
95.
96.     public void setTotalExamplaries(Integer totalExamplaries) {
97.         this.totalExamplaries = totalExamplaries;
98.     }
99.
100.     @Column(name = "AUTHOR")
101.     public String getAuthor() {
102.         return author;
103.     }
104.
105.     public void setAuthor(String author) {
106.         this.author = author;
107.     }
108.
109.     @ManyToOne(optional = false)
110.     @JoinColumn(name = "CAT_CODE", referencedColumnName = "CODE")
111.     public Category getCategory() {
112.         return category;
113.     }
114.
115.     public void setCategory(Category category) {
116.         this.category = category;
117.     }
118.
119.     @OneToMany(fetch = FetchType.LAZY, mappedBy = "pk.book", cascade = CascadeType.ALL)
120.     public Set<Loan> getLoans() {
121.         return loans;
122.     }
123.
124.     public void setLoans(Set<Loan> loans) {
125.         this.loans = loans;
```

```
126.     }
127.
128.     // + les méthodes hashCode() et equals() que vous retrouverez dans les sources de cet
 article
129. }
```

- Classe entité *Customer* :

```
1. package com.gkemayo.library.customer;
2.
3. import java.time.LocalDate;
4. import java.util.HashSet;
5. import java.util.Set;
6.
7. import javax.persistence.CascadeType;
8. import javax.persistence.Column;
9. import javax.persistence.Entity;
10. import javax.persistence.FetchType;
11. import javax.persistence.GeneratedValue;
12. import javax.persistence.GenerationType;
13. import javax.persistence.Id;
14. import javax.persistence.OneToMany;
15. import javax.persistence.Table;
16.
17. import com.gkemayo.library.loan.Loan;
18.
19. @Entity
20. @Table(name = "CUSTOMER")
21. public class Customer {
22.
23.     private Integer id;
24.
25.     private String firstName;
26.
27.     private String lastName;
28.
29.     private String job;
30.
31.     private String address;
32.
33.     private String email;
34.
35.     private LocalDate creationDate;
36.
37.     Set<Loan> loans = new HashSet<Loan>();
38.
39.     @Id
40.     @GeneratedValue(strategy = GenerationType.AUTO)
41.     @Column(name = "CUSTOMER_ID")
42.     public Integer getId() {
43.         return id;
44.     }
45.
46.     public void setId(Integer id) {
47.         this.id = id;
48.     }
49.
50.     @Column(name = "FIRST_NAME", nullable = false)
51.     public String getFirstName() {
52.         return firstName;
53.     }
54.
55.     public void setFirstName(String firstName) {
56.         this.firstName = firstName;
57.     }
58.
59.     @Column(name = "LAST_NAME", nullable = false)
60.     public String getLastName() {
61.         return lastName;
62.     }
63.
```

```
64.     public void setLastName(String lastName) {
65.         this.lastName = lastName;
66.     }
67.
68.     @Column(name = "JOB")
69.     public String getJob() {
70.         return job;
71.     }
72.
73.     public void setJob(String job) {
74.         this.job = job;
75.     }
76.
77.     @Column(name = "ADDRESS")
78.     public String getAddress() {
79.         return address;
80.     }
81.
82.     public void setAddress(String address) {
83.         this.address = address;
84.     }
85.
86.     @Column(name = "EMAIL", nullable = false, unique = true)
87.     public String getEmail() {
88.         return email;
89.     }
90.
91.     public void setEmail(String email) {
92.         this.email = email;
93.     }
94.
95.     @Column(name = "CREATION_DATE", nullable = false)
96.     public LocalDate getCreationDate() {
97.         return creationDate;
98.     }
99.
100.     public void setCreationDate(LocalDate creationDate) {
101.         this.creationDate = creationDate;
102.     }
103.
104.     @OneToMany(fetch = FetchType.LAZY, mappedBy = "pk.customer", cascade = CascadeType.ALL)
105.     public Set<Loan> getLoans() {
106.         return loans;
107.     }
108.
109.     public void setLoans(Set<Loan> loans) {
110.         this.loans = loans;
111.     }
112.
113.     // + les méthodes hashCode() et equals() que vous retrouverez dans les sources de cet
 article
114. }
```

- Classe entité *Loan* :

```
1. package com.gkemayo.library.loan;
2.
3. import java.io.Serializable;
4. import java.time.LocalDateTime;
5.
6. import javax.persistence.CascadeType;
7. import javax.persistence.Column;
8. import javax.persistence.Embeddable;
9. import javax.persistence.FetchType;
10. import javax.persistence.JoinColumn;
11. import javax.persistence.ManyToOne;
12.
13. import com.gkemayo.library.book.Book;
14. import com.gkemayo.library.customer.Customer;
15.
16. @Embeddable
```

```
17. public class LoanId implements Serializable {
18. 
19.     /**
20.      *
21.      */
22.     private static final long serialVersionUID = 3912193101593832821L;
23. 
24.     private Book book;
25. 
26.     private Customer customer;
27. 
28.     private LocalDateTime creationDateTime;
29. 
30.     public LoanId() {
31.         super();
32.     }
33. 
34.     public LoanId(Book book, Customer customer) {
35.         super();
36.         this.book = book;
37.         this.customer = customer;
38.         this.creationDateTime = LocalDateTime.now();
39.     }
40. 
41.     @ManyToOne
42.     public Book getBook() {
43.         return book;
44.     }
45. 
46.     public void setBook(Book bbok) {
47.         this.book = bbok;
48.     }
49. 
50.     @ManyToOne
51.     public Customer getCustomer() {
52.         return customer;
53.     }
54. 
55.     public void setCustomer(Customer customer) {
56.         this.customer = customer;
57.     }
58. 
59.     @Column(name = "CREATION_DATE_TIME")
60.     public LocalDateTime getCreationDateTime() {
61.         return creationDateTime;
62.     }
63. 
64.     public void setCreationDateTime(LocalDateTime creationDateTime) {
65.         this.creationDateTime = creationDateTime;
66.     }
67. 
68.     // + les méthodes hashCode() et equals() que vous retrouverez dans les sources de cet
 article
69. }
```

```
1. package com.gkemayo.library.loan;
2. 
3. import java.io.Serializable;
4. import java.time.LocalDate;
5. 
6. import javax.persistence.AssociationOverride;
7. import javax.persistence.AssociationOverrides;
8. import javax.persistence.Column;
9. import javax.persistence.EmbeddedId;
10. import javax.persistence.Entity;
11. import javax.persistence.EnumType;
12. import javax.persistence.Enumerated;
13. import javax.persistence.JoinColumn;
14. import javax.persistence.Table;
15. 
16. @Entity
```

```
@Table(name = "LOAN")
@AssociationOverrides({
@AssociationOverride(name = "pk.book", joinColumns = @JoinColumn(name = "BOOK_ID")),
@AssociationOverride(name = "pk.customer", joinColumns = @JoinColumn(name = "CUSTOMER_ID"))
})
public class Loan implements Serializable {

    /**
     *
     */
    private static final long serialVersionUID = 144293603488149743L;

    private LoanId pk = new LoanId();

    private LocalDate beginDate;

    private LocalDate endDate;

    private LoanStatus status;

    @EmbeddedId
    public LoanId getPk() {
        return pk;
    }

    public void setPk(LoanId pk) {
        this.pk = pk;
    }

    @Column(name = "BEGIN_DATE", nullable = false)
    public LocalDate getBeginDate() {
        return beginDate;
    }

    public void setBeginDate(LocalDate beginDate) {
        this.beginDate = beginDate;
    }

    @Column(name = "END_DATE", nullable = false)
    public LocalDate getEndDate() {
        return endDate;
    }

    public void setEndDate(LocalDate endDate) {
        this.endDate = endDate;
    }

    @Enumerated(EnumType.STRING)
    @Column(name = "STATUS")
    public LoanStatus getStatus() {
        return status;
    }

    public void setStatus(LoanStatus status) {
        this.status = status;
    }

    // + les méthodes hashCode() et equals() que vous retrouverez dans les sources de cet
 article
}
```

Remarque : les différentes classes Java ci-dessus comportent plusieurs annotations :

- ***@Entity***, qui permet à hibernate/JPA de les considérer comme des ORM (Object Relational Mapping) devant transporter des données entre l'application et la base de données ;
- ***@Table***, qui permet de mapper cet ORM sur une table physique en base de données ;
- ***@Id***, qui permet de consacrer un attribut de la classe comme étant sa clef primaire ; et ***@GeneratedValue*** pour la stratégie de génération des valeurs de cette clef primaire ;
- ***@Column***, pour le mapping d'un attribut de classe à une colonne de table en base de données ;

- ***@AssociationOverrides***, ***@Embeddable*** et ***@EmbeddedId***, pour la gestion de clef primaire composée et de migration de clef étrangère ;
- ***@ManyToOne***, ***@OneToMany*** et ***@JoinColumn***, pour la gestion des associations *n-1*, *1-n* entre deux entités.

VIII - Les Dao Spring Data JPA

Dans notre application, nous avons choisi pour nos classes **DAO** (Data Access Object) d'utiliser le framework *Spring Data JPA* pour la gestion d'accès aux données vers sa base H2.

VIII-A - Qu'est-ce que Spring Data JPA ?

Spring Data JPA est un framework qui a été construit pour faciliter le développement de la couche DAO chargée de la persistance et du requêtage des données dans une base de données relationnelle. C'est une sorte de surcouche ou alors une implémentation de la spécification JPA 2 (Java Persistence API). À cet effet, il propose des fonctionnalités standardisées pour la réalisation des opérations *CRUD (Create, Read, Update, Delete)* sur une base de données. En outre, il propose des fonctionnalités dédiées au tri, à la pagination, à la gestion transactionnelle, etc.

Il se distingue d'*hibernate* (qui est une autre implémentation de JPA) par le fait qu'il permet de désalourdir et de détacher le développeur des tâches de configuration liées à la gestion de la persistance des données, tant au niveau applicatif qu'au niveau plus fin des classes DAO. Cela permet ainsi au développeur de se focaliser sur la réalisation des règles métier plutôt que sur les tâches purement techniques. Ceux qui ont déjà eu à monter et utiliser hibernate ou JPA dans une application Java (surtout dans leur version xml) savent bien ce à quoi renvoient les configurations évoquées. Pour information, notez que JPA est une spécification, mais dispose également de sa propre implémentation du même nom. Nous ne nous étendrons pas plus dans cet article sur les problématiques de configuration hibernate/JPA.

Note : Spring Data JPA et hibernate sont tout à fait compatibles et peuvent cohabiter dans une même application.

- Dans un contexte où une application n'est pas *Spring Bootée* (c'est-à-dire, non générée par Spring Boot), pour utiliser *Spring Data JPA*, il suffit d'injecter la dépendance ci-dessous et d'utiliser l'annotation ***@EnableJpaRepositories*** sur la classe de configuration qui créera les beans Spring chargés de la persistance des données (DAO) :

```
1. <dependency>
2.     <groupId>org.springframework.data</groupId>
3.     <artifactId>spring-data-jpa</artifactId>
4.     <version>${version-souhaitée}</version>
5. </dependency>
```

- Dans un contexte Spring Boot où le framework *Spring Data JPA* est chargé via le starter *spring-boot-starter-data-jpa* et autoconfiguré via l'annotation *@SpringBootApplication* sur la classe de démarrage, nous n'avons rien à faire concernant une quelconque configuration.

Dans tous les cas, pour qu'une classe DAO de votre application soit prise en charge par le framework *Spring Data JPA* et considérée comme une couche d'accès aux données devant bénéficier de tous les services et facilités que propose ce dernier, elle devra respecter les conditions suivantes :

- être une interface Java ;
- porter l'annotation **@Repository**, pour permettre à Spring de l'injecter comme bean dans l'application;
- étendre l'une des interfaces suivantes : **Repository**, **CrudRepository**, **JpaRepository** ou ***PagingAndSortingRepository***.

*Note : il existe une hiérarchie ascendante entre les interfaces **Repository**, **CrudRepository** et **JpaRepository** qui chacune ajoute de nouvelles fonctionnalités au bénéfice du développeur. Mais ce n'est pas l'objet dans cet article de s'y attarder.*

Le bout de code ci-dessous représente un exemple de classe Dao *Spring Data JPA*, nommée *MyDao*, dans laquelle *T* correspond à l'entité hibernate concernée par les requêtes dans cette classe et *ID* correspond au type de données de sa clef primaire :

```
1. @Repository
2. public interface MyDao extends JpaRepository<T, ID> {
3.
4. }
```

*Lorsqu'une interface étend **JpaRepository**, elle hérite de toutes les fonctionnalités CRUD (Create, Read, Update, Delete) que ce dernier fournit. On peut en citer quelques-unes : save(), saveAll(), delete(), deleteById(), deleteAll(), findById(), findAll(), exists(), existsById(), etc. C'est alors Spring Data JPA qui se chargera de créer pour nous une classe d'implémentation de notre DAO.*

Nous l'avons dit, il existe un large panel de fonctionnalités que propose *Spring Data JPA* pour la gestion des données. Nous pensons que les plus immédiates que vous pourrez avoir besoin sont celles focalisées sur les opérations *CRUD*. Les fonctionnalités d'insertion (*Create*), de mise à jour (*Update*) et de suppression (*Delete*) ne posent en général pas de problème de variabilité dans leur utilisation. Ce qui n'est pas le cas pour la fonctionnalité de lecture (*Read*) de données en base où il existe plusieurs politiques. Nous nous arrêtons donc sur les trois politiques de lecture de données suivantes :

- l'utilisation de l'annotation **@NamedQuery** pour la construction des requêtes nommées. La requête nommée se marque sur l'entité concernée puis invoquée par la DAO à travers son nom. Exemple :

```
1. @Entity
2. @NamedQuery(name = "T.getAll", query = "select t from T t")
3. public class T {
4. }
```

```
1. @Repository
2. public interface MyDao extends JpaRepository<T, ID> {
3.     public List<T> getAll();
4. }
```

- l'utilisation de l'annotation **@Query** sur une méthode, directement dans la classe DAO. Exemple :

```
1. @Repository
2. public interface MyDao extends JpaRepository<T, ID> {
3.
4.     @Query(query = "select t from T t")
5.     public List<T> getAll();
6. }
```

- l'utilisation des méthodes prédéfinies qui doivent respecter un certain format afin de permettre à *Spring Data JPA* de générer la requête JPQL par déduction du nom de la méthode et de ses paramètres d'entrées. Exemple :

```
1. @Repository
2. public interface MyDao extends JpaRepository<T, ID> {
3.
4.     public List<T> findAll();
5.     public List<T> findByXxxAndYyyIgnoreCase(String xXX, String yYYY);
6.
```

```
7. }
```

En général, toute méthode dans une classe *Spring Data JPA* qui n'est marquée d'aucune annotation et qui commence par le préfixe ***find*** ou ***findBy*** est prise en charge par ce dernier. Dans cet exemple, il génèrera à l'invocation de chacune de ces méthodes les requêtes suivantes :

- findAll() : *select t from T t* ;
- findByXxxAndYyyIgnoreCase(String xXX, String yYYY) : *select t from T t where lower(t.xXX) = lower(xXX) and lower(t.yYY) = lower(yYYY)*. La casse est ignorée lors de la comparaison sur les paramètres d'entrée.

S'il n'arrive pas générer la requête pour une méthode *find*, une exception de type ***InvalidJpaQueryMethodException*** est levée.

Pour terminer cette section, nous vous invitons à consulter le **guide** *Spring Data JPA* pour plus d'approfondissements.

VIII-B - Les classes Spring Data JPA de l'application

Pour notre application de gestion de la bibliothèque, ***Library***, nous avons donc quatre classes Spring Data JPA, une pour chaque domaine :

- domaine Book -> *IbookDao* ;
- domaine Customer -> *IcustomerDao* ;
- domaine Category -> *IcategoryDao ;*
- domaine Loan -> *ILoanDao*.

Revenons aux users stories de notre application listées à la section **I-B**. Nous pouvons remarquer que les problématiques se réduisent à la mise en place des fonctions de création/recherche/mise-à-jour/suppression de livres/clients/prêts. Comme nous l'avons expliqué dans les sections ci-dessus, nos traitements de création/mise-à-jour/suppression peuvent directement être délégués aux fonctionnalités *Spring Data JPA* préfournies. Nos classes DAO se focaliseront donc sur les traitements de recherche complexes. Nous exposons ci-dessous deux exemples de classes DAO de l'application Library. Vous pourrez consulter le reste dans le *code source* téléchargeable à la fin de cet article.

```
1. package com.gkemayo.library.book;
2. import java.util.List;
3. import org.springframework.data.jpa.repository.JpaRepository;
4. import org.springframework.data.jpa.repository.Query;
5. import org.springframework.stereotype.Repository;
6.
7. @Repository
8. public interface IBookDao extends JpaRepository<Book, Integer> {
9.
10.     public Book findByIsbnIgnoreCase(String isbn);
11.
12.     public List<Book> findByTitleLikeIgnoreCase(String title);
13.
14.        @Query("SELECT b "
15.             + "FROM Book b "
16.             + "INNER JOIN b.category cat "
17.             + "WHERE cat.code = :code"
18.           )
19.     public List<Book> findByCategory(@Param("code") String codeCategory);
20. }
```

```
1. package com.gkemayo.library.loan;
2.
3. import java.time.LocalDate;
4. import java.util.List;
5. import org.springframework.data.jpa.repository.JpaRepository;
```

```
6. import org.springframework.data.jpa.repository.Query;
7. import org.springframework.stereotype.Repository;
8.
9. @Repository
10. public interface ILoanDao extends JpaRepository<Loan, Integer> {
11.
12.    public List<Loan> findByEndDateBefore(LocalDate maxEndDate);
13.
14.        @Query("SELECT lo "
15.             + "FROM Loan lo "
16.             + "INNER JOIN lo.pk.customer c "
17.             + "WHERE UPPER(c.email) = UPPER(?1) "
18.             + "   AND lo.status = ?2 ")
19.     public List<Loan> getAllOpenLoansOfThisCustomer(String email, LoanStatus status);
20.
21.        @Query("SELECT lo "
22.             + "FROM Loan lo "
23.             + "INNER JOIN lo.pk.book b "
24.             + "INNER JOIN lo.pk.customer c "
25.             + "WHERE b.id =    ?1 "
26.             + "   AND c.id = ?2 "
27.             + "   AND lo.status = ?3 ")
28.     public Loan getLoanByCriteria(Integer bookId, Integer customerId, LoanStatus status);
29. }
```

IX - Les classes de services

Les classes de services de notre application Library sont celles qui font directement appel aux DAO présentés précédemment, afin de récupérer les données, les traiter si nécessaire et les faire transiter vers les services de niveau supérieur qui en ont fait la demande (en l'occurrence les **contrôleurs REST** que nous présenterons dans la section suivante). Il s'agit donc d'une classe intermédiaire entre la classe DAO et la classe Contrôleur qu'il faut implémenter afin de respecter la hiérarchie des appels dans une application de type **SOA** (Service Oriented Architecture).

Comme dans la section précédente, nous vous présentons le contenu des classes chargées de la gestion du domaine des livres (Book) et des prêts (Loan). Ceux des domaines Catégorie (Category) et Client (Customer) sont totalement similaires et vous pourrez les consulter dans le *code source* téléchargeable à la fin de cet article.

- Domaine Book :

```
1. package com.gkemayo.library.book;
2.
3. import java.util.List;
4.
5. public interface IBookService {
6.
7.     public Book saveBook(Book book);
8.
9.     public Book updateBook(Book book);
10.
11.     public void deleteBook(Integer bookId);
12.
13.     public List<Book> findBooksByTitleOrPartTitle(String title);
14.
15.     public Book findBookByIsbn(String isbn);
16.
17.     public boolean checkIfIdexists(Integer id);
18.
19.     public List<Book> getBooksByCategory(String codeCategory);
20. }
```

```
1. package com.gkemayo.library.book;
2.
3. import java.util.List;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.stereotype.Service;
6. import org.springframework.transaction.annotation.Transactional;
```

```
7.
8. @Service("bookService")
9. @Transactional
10. public class BookServiceImpl implements IBookService {
11.
12.     @Autowired
13.     private IBookDao bookDao;
14.
15.     @Override
16.     public Book saveBook(Book book) {
17.         return bookDao.save(book);
18.     }
19.
20.     @Override
21.     public Book updateBook(Book book) {
22.         return bookDao.save(book);
23.     }
24.
25.     @Override
26.     public void deleteBook(Integer bookId) {
27.         bookDao.deleteById(bookId);
28.     }
29.
30.     @Override
31.     public boolean checkIfIdExists(Integer id) {
32.         return bookDao.existsById(id);
33.     }
34.
35.     @Override
36.     public List<Book> findBooksByTitleOrPartTitle(String title) {
37.         return
 bookDao.findByTitleLikeIgnoreCase((new StringBuilder()).append("%").append(title).append("%").toString());
38.     }
39.
40.     @Override
41.     public Book findBookByIsbn(String isbn) {
42.         return bookDao.findByIsbnIgnoreCase(isbn);
43.     }
44.
45.     @Override
46.     public List<Book> getBooksByCategory(String codeCategory) {
47.         return bookDao.findByCategory(codeCategory);
48.     }
49. }
```

- Domaine Loan :

```
1. package com.gkemayo.library.loan;
2.
3. import java.time.LocalDate;
4. import java.util.List;
5.
6. public interface ILoanService {
7.
8.     public List<Loan> findAllLoansByEndDateBefore(LocalDate maxEndDate);
9.
10.     public List<Loan> getAllOpenLoansOfThisCustomer(String email, LoanStatus status);
11.
12.     public Loan getOpenedLoan(SimpleLoanDTO simpleLoanDTO);
13.
14.     public boolean checkIfLoanExists(SimpleLoanDTO simpleLoanDTO);
15.
16.     public Loan saveLoan(Loan loan);
17.
18.     public void closeLoan(Loan loan);
19. }
```

```
1. package com.gkemayo.library.loan;
2.
3. import java.time.LocalDate;
```

```
4. import java.util.List;
5. import org.springframework.beans.factory.annotation.Autowired;
6. import org.springframework.stereotype.Service;
7. import org.springframework.transaction.annotation.Transactional;
8.
9. @Service("loanService")
10. @Transactional
11. public class LoanServiceImpl implements ILoanService {
12.
13.     @Autowired
14.     private ILoanDao loanDao;
15.
16.     @Override
17.     public List<Loan> findAllLoansByEndDateBefore(LocalDate maxEndDate) {
18.         return loanDao.findByEndDateBefore(maxEndDate);
19.     }
20.
21.     @Override
22.     public List<Loan> getAllOpenLoansOfThisCustomer(String email, LoanStatus status) {
23.         return loanDao.getAllOpenLoansOfThisCustomer(email, status);
24.     }
25.
26.     @Override
27.     public Loan getOpenedLoan(SimpleLoanDTO simpleLoanDTO) {
28.         return loanDao.getLoanByCriteria(simpleLoanDTO.getBookId(),
 simpleLoanDTO.getCustomerId(), LoanStatus.OPEN);
29.     }
30.
31.     @Override
32.     public boolean checkIfLoanExists(SimpleLoanDTO simpleLoanDTO) {
33.         Loan loan = loanDao.getLoanByCriteria(simpleLoanDTO.getBookId(),
 simpleLoanDTO.getCustomerId(), LoanStatus.OPEN);
34.         if(loan != null) {
35.             return true;
36.         }
37.         return false;
38.     }
39.
40.     @Override
41.     public Loan saveLoan(Loan loan) {
42.         return loanDao.save(loan);
43.     }
44.
45.     /**
46.      * On fera de la suppression logique, car le statut de l'objet Loan est positionné à
 CLOSE.
47.      */
48.     @Override
49.     public void closeLoan(Loan loan) {
50.         loanDao.save(loan);
51.     }
52. }
```

Dans ces différentes classes, nous notons l'utilisation des annotations suivantes :

- ***@Service***, qui permet à Spring de considérer la classe qui la porte comme un bean qu'il créera au démarrage de l'application ;
- ***@Transactional*** au niveau classe, qui ordonne à spring de traiter toutes les méthodes publiques de la classe en mode transactionnel ;
- ***@Autowired***, qui permet à Spring d'invoquer et d'injecter un bean « supposé » existant dans le contexte de la classe appelante.

LoanStatus est une simple enum Java contenant les valeurs ***OPEN*** et ***CLOSE***. Et **SimpleLoanDTO** est un POJO contenant les champs *bookId*, *customerId*, *beginDate* et *endDate* ainsi que leurs getter/setter.

X - Les contrôleurs REST

X-A - Quelques notions sur les API RESTful

Dans un système distribué où les applications ont besoin d'intercommuniquer pour s'échanger des données, il existe plusieurs moyens pour y parvenir. Parmi ceux-ci, nous avons les moyens de communication synchrone implémentés par des technologies telles que **RMI** (Remote Method Invocation), **CORBA** (Common Object Request Broker Architecture), **SOAP** (Simple Object Access Protocol) et **REST** (REspresentional State Transfer) et les moyens de communication asynchrone implémentés par des technologies de type **JMS** (Java Messaging Service) ou des architectures de type **CQRS** (Command and Query Responsibility Segregation) aussi appelées **Event Sourcing**.

Dans les applications web qui sont donc de type client/serveur, la technologie REST est celle qui est la plus utilisée aujourd'hui (en 2019) à raison de la simplicité de sa mécanique. En effet, REST s'appuie sur le protocole HTTP pour assurer la communication entre un client et un serveur. Il correspond donc à une *API* (Application Programming Interface) qui utilise et étend les méthodes HTTP pour standardiser les moyens de communication entre client et serveur. Les méthodes HTTP -- aussi appelées *verbes* HTTP -- les plus utilisées sont :

- ***GET***, dédiée à la lecture d'une ressource exposée sur un serveur ;
- ***POST***, permet la création d'un ou de plusieurs objets sur un serveur au travers d'une ressource dédiée ;
- ***PUT***, permet la mise à jour d'un ou de plusieurs objets sur un serveur au travers d'une ressource dédiée ;
- ***DELETE***, permet la suppression d'un ou de plusieurs objets sur un serveur au travers d'une ressource.

De façon prosaïque, nous pouvons voir une **ressource** comme tout élément construit sur le serveur et qui peut n'être accédée qu'à travers un unique chemin appelé **URI** (Uniform Resource Identifier) afin de récupérer, créer, mettre à jour ou supprimer des données. Dans les applications web Java, ces ressources sont généralement représentées par des ***méthodes publiques*** qualifiées de ***web services*** et implémentées au sein de classes appelées **contrôleur REST**. Tout contrôleur REST exposant des web services et respectant les principes de la spécification REST est qualifié d'*API RESTful*. Pour votre culture, il existe une échelle appelée **Model de Maturité de Richardson** qui spécifie un classement des niveaux d'une API dite RESTful.

Il existe plusieurs frameworks permettant d'implémenter des API RESTful dans nos applications. Les plus connus sont ***Jersey*** et ***Spring Webmvc*** qui proposent plusieurs annotations et fonctionnalités permettant de spécifier des ressources et leurs méthodes d'accès (GET, POST, PUT, DELETE, etc.) sur un serveur d'application.

Dans notre application *Library*, puisque nous utilisons Spring Boot et sommes donc en écosystème Spring, nous choisissons de réaliser nos contrôleurs REST via la dépendance *Spring Webmvc* qui a été automatiquement injectée par le starter *spring-boot-starter-web*.

X-B - Un petit détour sur Spring Mail

Notre application *Library*, conformément à la **User story 10** aura besoin d'envoyer des mails aux clients de la bibliothèque. L'implémentation de ce service d'envoi de mail sera réalisée dans l'un des contrôleurs REST dont nous afficherons le code source dans la section suivante. En attendant, nous vous présentons rapidement ci-dessous les préalables nécessaires pour faire du mailing avec Java Spring.

1. Il faut ajouter à votre application dans un fichier *properties* qui sera chargé par Spring, les ressources nécessaires. Exemple de l'application.properties exposé à la **section V** que nous reprécisons ci-dessous :

```
1. ############### DataSource Config #################
2. spring.mail.default-encoding=UTF-8
3. spring.mail.protocol=smtp
4. spring.mail.host=smtp.gmail.com
5. spring.mail.username=noreply.library.test@gmail.com
6. spring.mail.password=password1Test
7. spring.mail.port= 587
8. spring.mail.properties.mail.smtp.auth=true
```

```
spring.mail.properties.mail.smtp.starttls.enable=true
spring.mail.test-connection=false
#https://www.google.com/settings/security/lesssecureapps
```

Dans ce paramétrage, *UTF-8* correspond à l'encodage du texte contenu dans les mails. Le protocole de mailing utilisé est **SMTP** (Simple Message Transfer Protocol) avec comme serveur celui *gmail* de Google sur le port 587. Notre application utilisera le compte *noreply.library.test@gmail.com/passwordTest* pour envoyer ses mails dans un contexte sécurisé (*spring.mail.properties.mail.smtp.auth=true*). Enfin, la connexion de l'application au serveur smtp utilisera le protocole TLS (*spring.mail.properties.mail.smtp.starttls.enable=true*). Pour information, pour permettre à une application/robot d'envoyer des mails via gmail, il faut se connecter une première fois manuellement dans le compte gmail concerné, puis aller à l'adresse suivante **https://www.google.com/settings/security/lesssecureapps** pour désactiver la sécurité manuelle.

2. Injection de la dépendance Maven pour Spring Mail :

- Si votre application n'est pas *Spring Bootée*, vous devez ajouter la dépendance suivante dans le pom.xml :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${version-souhaitée}</version>
</dependency>
```

- Si votre application est générée via *Spring Boot* comme l'application *Library*, vous devez ajouter le starter suivant :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mail</artifactId>
    //Spring Boot chargera lui même la version
</dependency>
```

3. Enfin, dans votre classe Java qui va gérer l'envoi de mail, il faut injecter, via l'annotation *@Autowired*, le bean **JavaMailSender**, fourni par Spring Mail. Il ne nous restera plus qu'à utiliser sa méthode *send()* pour envoyer effectivement le mail. Exemple :

```
public class MailSender {

    @Autowired
    private JavaMailSender javaMailSender;

    public void sendMail() throws MailException {
      //... créer ici l'objet message (de type SimpleMailMessage ou MimeMessage) à envoyer
      javaMailSender.send(message);
    }
}
```

X-C - Quelques contrôleurs Rest de l'application

Nous vous exposons ci-dessous, les classes *CustomerRestController* et *LoanRestController*. Vous pouvez consulter les contrôleurs *BookRestController* et *CategoryRestController*, similaires aux deux premiers, directement dans le code source téléchargable à la fin de cet article.

```
package com.gkemayo.library.customer;

import java.time.LocalDate;
import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;

import org.modelmapper.ModelMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```
11. import org.springframework.beans.factory.annotation.Autowired;
12. import org.springframework.data.domain.Page;
13. import org.springframework.http.HttpStatus;
14. import org.springframework.http.ResponseEntity;
15. import org.springframework.mail.MailException;
16. import org.springframework.mail.SimpleMailMessage;
17. import org.springframework.mail.javamail.JavaMailSender;
18. import org.springframework.util.CollectionUtils;
19. import org.springframework.util.StringUtils;
20. import org.springframework.web.bind.annotation.DeleteMapping;
21. import org.springframework.web.bind.annotation.GetMapping;
22. import org.springframework.web.bind.annotation.PathVariable;
23. import org.springframework.web.bind.annotation.PostMapping;
24. import org.springframework.web.bind.annotation.PutMapping;
25. import org.springframework.web.bind.annotation.RequestBody;
26. import org.springframework.web.bind.annotation.RequestMapping;
27. import org.springframework.web.bind.annotation.RequestParam;
28. import org.springframework.web.bind.annotation.RestController;
29. import org.springframework.web.util.UriComponentsBuilder;
30.
31. import io.swagger.annotations.Api;
32. import io.swagger.annotations.ApiOperation;
33. import io.swagger.annotations.ApiResponse;
34. import io.swagger.annotations.ApiResponses;
35.
36. @RestController
37. @RequestMapping("/rest/customer/api")
38. public class CustomerRestController {
39.
40.     public static final Logger LOGGER =
 LoggerFactory.getLogger(CustomerRestController.class);
41.
42.     @Autowired
43.     private CustomerServiceImpl customerService;
44.
45.     @Autowired
46.     private JavaMailSender javaMailSender;
47.
48.     /**
49.      * Ajoute un nouveau client dans la base de données H2. Si le client existe déjà, on
 retourne un code indiquant que la création n'a pas abouti.
50.      * @param customerDTORequest
51.      * @return
52.      */
53.     @PostMapping("/addCustomer")
54.     public ResponseEntity<CustomerDTO> createNewCustomer(@RequestBody CustomerDTO
 customerDTORequest) {
55.         //, UriComponentsBuilder uriComponentBuilder
56.         Customer existingCustomer =
 customerService.findCustomerByEmail(customerDTORequest.getEmail());
57.         if (existingCustomer != null) {
58.             return new ResponseEntity<CustomerDTO>(HttpStatus.CONFLICT);
59.         }
60.         Customer customerRequest = mapCustomerDTOToCustomer(customerDTORequest);
61.         customerRequest.setCreationDate(LocalDate.now());
62.         Customer customerResponse = customerService.saveCustomer(customerRequest);
63.         if (customerResponse != null) {
64.             CustomerDTO customerDTO = mapCustomerToCustomerDTO(customerResponse);
65.             return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.CREATED);
66.         }
67.         return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_MODIFIED);
68.     }
69.
70.     /**
71.      * Met à jour les données d'un client dans la base de données H2. Si le client n'est pas
 retrouvé, on retourne un code indiquant que la mise à jour n'a pas abouti.
72.      * @param customerDTORequest
73.      * @return
74.      */
75.     @PutMapping("/updateCustomer")
76.     public ResponseEntity<CustomerDTO> updateCustomer(@RequestBody CustomerDTO
 customerDTORequest) {
```

```
77.         //, UriComponentsBuilder uriComponentBuilder
78.         if (!customerService.checkIfIdexists(customerDTORequest.getId())) {
79.             return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_FOUND);
80.         }
81.         Customer customerRequest = mapCustomerDTOToCustomer(customerDTORequest);
82.         Customer customerResponse = customerService.updateCustomer(customerRequest);
83.         if (customerResponse != null) {
84.             CustomerDTO customerDTO = mapCustomerToCustomerDTO(customerResponse);
85.             return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.OK);
86.         }
87.         return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_MODIFIED);
88.     }
89.
90.     /**
91.      * Supprime un client dans la base de données H2. Si le client n'est pas retrouvé, on
 retourne le Statut HTTP NO_CONTENT.
92.      * @param customerId
93.      * @return
94.      */
95.     @DeleteMapping("/deleteCustomer/{customerId}")
96.     public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) {
97.         customerService.deleteCustomer(customerId);
98.         return new ResponseEntity<String>(HttpStatus.NO_CONTENT);
99.     }
100.
101.     /**
102.      * Retourne le client ayant l'adresse email passée en paramètre.
103.      * @param email
104.      * @return
105.      */
106.     @GetMapping("/searchByEmail")
107.     public ResponseEntity<CustomerDTO> searchCustomerByEmail(@RequestParam("email") String
 email) {
108.         //, UriComponentsBuilder uriComponentBuilder
109.         Customer customer = customerService.findCustomerByEmail(email);
110.         if (customer != null) {
111.             CustomerDTO customerDTO = mapCustomerToCustomerDTO(customer);
112.             return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.OK);
113.         }
114.         return new ResponseEntity<CustomerDTO>(HttpStatus.NO_CONTENT);
115.     }
116.
117.     /**
118.      * Retourne la liste des clients ayant le nom passé en paramètre.
119.      * @param lastName
120.      * @return
121.      */
122.     @GetMapping("/searchByLastName")
123.     public ResponseEntity<List<CustomerDTO>> searchBookByLastName(@RequestParam("lastName")
 String lastName) {
124.         //,    UriComponentsBuilder uriComponentBuilder
125.         List<Customer> customers = customerService.findCustomerByLastName(lastName);
126.         if (customers != null && !CollectionUtils.isEmpty(customers)) {
127.             List<CustomerDTO> customerDTOs = customers.stream().map(customer -> {
128.                 return mapCustomerToCustomerDTO(customer);
129.             }).collect(Collectors.toList());
130.             return new ResponseEntity<List<CustomerDTO>>(customerDTOs, HttpStatus.OK);
131.         }
132.         return new ResponseEntity<List<CustomerDTO>>(HttpStatus.NO_CONTENT);
133.     }
134.
135.     /**
136.      * Envoie un mail à un client. L'objet MailDTO contient l'identifiant et l'email du
 client concerné, l'objet du mail et le contenu du message.
137.      * @param loanMailDto
138.      * @param uriComponentBuilder
139.      * @return
140.      */
141.     @PutMapping("/sendEmailToCustomer")
142.     public ResponseEntity<Boolean> sendMailToCustomer(@RequestBody MailDTO loanMailDto,
 UriComponentsBuilder uriComponentBuilder) {
143.
```

```
144.         Customer customer = customerService.findCustomerById(loanMailDto.getCustomerId());
145.         if (customer == null) {
146.             String errorMessage = "The selected Customer for sending email is not found in
 the database";
147.             LOGGER.info(errorMessage);
148.             return new ResponseEntity<Boolean>(false, HttpStatus.NOT_FOUND);
149.         } else if (customer != null && StringUtils.isEmpty(customer.getEmail())) {
150.             String errorMessage = "No existing email for the selected Customer for sending
 email to";
151.             LOGGER.info(errorMessage);
152.             return new ResponseEntity<Boolean>(false, HttpStatus.NOT_FOUND);
153.         }
154.
155.         SimpleMailMessage mail = new SimpleMailMessage();
156.         mail.setFrom(loanMailDto.MAIL_FROM);
157.         mail.setTo(customer.getEmail());
158.         mail.setSentDate(new Date());
159.         mail.setSubject(loanMailDto.getEmailSubject());
160.         mail.setText(loanMailDto.getEmailContent());
161.
162.         try {
163.             javaMailSender.send(mail);
164.         } catch (MailException e) {
165.             return new ResponseEntity<Boolean>(false, HttpStatus.FORBIDDEN);
166.         }
167.
168.         return new ResponseEntity<Boolean>(true, HttpStatus.OK);
169.     }
170.
171.     /**
172.      * Transforme une entity Customer en un POJO CustomerDTO
173.      *
174.      * @param customer
175.      * @return
176.      */
177.     private CustomerDTO mapCustomerToCustomerDTO(Customer customer) {
178.         ModelMapper mapper = new ModelMapper();
179.         CustomerDTO customerDTO = mapper.map(customer, CustomerDTO.class);
180.         return customerDTO;
181.     }
182.
183.     /**
184.      * Transforme un POJO CustomerDTO en une entity Customer
185.      *
186.      * @param customerDTO
187.      * @return
188.      */
189.     private Customer mapCustomerDTOToCustomer(CustomerDTO customerDTO) {
190.         ModelMapper mapper = new ModelMapper();
191.         Customer customer = mapper.map(customerDTO, Customer.class);
192.         return customer;
193.     }
194. }
```

```
1. package com.gkemayo.library.loan;
2.
3. import java.time.LocalDate;
4. import java.time.LocalDateTime;
5. import java.util.Collections;
6. import java.util.List;
7. import java.util.function.Function;
8. import java.util.stream.Collectors;
9.
10. import org.slf4j.Logger;
11. import org.slf4j.LoggerFactory;
12. import org.springframework.beans.factory.annotation.Autowired;
13. import org.springframework.format.annotation.DateTimeFormat;
14. import org.springframework.http.HttpStatus;
15. import org.springframework.http.ResponseEntity;
16. import org.springframework.util.CollectionUtils;
17. import org.springframework.web.bind.annotation.GetMapping;
```

```
18. import org.springframework.web.bind.annotation.PostMapping;
19. import org.springframework.web.bind.annotation.RequestBody;
20. import org.springframework.web.bind.annotation.RequestMapping;
21. import org.springframework.web.bind.annotation.RequestParam;
22. import org.springframework.web.bind.annotation.RestController;
23. import org.springframework.web.util.UriComponentsBuilder;
24.
25. import com.gkemayo.library.book.Book;
26. import com.gkemayo.library.customer.Customer;
27.
28. import io.swagger.annotations.Api;
29. import io.swagger.annotations.ApiOperation;
30. import io.swagger.annotations.ApiResponse;
31. import io.swagger.annotations.ApiResponses;
32.
33. @RestController
34. @RequestMapping("/rest/loan/api")
35. public class LoanRestController {
36.
37.     public static final Logger LOGGER = LoggerFactory.getLogger(LoanRestController.class);
38.
39.     @Autowired
40.     private LoanServiceImpl loanService;
41.
42.     /**
43.      * Retourne l'historique des prêts en cours dans la bibliothèque jusqu'à une certaine
 date maximale.
44.      * @param maxEndDateStr
45.      * @return
46.      */
47.     @GetMapping("/maxEndDate")
48.     public
 ResponseEntity<List<LoanDTO>> searchAllBooksLoanBeforeThisDate(@RequestParam("date") String
 maxEndDateStr) {
49.         List<Loan> loans =
 loanService.findAllLoansByEndDateBefore(LocalDate.parse(maxEndDateStr));
50.         // on retire tous les élts null que peut contenir cette liste => pour éviter les NPE
 par la suite
51.         loans.removeAll(Collections.singleton(null));
52.         List<LoanDTO> loanInfosDtos = mapLoanDtosFromLoans(loans);
53.         return new ResponseEntity<List<LoanDTO>>(loanInfosDtos, HttpStatus.OK);
54.     }
55.
56.     /**
57.      * Retourne la liste des prêts en cours d'un client.
58.      * @param email
59.      * @return
60.      */
61.     @GetMapping("/customerLoans")
62.     public
 ResponseEntity<List<LoanDTO>> searchAllOpenedLoansOfThisCustomer(@RequestParam("email") String
 email) {
63.         List<Loan> loans = loanService.getAllOpenLoansOfThisCustomer(email, LoanStatus.OPEN);
64.         // on retire tous les élts null que peut contenir cette liste => pour éviter les NPE
 par la suite
65.         loans.removeAll(Collections.singleton(null));
66.         List<LoanDTO> loanInfosDtos = mapLoanDtosFromLoans(loans);
67.         return new ResponseEntity<List<LoanDTO>>(loanInfosDtos, HttpStatus.OK);
68.     }
69.
70.     /**
71.      * Ajoute un nouveau prêt dans la base de données H2.
72.      * @param simpleLoanDTORequest
73.      * @param uriComponentBuilder
74.      * @return
75.      */
76.     @PostMapping("/addLoan")
77.     public ResponseEntity<Boolean> createNewLoan(@RequestBody SimpleLoanDTO
 simpleLoanDTORequest,
78.             UriComponentsBuilder uriComponentBuilder) {
79.         boolean isLoanExists = loanService.checkIfLoanExists(simpleLoanDTORequest);
80.         if (isLoanExists) {
```

```
81.             return new ResponseEntity<Boolean>(false, HttpStatus.CONFLICT);
82.         }
83.         Loan LoanRequest = mapSimpleLoanDTOToLoan(simpleLoanDTORequest);
84.         Loan loan = loanService.saveLoan(LoanRequest);
85.         if (loan != null) {
86.             return new ResponseEntity<Boolean>(true, HttpStatus.CREATED);
87.         }
88.         return new ResponseEntity<Boolean>(false, HttpStatus.NOT_MODIFIED);
89.     }
90.
91.     /**
92.      * Clôture le prêt de livre d'un client.
93.      * @param simpleLoanDTORequest
94.      * @param uriComponentBuilder
95.      * @return
96.      */
97.     @PostMapping("/closeLoan")
98.     public ResponseEntity<Boolean> closeLoan(@RequestBody SimpleLoanDTO simpleLoanDTORequest,
99.             UriComponentsBuilder uriComponentBuilder) {
100.         Loan existingLoan = loanService.getOpenedLoan(simpleLoanDTORequest);
101.         if (existingLoan == null) {
102.             return new ResponseEntity<Boolean>(false, HttpStatus.NO_CONTENT);
103.         }
104.         existingLoan.setStatus(LoanStatus.CLOSE);
105.         Loan loan = loanService.saveLoan(existingLoan);
106.         if (loan != null) {
107.             return new ResponseEntity<Boolean>(true, HttpStatus.OK);
108.         }
109.         return new ResponseEntity<Boolean>(HttpStatus.NOT_MODIFIED);
110.     }
111.
112.     /**
113.      * Transforme une liste d'entités Lo Loan en liste LoanDTO.
114.      *
115.      * @param loans
116.      * @return
117.      */
118.     private List<LoanDTO> mapLoanDtosFromLoans(List<Loan> loans) {
119.
120.         Function<Loan, LoanDTO> mapperFunction = (loan) -> {
121.             // dans loanDTO on n'ajoute que les données nécessaires
122.             LoanDTO loanDTO = new LoanDTO();
123.             loanDTO.getBookDTO().setId(loan.getPk().getBook().getId());
124.             loanDTO.getBookDTO().setIsbn(loan.getPk().getBook().getIsbn());
125.             loanDTO.getBookDTO().setTitle(loan.getPk().getBook().getTitle());
126.
127.             loanDTO.getCustomerDTO().setId(loan.getPk().getCustomer().getId());
128.
 loanDTO.getCustomerDTO().setFirstName(loan.getPk().getCustomer().getFirstName());
129.             loanDTO.getCustomerDTO().setLastName(loan.getPk().getCustomer().getLastName());
130.             loanDTO.getCustomerDTO().setEmail(loan.getPk().getCustomer().getEmail());
131.
132.             loanDTO.setLoanBeginDate(loan.getBeginDate());
133.             loanDTO.setLoanEndDate(loan.getEndDate());
134.             return loanDTO;
135.         };
136.
137.         if (!CollectionUtils.isEmpty(loans)) {
138.             return loans.stream().map(mapperFunction).sorted().collect(Collectors.toList());
139.         }
140.         return null;
141.     }
142.
143.     /**
144.      * Transforme un SimpleLoanDTO en Loan avec les données minimalistes nécessaires
145.      *
146.      * @param loanDTORequest
147.      * @return
148.      */
149.     private Loan mapSimpleLoanDTOToLoan(SimpleLoanDTO simpleLoanDTO) {
150.         Loan loan = new Loan();
151.         Book book = new Book();
```

```
152.          book.setId(simpleLoanDTO.getBookId());
153.          Customer customer = new Customer();
154.          customer.setId(simpleLoanDTO.getCustomerId());
155.          LoanId loanId = new LoanId(book, customer);
156.          loan.setPk(loanId);
157.          loan.setBeginDate(simpleLoanDTO.getBeginDate());
158.          loan.setEndDate(simpleLoanDTO.getEndDate());
159.          loan.setStatus(LoanStatus.OPEN);
160.          return loan;
161.      }
162. }
```

Suite à la l'affichage de ces deux contrôleurs REST qui réalisent des opérations de création/modification/suppression/mise à jour d'un nouveau client/Prêts + l'envoi de mail à un client (*CustomerRestController*), vous remarquez qu'un bon nombre d'annotations Spring ont été utilisées. Elles sont possibles grâce au starter *spring-boot-starter-web* ajouté dans le pom.xl, qui à son tour injectera la dépendance *Spring Webmvc* correspondant à l'implémentation Spring d'API RESTful :

- **@RestController** : permet de marquer une classe comme étant une qui exposera des ressources appelées *web services ;*
- **@RequestMapping** : permet de spécifier l'URI d'un web service ou d'une classe représentant le Contrôleur REST ;
- **@GetMapping** : marque une ressource (et donc un web service) comme accessible par la méthode GET de HTTP. Spécifie aussi l'URI de la ressource ;
- **@PostMapping** : marque une ressource comme accessible par la méthode POST de HTTP. Spécifie aussi l'URI de la ressource ;
- **@PutMapping** : marque une ressource comme accessible par la méthode PUT de HTTP. Spécifie aussi l'URI de la ressource ;
- **@DeleteMapping** : marque une ressource comme accessible par la méthode DELETE de HTTP. Spécifie aussi l'URI de la ressource.

En appliquant les définitions données ci-dessus, nous observons que notre application *Library* expose :

- pour le contrôleur **CustomerRestController**, sept web services représentés par les méthodes publiques suivantes : *createNewCustomer, updateCustomer, deleteCustomer, searchCustomers, searchCustomerByEmail, searchBookByLastName, sendMailToCustomer* ;
- pour le contrôleur **LoanRestController**, quatre web services représentés par les méthodes suivantes : *searchAllBooksLoanBeforeThisDate, searchAllOpenedLoansOfThisCustomer, createNewLoan, closeLoan*.

Ces contrôleurs REST s'appuient sur les classes de services (exemple : *CustomerService*, *LoanService*) présentées plus haut pour faire appel aux classes DAO afin d'accéder à la base de données H2.

Dans les classes *CustomerRestController* et *LoanRestController*, nous avons utilisé d'autres annotations fournies par Spring qui concourent à la mise en place complète de web services. À savoir, **@RequestBody**, **@RequestParam**, **@PathVariable** qui sont les moyens de passage de paramètres du client vers le serveur. Nous avons aussi utilisé l'objet **ResponseEntity** qui joue l'effet inverse permettant ainsi au serveur d'encapsuler les données qu'il renverra au client. Nous n'entrerons pas plus dans les détails, il existe de nombreux articles sur Internet qui s'étendent de long en large sur ces notions.

Note : nous avons ajouté le framework ModelMapper pour gérer le transfert de données entre deux objets Java de même nature.

XI - Documenter et tester l'API REST avec Swagger

XI-A - Swagger et comment on le configure ?

Définition

Swagger est une méthode formelle de spécifications permettant de décrire et de produire une documentation au format JSON de l'API REST d'une application. En d'autres termes, il a pour objectif de regrouper au sein d'un même objet JSON, une description de chaque web service exposé dans votre application. Cette description peut porter sur : le type de méthode d'accès (GET, POST, etc.), l'URI du Web Service, les paramètres d'entrée et de sortie du web service, les codes HTTP de retour possibles, etc.
Swagger a été créé en 2011 et a vu au fil du temps sa spécification évoluer d'une version 1 à la version 3 actuelle. Il existe plusieurs frameworks qui implémentent swagger et qui proposent une interface web de l'objet JSON. Cette interface graphique est plus simple à la lecture et permet même de faire des tests réels de votre API REST.

Configuration

1- Dans l'application *Library*, nous utilisons le framework ***Springfox*** dont voici les dépendances à ajouter au pom.xml de l'application. Notez que Spring Boot ne propose aucun starter permettant d'inclure et d'utiliser directement swagger.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

2- Ajouter l'annotation ***@EnableSwagger2*** et configurer le bean ***Docket*** au niveau de la classe de démarrage, *LibraryApplication* de l'application. Voici l'exemple de ce que nous avons configuré :

```
1. package com.gkemayo.library;
2.
3. import org.springframework.boot.SpringApplication;
4. import org.springframework.boot.autoconfigure.SpringBootApplication;
5. import org.springframework.context.annotation.Bean;
6.
7. import springfox.documentation.builders.ApiInfoBuilder;
8. import springfox.documentation.builders.PathSelectors;
9. import springfox.documentation.builders.RequestHandlerSelectors;
10. import springfox.documentation.service.ApiInfo;
11. import springfox.documentation.service.Contact;
12. import springfox.documentation.spi.DocumentationType;
13. import springfox.documentation.spring.web.plugins.Docket;
14. import springfox.documentation.swagger2.annotations.EnableSwagger2;
15.
16. @SpringBootApplication
17. @EnableSwagger2
18. public class LibraryApplication {
19.
20.     public static void main(String[] args) {
21.        SpringApplication.run(LibraryApplication.class, args);
22.     }
23.
24.     @Bean
25.     public Docket api() {
26.         return new Docket(DocumentationType.SWAGGER_2)
27.           .select()
28.           .apis(RequestHandlerSelectors.basePackage("com.gkemayo.library"))
29.           .paths(PathSelectors.any())
```

```
30.             .build()
31.             .apiInfo(apiInfo());
32.     }
33.
34.     private ApiInfo apiInfo() {
35.         return new ApiInfoBuilder().title("Library Spring Boot REST API Documentation")
36.             .description("REST APIs For Managing Books loans in a Library")
37.             .contact(new Contact("Georges Kemayo", "https://
gkemayo.developpez.com/", "noreply.library.test@gmail.com"))
38.             .version("1.0")
39.             .build();
40.     }
41. }
```

Docket est le bean qui permet de configurer les données du rendu graphique de la documentation JSON de notre application. Il propose un ensemble d'attributs permettant de configurer : les packages contenant l'API REST concerné par la documentation Swagger (ici, *com.gkemayo.library*), les URI spécifiques des API REST à documenter (ici, tous les URI seront documentés : *PathSelectors.any()*), et d'autres informations générales à afficher (titre d'entête de la page, contact du/des contributeurs, version de l'API REST documentée, etc.).

Pour l'application *Library* déployée dans un Tomcat sur le port 8082, nous utilisons le lien *http://localhost:8082/library/swagger-ui.html#/* pour afficher la page web Swagger.

API REST Swagger
de l'application

Cette figure présente ainsi l'API REST de l'application *Library*. Les différents web services exposés sont repartis suivant les contrôleurs REST dans lesquels ils ont été définis. L'on peut remarquer la simplicité et la lisibilité au niveau de la description de chaque web service. Cette documentation a par conséquent l'avantage de permettre aux consommateurs de cette API REST de capter rapidement ce que fait chaque web service. Si l'on clique sur l'une des lignes représentant un web service, par exemple */addBook*, elle se déplie et affiche une fenêtre qui apporte plus de détails sur le web service (ses paramètres d'entrées, son objet retour, les potentiels codes retour HTTP qu'il peut renvoyer, etc.) et même son exécution à travers le bouton *Try it out*.

Swagger API
REST for Library

Enfin, si vous avez besoin de l'objet JSON correspond à l'API REST de l'application, il s'obtient sur l'URL relatif suivant : */v2/api-docs*. Exemple de l'application *Library* : *http://localhost:8082/library/v2/api-docs*.

XI-B - Exemple du CustomerRestController

Dans la précédente section, nous avons vu que le bean de type *Docket* permettait de configurer, entre autres, l'affichage des informations générales sur la page web Swagger, à savoir le titre, le contact des contributeurs de l'API REST, la version actuelle de l'API, etc.). Mais nous avons observé sur les captures affichées dans cette même section que les différents contrôleurs REST et leurs web services étaient aussi documentés. Mais comment est-ce possible ? me demanderez-vous. Eh bien, sachez que cela ne se fait pas automatiquement. La documentation Swagger des contrôleurs REST et des web services qui y sont exposés n'est possible que par le concours de quelques annotations. Pour le cas de l'application Library, nous avons utilisé les annotations suivantes :

- ***@Api*** : utilisée sur une classe de type contrôleur REST pour décrire globalement ce qu'elle fait ;
- ***@ApiOperation*** : utilisée sur un web service pour préciser ce qu'il fait exactement et aussi son objet retour ;

- ***@ApiResponses*** et ***@ApiResponse*** : pour décrire les différents codes retour HTTP que peut renvoyer un web service ;
- ***@ApiModel*** et ***@ApiModelProperty*** : utilisées pour décrire respectivement une classe POJO portant des données échangées entre le client et le serveur et pour décrire chaque attribut du POJO.

Voici donc, l'exemple de la classe *CustomerRestController* avec les annotations Swagger :

```
1. package com.gkemayo.library.customer;
2.
3. import java.time.LocalDate;
4. import java.util.Date;
5. import java.util.List;
6. import java.util.stream.Collectors;
7.
8. import org.modelmapper.ModelMapper;
9. import org.slf4j.Logger;
10. import org.slf4j.LoggerFactory;
11. import org.springframework.beans.factory.annotation.Autowired;
12. import org.springframework.data.domain.Page;
13. import org.springframework.http.HttpStatus;
14. import org.springframework.http.ResponseEntity;
15. import org.springframework.mail.MailException;
16. import org.springframework.mail.SimpleMailMessage;
17. import org.springframework.mail.javamail.JavaMailSender;
18. import org.springframework.util.CollectionUtils;
19. import org.springframework.util.StringUtils;
20. import org.springframework.web.bind.annotation.DeleteMapping;
21. import org.springframework.web.bind.annotation.GetMapping;
22. import org.springframework.web.bind.annotation.PathVariable;
23. import org.springframework.web.bind.annotation.PostMapping;
24. import org.springframework.web.bind.annotation.PutMapping;
25. import org.springframework.web.bind.annotation.RequestBody;
26. import org.springframework.web.bind.annotation.RequestMapping;
27. import org.springframework.web.bind.annotation.RequestParam;
28. import org.springframework.web.bind.annotation.RestController;
29. import org.springframework.web.util.UriComponentsBuilder;
30.
31. import io.swagger.annotations.Api;
32. import io.swagger.annotations.ApiOperation;
33. import io.swagger.annotations.ApiResponse;
34. import io.swagger.annotations.ApiResponses;
35.
36. @RestController
37. @RequestMapping("/rest/customer/api")
38. @Api(value = "Customer Rest Controller: contains all operations for managing customers")
39. public class CustomerRestController {
40.
41.     public static final Logger LOGGER =
 LoggerFactory.getLogger(CustomerRestController.class);
42.
43.     @Autowired
44.     private CustomerServiceImpl customerService;
45.
46.     @Autowired
47.     private JavaMailSender javaMailSender;
48.
49.     /**
50.      * Ajoute un nouveau client dans la base de données H2. Si le client existe déjà, on
 retourne un code indiquant que la création n'a pas abouti.
51.      * @param customerDTORequest
52.      * @return
53.      */
54.     @PostMapping("/addCustomer")
55.     @ApiOperation(value = "Add a new Customer in the Library", response = CustomerDTO.class)
56.     @ApiResponses(value = { @ApiResponse(code = 409, message = "Conflict: the customer
 already exist"),
57.             @ApiResponse(code = 201, message = "Created: the customer is successfully
 inserted"),
58.             @ApiResponse(code = 304, message = "Not Modified: the customer is unsuccessfully
 inserted") })
```

```
59.     public ResponseEntity<CustomerDTO> createNewCustomer(@RequestBody CustomerDTO
 customerDTORequest) {
60.         //, UriComponentsBuilder uriComponentBuilder
61.         Customer existingCustomer =
 customerService.findCustomerByEmail(customerDTORequest.getEmail());
62.         if (existingCustomer != null) {
63.             return new ResponseEntity<CustomerDTO>(HttpStatus.CONFLICT);
64.         }
65.         Customer customerRequest = mapCustomerDTOToCustomer(customerDTORequest);
66.         customerRequest.setCreationDate(LocalDate.now());
67.         Customer customerResponse = customerService.saveCustomer(customerRequest);
68.         if (customerResponse != null) {
69.             CustomerDTO customerDTO = mapCustomerToCustomerDTO(customerResponse);
70.             return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.CREATED);
71.         }
72.         return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_MODIFIED);
73.     }
74.
75.     /**
76.      * Met à jour les données d'un client dans la base de données H2. Si le client n'est pas
 retrouvé, on retourne un code indiquant que la mise à jour n'a pas abouti.
77.      * @param customerDTORequest
78.      * @return
79.      */
80.     @PutMapping("/updateCustomer")
81.     @ApiOperation(value = "Update/Modify an existing customer in the Library", response =
 CustomerDTO.class)
82.     @ApiResponses(value = { @ApiResponse(code = 404, message = "Not Found : the customer does
 not exist"),
83.             @ApiResponse(code = 200, message = "Ok: the customer is successfully updated"),
84.             @ApiResponse(code = 304, message = "Not Modified: the customer is unsuccessfully
 updated") })
85.     public ResponseEntity<CustomerDTO> updateCustomer(@RequestBody CustomerDTO
 customerDTORequest) {
86.         //, UriComponentsBuilder uriComponentBuilder
87.         if (!customerService.checkIfIdexists(customerDTORequest.getId())) {
88.             return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_FOUND);
89.         }
90.         Customer customerRequest = mapCustomerDTOToCustomer(customerDTORequest);
91.         Customer customerResponse = customerService.updateCustomer(customerRequest);
92.         if (customerResponse != null) {
93.             CustomerDTO customerDTO = mapCustomerToCustomerDTO(customerResponse);
94.             return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.OK);
95.         }
96.         return new ResponseEntity<CustomerDTO>(HttpStatus.NOT_MODIFIED);
97.     }
98.
99.     /**
100.      * Supprime un client dans la base de données H2. Si le client n'est pas retrouvé, on
 retourne le Statut HTTP NO_CONTENT.
101.      * @param customerId
102.      * @return
103.      */
104.     @DeleteMapping("/deleteCustomer/{customerId}")
105.     @ApiOperation(value = "Delete a customer in the Library, if the customer does not exist,
 nothing is done", response = String.class)
106.     @ApiResponse(code = 204, message = "No Content: customer sucessfully deleted")
107.     public ResponseEntity<String> deleteCustomer(@PathVariable Integer customerId) {
108.         customerService.deleteCustomer(customerId);
109.         return new ResponseEntity<String>(HttpStatus.NO_CONTENT);
110.     }
111.
112.     @GetMapping("/paginatedSearch")
113.     @ApiOperation(value="List customers of the Library in a paginated way", response =
 List.class)
114.     @ApiResponses(value = {
115.             @ApiResponse(code = 200, message = "Ok: successfully listed"),
116.             @ApiResponse(code = 204, message = "No Content: no result founded"),
117.     })
118.     public ResponseEntity<List<CustomerDTO>> searchCustomers(@RequestParam("beginPage") int
 beginPage,
119.             @RequestParam("endPage") int endPage) {
```

```
120.          //, UriComponentsBuilder uriComponentBuilder
121.          Page<Customer> customers = customerService.getPaginatedCustomersList(beginPage,
 endPage);
122.          if (customers != null) {
123.              List<CustomerDTO> customerDTOs = customers.stream().map(customer -> {
124.                  return mapCustomerToCustomerDTO(customer);
125.              }).collect(Collectors.toList());
126.              return new ResponseEntity<List<CustomerDTO>>(customerDTOs, HttpStatus.OK);
127.          }
128.          return new ResponseEntity<List<CustomerDTO>>(HttpStatus.NO_CONTENT);
129.      }
130.
131.      /**
132.       * Retourne le client ayant l'adresse email passée en paramètre.
133.       * @param email
134.       * @return
135.       */
136.      @GetMapping("/searchByEmail")
137.      @ApiOperation(value="Search a customer in the Library by its email", response =
 CustomerDTO.class)
138.      @ApiResponses(value = {
139.              @ApiResponse(code = 200, message = "Ok: successfull research"),
140.              @ApiResponse(code = 204, message = "No Content: no result founded"),
141.      })
142.      public ResponseEntity<CustomerDTO> searchCustomerByEmail(@RequestParam("email") String
 email) {
143.          //, UriComponentsBuilder uriComponentBuilder
144.          Customer customer = customerService.findCustomerByEmail(email);
145.          if (customer != null) {
146.              CustomerDTO customerDTO = mapCustomerToCustomerDTO(customer);
147.              return new ResponseEntity<CustomerDTO>(customerDTO, HttpStatus.OK);
148.          }
149.          return new ResponseEntity<CustomerDTO>(HttpStatus.NO_CONTENT);
150.      }
151.
152.      /**
153.       * Retourne la liste des clients ayant le nom passé en paramètre.
154.       * @param lastName
155.       * @return
156.       */
157.      @GetMapping("/searchByLastName")
158.      @ApiOperation(value="Search a customer in the Library by its Last name", response =
 List.class)
159.      @ApiResponses(value = {
160.              @ApiResponse(code = 200, message = "Ok: successfull research"),
161.              @ApiResponse(code = 204, message = "No Content: no result founded"),
162.      })
163.      public ResponseEntity<List<CustomerDTO>> searchBookByLastName(@RequestParam("lastName")
 String lastName) {
164.          //,    UriComponentsBuilder uriComponentBuilder
165.          List<Customer> customers = customerService.findCustomerByLastName(lastName);
166.          if (customers != null && !CollectionUtils.isEmpty(customers)) {
167.              List<CustomerDTO> customerDTOs = customers.stream().map(customer -> {
168.                  return mapCustomerToCustomerDTO(customer);
169.              }).collect(Collectors.toList());
170.              return new ResponseEntity<List<CustomerDTO>>(customerDTOs, HttpStatus.OK);
171.          }
172.          return new ResponseEntity<List<CustomerDTO>>(HttpStatus.NO_CONTENT);
173.      }
174.
175.      /**
176.       * Envoie un mail à un client. L'objet MailDTO contient l'identifiant et l'email du
 client concerné, l'objet du mail et le contenu du message.
177.       * @param loanMailDto
178.       * @param uriComponentBuilder
179.       * @return
180.       */
181.      @PutMapping("/sendEmailToCustomer")
182.      @ApiOperation(value="Send an email to customer of the Library", response = String.class)
183.      @ApiResponses(value = {
184.              @ApiResponse(code = 200, message = "Ok: Email successfully sent"),
```

```
185.             @ApiResponse(code = 404, message = "Not Found: no customer found, or wrong
 email"),
186.             @ApiResponse(code = 403, message = "Forbidden: Email cannot be sent")
187.     })
188.     public ResponseEntity<Boolean> sendMailToCustomer(@RequestBody MailDTO loanMailDto,
 UriComponentsBuilder uriComponentBuilder) {
189.
190.         Customer customer = customerService.findCustomerById(loanMailDto.getCustomerId());
191.         if (customer == null) {
192.             String errorMessage = "The selected Customer for sending email is not found in
 the database";
193.             LOGGER.info(errorMessage);
194.             return new ResponseEntity<Boolean>(false, HttpStatus.NOT_FOUND);
195.         } else if (customer != null && StringUtils.isEmpty(customer.getEmail())) {
196.             String errorMessage = "No existing email for the selected Customer for sending
 email to";
197.             LOGGER.info(errorMessage);
198.             return new ResponseEntity<Boolean>(false, HttpStatus.NOT_FOUND);
199.         }
200.
201.         SimpleMailMessage mail = new SimpleMailMessage();
202.         mail.setFrom(loanMailDto.MAIL_FROM);
203.         mail.setTo(customer.getEmail());
204.         mail.setSentDate(new Date());
205.         mail.setSubject(loanMailDto.getEmailSubject());
206.         mail.setText(loanMailDto.getEmailContent());
207.
208.         try {
209.             javaMailSender.send(mail);
210.         } catch (MailException e) {
211.             return new ResponseEntity<Boolean>(false, HttpStatus.FORBIDDEN);
212.         }
213.         return new ResponseEntity<Boolean>(true, HttpStatus.OK);
214.     }
215.
216.     /**
217.      * Transforme un entity Customer en un POJO CustomerDTO
218.      *
219.      * @param customer
220.      * @return
221.      */
222.     private CustomerDTO mapCustomerToCustomerDTO(Customer customer) {
223.         ModelMapper mapper = new ModelMapper();
224.         CustomerDTO customerDTO = mapper.map(customer, CustomerDTO.class);
225.         return customerDTO;
226.     }
227.
228.     /**
229.      * Transforme un POJO CustomerDTO en en entity Customer
230.      *
231.      * @param customerDTO
232.      * @return
233.      */
234.     private Customer mapCustomerDTOToCustomer(CustomerDTO customerDTO) {
235.         ModelMapper mapper = new ModelMapper();
236.         Customer customer = mapper.map(customerDTO, Customer.class);
237.         return customer;
238.     }
239. }
```

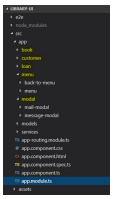
Dans la page suivante, nous vous présenterons l'implémentation du front-end de l'application *Library*. Ce front-end autrement appelé *Client* est une application *Angular* dans laquelle nous développerons des services qui consommeront les web services exposés dans *Library*.

XII - Objectifs du front-end

L'application *Library* développée et présentée dans la page précédente, dispose d'une API REST composée de web services concourant à répondre aux besoins exprimés dans les **Users Stories**. Dans ce contexte, elle ne peut gagner toute son utilité que s'il existe une ou plusieurs application(s) cliente(s) qui consomme(nt) les web services exposés.

Dans cette partie de notre article, notre objectif est de développer un client web, baptisé ***Library-ui***, qui offrira une vue IHM (Interface Homme Machine) pour la gestion de la bibliothèque. Ce client web sera composé de pages web pour l'administration des livres, des clients et des prêts. Il disposera aussi d'un menu général permettant de naviguer vers les différentes pages. Chacune de ces pages comportera un bouton *back* permettant de retourner sur le menu général. ***Library-ui*** devra être un site web dynamique qui consomme les API REST exposées en back-end dans le projet *Library*, afin de permettre à l'administrateur de l'application de réaliser tous ses besoins.

Pour ce faire, nous avons sélectionné le framework ***Angular*** qui nous permettra d'atteindre cette spécification. Sa version au moment où nous développions cette application était la 7. Au terme de cette partie, nous allons construire une application purement cliente dont la figure ci-dessous illustre les différents composants :



Application Library-ui

Les différents composants coloriés en jaune sont les suivants :

- le composant *Book*, qui proposera une page web pour la création, mise à jour, lecture et suppression de livres ;
- le composant *Customer*, qui proposera une page web pour la création, mise à jour, lecture et suppression de clients ;
- Le composant *Loan*, qui proposera une page web pour la création, mise à jour, lecture et suppression de prêts ;
- le composant *menu*, pour la gestion du menu général ;
- le composant *modal*, incorporé dans le composant Loan, qui proposera une fenêtre modale permettant d'envoyer des mails.

XIII - Mise en place du projet Angular Library-ui

XIII-A - Préambule et notions de base

Angular est un framework basé sur le langage *TypeScript* (une surcouche du *JavaScript*). Sa philosophie, comme celle de nombreux frameworks concurrents (*Vue.js*, *React.js*, etc.), peut se résumer par la phrase suivante : ***proposer un nouveau paradigme de construction de sites web qui améliore l'expérience utilisateur***. Plus simplement, cela veut dire, l'optimisation du temps de réponse et donc de la performance des pages web visitées par une tierce personne lorsqu'elle navigue sur un site web. Pour atteindre cet objectif, Angular s'appuie sur deux politiques architecturales majeures.

- La notion de **Single Page Application (SPA)** : il s'agit d'une approche qui consiste à considérer les différentes pages web d'un site comme des composants devant tous être incorporés dans une seule et unique page principale. Seule cette page principale est reconnue et gérée par le navigateur. Par des mécanismes très poussés que propose le framework, il sera alors possible de jouer sur l'affichage ou pas de

différents composants dans cette page principale afin de donner l'impression à l'utilisateur qu'il navigue sur des pages web différentes.
- **La maximisation des traitements côté client** : cette approche permet de créer des sites web dynamiques où la grande majorité des traitements derrière tout composant sont exécutés côté client, c'est-à-dire votre navigateur web (Chrome, Mozilla, IE, etc). Seuls les traitements nécessitant la consommation d'une API REST feront appel au serveur en back-end.

La philosophie ci-dessus expliquée du framework Angular se distingue donc de celle des technologies comme JSP/JSTL, PHP, etc., où le serveur héberge plusieurs pages web et se charge de les servir à votre navigateur à chaque requête. Cette approche provoque ainsi une actualisation par rechargement des pages sur le navigateur à la moindre action utilisateur. Ce qui provoque une perte de temps et une navigation dégradée. Or avec la politique SPA, les requêtes du client vers le serveur n'impliquent pas un rechargement de la page principale hébergée sur le serveur. Celle-ci n'est en effet chargée qu'une seule fois lors du démarrage de l'application cliente.

Angular fait partie des frameworks basés sur la plateforme ***node.js***. *Node.js* est un outil offrant, entre autres, la possibilité de compiler et d'exécuter du code Angular en phase de développement sur un poste de travail. Il s'installe aisément sur votre poste de travail. *Angular* s'appuie aussi sur le repository ***npm*** (désormais intégré dans *node.js*) pour vous permettre d'installer des modules qui lui permettent de vous offrir de nombreuses fonctionnalités exploitables dans votre projet Angular.

Pour l'installation de *node.js/npm*, nous vous demandons de suivre le lien suivant **https://nodejs.org/fr/** pour télécharger l'installateur et l'installer sur votre machine en suivant les indications.

Enfin, pour développer une application Angular, il faut un IDE (Integrated Developpement Environnement) approprié. Sur Eclipse, il existe le plugin **Angular Eclipse** d'Angelo Zerr, à installer via la MarketPlace. On peut aussi utiliser l'IDE gratuit **Visual Studio Code** qui intègre déjà tout le nécessaire pour développer des projets Angular. Enfin, il existe aussi l'IDE payant **Angular IDE**. Nous faisons le choix pour cet article d'utiliser Visual Studio Code.

Pour terminer cette section, si vous avez besoin de plus d'informations sur la notion de *Single Page Application*, vous pouvez lire les contenus suivants : **Arolla**, **Octo**.

XIII-B - Création du projet Library-ui

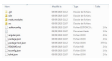
Maintenant que vous avez installé *node.js/npm* et *Visual Studio Code* pour la plateforme de développement, nous sommes presque prêts pour la création du projet Angular ***Library-ui***. Mais avant cette opération, il est nécessaire d'installer ***Angular CLI***. *Angular CLI* est un outil permettant de créer et de gérer aisément le cycle de vie d'un projet de type Angular. Toute proportion gardée, on peut voir *Angular CLI* comme le *maven* des projets Java. Au moment de développer ce projet, *Angular CLI* était à la version 7.3.3 correspondant donc à la version 7 du framework Angular. Son installation se réduit à l'instruction suivante dans un terminal et dans n'importe quel répertoire de poste de travail :

```
npm install -g @angular/cli@7.3.3
```

Maintenant nous pouvons donc créer notre projet à l'aide de la commande :

```
ng new Library-ui
```

Le projet imagé sur la figure ci-dessous est donc créé et il ne reste plus qu'à l'ouvrir avec notre IDE pour le compléter. Le répertoire */src* représente celui dans lequel nous ajouterons nos *codes sources*.



Nouveau projet Library-ui

XIII-C - Configuration du projet

Lors de la phase de développement, le framework Angular utilise le port 4200 sur votre localhost pour visualiser en temps réel les IHM développées. L'URL d'accès est donc la suivante : ***http://localhost:4200***.
Nous voulons par ailleurs que notre application soit accessible via le lien ***http://localhost:4200/library-ui*** où *library-ui* est le point d'entrée de l'application permettant d'afficher le menu. configurer. Enfin, et comme nous l'avons expliqué, notre application *Library-ui* communiquera avec le back-end à l'aide d'appel REST. Il est donc important que nous vous présentions comment cela est mis en place.

1. Modification du fichier Package.json

Dans le répertoire d'installation du projet *Library-ui* (voir capture ci-dessus), ouvrez le fichier *Package.json* et modifiez la ligne coloriée en jaune pour ajouter l'instruction suivante : ***--base-href /library-ui --proxy-config proxy.conf.json***.

Config Package.json

- La commande ***--base-href /library-ui***, permet de marquer l'accessibilité de la page web principale sur le suffixe */library-ui*. Exemple : la saisie de l'URL *http://localhost:4200/**library-ui*** permettra d'afficher le menu principal.
- La commande ***--proxy-config proxy-conf.json***, permet d'indiquer l'adresse des différents serveurs côté back-end auxquels l'application cliente Angular peut être amenée à appeler. En d'autres termes, c'est dans ce fichier que l'on configure les URL des serveurs qui hébergent les API REST auxquels *Library-ui* va faire appel.

2. Création du fichier proxy-conf.json

Dans le répertoire d'installation du projet *Library-ui* (voir capture ci-dessus) et au même niveau que le fichier *package.json*, créez le fichier de nom *proxy-conf.json* et ajoutez le contenu présenté sur la capture. Pour aller plus loin sur la configuration, vous pouvez consulter le lien **suivant**. Notez que le contenu de ce fichier indique que le serveur back-end répond sur l'URL *http://localhost:8082* en n'admettant uniquement les requêtes http contenant le préfixe **/library**. Cette configuration s'adapte donc aux URI de nos API REST exposés dans le projet Library côté back-end dans les contrôleurs REST.

Config proxy-conf.json

Une fois les deux premières configurations ci-dessus effectuées, ouvrez le fichier *angular.json* et dans la rubrique *serve*, ajoutez la commande "***proxyConfig": "src/proxy.conf.json"***. Il suffit de se positionner à la racine du répertoire du projet *Library-ui* et d'exécuter la commande ***npm start*** sur un terminal pour démarrer l'exécution de cette application.

npm start

3. Dossier ressources

Notez, que le projet *Library-ui* contient un dossier nommé *assets* obtenu lors de la création du projet à l'aide de la commande *ng new*… Ce dossier *assets* est destiné à contenir tous les fichiers ressources de l'application à l'instar des images. C'est ce que nous avons fait en ajoutant l'image que vous visualiserez sur la page de menu de l'application.

4. Configuration du spinner

Nous avons rajouté un spinner dans notre application *Library-ui*. Cela nécessite d'installer un nouveau module nommé *ngx-spinner*. Pour rappel, un Spinner est un composant HTML graphique et dynamique qui permet de signaler l'indisponibilité totale d'une page web quand il est en train de charger les éléments constituants son contenu. Pour en savoir plus sur son installation et sa configuration, suivre le lien **suivant**.

5. Configuration du DatePicker

Sur nos pages HTML, nous aurons besoin d'utiliser des calendriers afin de compléter des champs de saisie avec des dates (ex. : date de sortie d'un livre, date de début d'un prêt, etc.). Pour ce faire, Angular propose un module offrant des outils permettant de gérer aisément l'affichage et la manipulation d'un calendrier dans un élément HTML de type *input*. Cependant, ce module n'est pas présent dans la liste des modules de base obtenus lors de l'installation de l'Angular CLI. Nous devons donc installer le module ***Angular Material***. Vous pouvez suivre le lien **suivant**, pour comprendre comment nous l'avons intégré dans notre application.

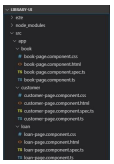
XIV - Pages web et services HttpClient

Dans cette partie, nous présentons succinctement les trois différentes pages web que nous avons développées pour gérer respectivement les livres, clients et prêts. Nous présentons aussi les services Angular associés permettant de consommer les API REST exposées côté back-end. Comme nous vous l'avons expliqué dans la section précédente, le terme *page web* est un abus de langage d'un point de vue Angular. En effet, ce framework crée et gère des *composants HTML* qu'il intègre dans une seule et unique page principale (l'*index.html* située à la racine du projet). Ainsi donc, dans le sens pur du terme, Angular permet de développer des composants.

Pour créer un composant Angular, allez dans le répertoire *library-ui/src/app* et saissez la commande suivante :

```
ng g component <nom-composant>
```

Cette commande nous permettra de créer respectivement les composants *Book-page*, *Customer-page*, *Loan-page*, etc. Leurs dossiers respectifs seront générés et contiendront les fichiers d'extension : *.html*, *.css*, *.ts* et *.spec.ts*. Exemple :



Composant Angular

- le fichier d'extension *.html*, est un fichier template qui contient tous les éléments HTML nécessaires à l'affichage du composant ;
- le fichier d'extension *.css*, contient les instructions de style pour le *.html* ;

- le fichier d'extension *.ts*, contient une classe dans laquelle nous allons développer nos fonctions *TypeScript* afin de répondre au besoin métier que doit gérer le composant. Ce fichier *.ts* contient donc toutes les variables utilisées par le *.html* pour afficher dynamiquement des données. Par ailleurs, le *.html* transmettra des informations au *.ts* à travers l'émission d'évènements basés sur des fonctions développées dans ce dernier. Enfin, notez que ce sont les fonctions développées dans le *.ts* qui consommeront réellement les API REST exposées côté back-end.
- le fichier d'extension *.spec.ts*, correspond au fichier de tests unitaires des fonctions développées dans le *.ts*

Pour des raisons de clarté architecturale, nous avons créé, pour chaque composant *Book-page*, *Customer-page* et *Loan-page*, une classe de services regroupant les différents appels REST nécessaires à la page. Ainsi donc, le fichier *.ts* de chaque composant n'aura plus qu'à appeler cette classe pour consommer les différents web service. Pour implémenter nos classes de services, nous avons créer un dossier *services* dans le répertoire *library-ui/src/app* et y avons exécuté la commande suivante :

```
ng g service <nom-service>
```

Pour la suite de cette partie, nous vous exposons le rendu de chacun des composants ainsi que la classe de services associée. Nous laissons le soin au lecteur de s'approprier le code source développé dans chacun des composants. Les *sources* de l'application sont disponibles à la fin de cet article.

XIV-A - Composant Book et ses services d'appel REST

Dans cette section, nous vous présentons la page de gestion des livres que nous avons développée. Vous pourrez retrouver son code source dans le composant *src/app/book* du projet *Library-ui* :

Composant Book-page

Cette page a pour besoin de consommer les différents web services exposés côté back-end dans le contrôleur *BookRestController*. La classe de service baptisée **BookService** définie dans le fichier */src/app/services/book.service.ts*, injecte via son constructeur, l'objet *HttpClient* fourni par Angular afin de pouvoir consommer des web services :

```
import { Injectable } from '@angular/core';
import { Observable } from "rxjs";
import { Category } from "src/app/models/category";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Book } from "src/app/models/book";

@Injectable({
  providedIn: 'root'
})
export class BookService {

    constructor(private http: HttpClient) { }

    /**
     * Get all book's categories as reference data from Backend server.
     */
     loadCategories(): Observable<Category[]>{
         let headers = new HttpHeaders();
         headers.append('content-type', 'application/json');
         headers.append('accept', 'application/json');
         return this.http.get<Category[]>('/library/rest/category/api/allCategories', {headers:
 headers});
     }

    /**
```

```
    * Save a new Book object in the Backend server data base.
    * @param book
    */
    saveBook(book: Book): Observable<Book>{
        return this.http.post<Book>('/library/rest/book/api/addBook', book);
    }

    /**
     * Update an existing Book object in the Backend server data base.
     * @param book
     */
     updateBook(book: Book): Observable<Book>{
         return this.http.put<Book>('/library/rest/book/api/updateBook', book);
     }

     /**
      * Delete an existing Book object in the Backend server data base.
      * @param book
      */
      deleteBook(book: Book): Observable<string>{
          return this.http.delete<string>('/library/rest/book/api/deleteBook/'+book.id);
      }

    /**
     * Search books by isbn
     * @param isbn
     */
    searchBookByIsbn(isbn: string): Observable<Book>{
        return  this.http.get<Book>('/library/rest/book/api/searchByIsbn?isbn='+isbn);
    }

   /**
    * Search books by title
    * @param title
    */
    searchBookByTitle(title: string): Observable<Book[]>{
            return this.http.get<Book[]>('/library/rest/book/api/searchByTitle?title='+title);
    }
}
```

HttPClient fournit entre autres, plusieurs méthodes permettant de faire des appels REST : *get()*, *delete()*, *put()*, *post()*. Chacune de ces méthodes prend en entrée l'URI du web service concerné et retourne un objet *Observable* encapsulant l'objet de retour du web service. La classe *Observable* fournissant de son côté des méthodes permettant d'exploiter l'objet de retour. C'est ce qui est fait dans le fichier ***book-page.component.ts***.

XIV-B - Composant Customer et ses services d'appel REST

Dans cette section, nous vous présentons la page de gestion des clients que nous avons développée. De même, vous pourrez retrouver son code source dans le composant *src/app/customer* du projet *Library-ui* :

Composant Customer-page

Cette page a pour besoin de consommer les différents web services exposés côté back-end dans le contrôleur *CustomerRestController*. De même la classe de services baptisée **CustomerService** et définie dans le fichier */src/app/services/customer.service.ts*, expose les différents appels :

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Customer } from '../models/customer';
```

```
@Injectable({
  providedIn: 'root'
})
export class CustomerService {

  constructor(private http: HttpClient) { }

    /**
     * Save a new Customer object in the Backend server data base.
     * @param book
     */
     saveCustomer(customer: Customer): Observable<Customer>{
         return this.http.post<Customer>('/library/rest/customer/api/addCustomer', customer);
     }

     /**
      * Update an existing Customer object in the Backend server data base.
      * @param customer
      */
      updateCustomer(customer: Customer): Observable<Customer>{
          return this.http.put<Customer>('/library/rest/customer/api/updateCustomer', customer);
      }

      /**
       * Delete an existing Customer object in the Backend server data base.
       * @param customer
       */
       deleteCustomer(customer: Customer): Observable<string>{
           return this.http.delete<string>('/library/rest/customer/api/
deleteCustomer/'+customer.id);
       }

     /**
      * Search customer by email
      * @param email
      */
     searchCustomerByEmail(email: string): Observable<Customer>{
         return  this.http.get<Customer>('/library/rest/customer/api/searchByEmail?
email='+email);
     }

    /**
     * Search books by pagination
     * @param beginPage
     * @param endPage,
     */
     searchCustomerByLastName(lastName: string): Observable<Customer[]>{
             return this.http.get<Customer[]>('/library/rest/customer/api/searchByLastName?
lastName='+lastName);
     }
}
```

XIV-C - Composant Loan et ses services d'appel REST

Dans cette section, nous vous présentons la page de gestion des prêts que nous avons développée. Le code source qui permet la construction de ce composant est situé dans le dossier *src/app/loan* du projet *Library-ui* :

Composant Loan-page

Cette page a pour besoin de consommer les différents web services exposés côté back-end dans le contrôleur *LoanRestController*. Sa classe de services baptisée **LoanService** est définie dans le fichier */src/app/services/loan.service.ts* est la suivante :

```
import { Injectable } from '@angular/core';
import { SimpleLoan } from '../models/simple-loan';
import { Loan } from '../models/loan';
import { Observable } from 'rxjs';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Mail } from '../models/mail';

@Injectable({
  providedIn: 'root'
})
export class LoanService {

  constructor(private http: HttpClient) { }

    /**
     * Save a new simpleLoan object in the Backend server data base.
     * @param book
     */
    saveLoan(simpleLoan: SimpleLoan): Observable<Loan>{
        return this.http.post<Loan>('/library/rest/loan/api/addLoan', simpleLoan);
     }

      /**
       * Close an existing loan object in the Backend server data base.
       * @param loan
       */
      closeLoan(simpleLoan: SimpleLoan): Observable<Boolean>{
          return this.http.post<Boolean>('/library/rest/loan/api/closeLoan/', simpleLoan);
       }

     /**
      * Search Loans by email
      * @param email
      */
     searchLoansByEmail(email: string): Observable<Loan[]>{
         return  this.http.get<Loan[]>('/library/rest/loan/api/customerLoans?email='+email);
     }

     /**
      * Search Loans by maximum date
      * @param maxDate
      */
     searchLoansByMaximumDate(maxDate: Date): Observable<Loan[]>{
       let month : string =
 maxDate.getMonth() < 10 ? '0'+(maxDate.getMonth()+1): ''+(maxDate.getMonth()+1);
       let dayOfMonth : string =
 maxDate.getDate() < 10 ? '0'+maxDate.getDate(): ''+maxDate.getDate();
       let maxDateStr : string = maxDate.getFullYear() + '-' + month + '-' + dayOfMonth;
      return  this.http.get<Loan[]>('/library/rest/loan/api/maxEndDate?date='+maxDateStr);
  }

    /**
     * Send an email to a customer
     * @param mail
     */
    sendEmail(mail: Mail): Observable<boolean>{
      //let headers = new HttpHeaders();
      //headers.append('responseType', 'arraybuffer'); , {headers: headers}
      return this.http.put<boolean>('/library/rest/customer/api/sendEmailToCustomer', mail);
    }
}
```

XIV-D - Le menu principal

Nous avons développé une page principale servant de menu et permettant de naviguer d'une page à l'autre. Son code source se trouve dans le dossier */src/app/menu/menu*. Par contre ce composant ne fait pas d'appel REST. Il n'y a donc pas de classe de service pour cette dernière.

Composant Menu-page

Dans la page suivante, nous allons vous présenter comment faire un déploiement de notre application (front-end et back-end) sur un serveur Tomcat. Nous vous présenterons aussi une vidéo dans laquelle nous présentons l'application en *live*.

XV - Déploiement de l'application dans Tomcat

Cette partie de l'article va se consacrer à la présentation du déploiement, sur un système d'exploitation Windows, de notre application Library (front-end et back-end) dans un serveur d'application, en l'occurrence Tomcat. Le déploiement que nous présenterons ici suivra un processus manuel, mais représente très exactement ce que des outils d'intégration comme *Jenkins* ou encore *Continium*, automatisent. Nous vous proposerons un prochain article dédié à l'intégration continue et le déploiement automatisé d'application.

XV-A - Préparation des livrables du front-end

Étape 1 : modifier le fichier *package.json* du projet *Library-ui* pour ajouter l'instruction **--prod --base-href /library-ui/**.

Configuration build prod Library-ui

- **--prod** : cette option permet de compiler le projet en mode production. Les fichiers qui seront générés seront minifiés de façon à optimiser le poids en termes d'octets de ces derniers.
- **--base-href /library-ui/** : cette option permet de spécifier l'URL de base des pages web du projet.

Étape 2 : : modifier le fichier *app-routing.module.ts* du projet *Library-ui* pour configurer l'ancre afin d'empêcher de tomber sur la page d'erreur 404 lorsqu'on rafraîchit une page de l'application dans le navigateur.

Cette étape n'est pas obligatoire, mais son absence se fera remarquer une fois que vous aurez livré votre application dans Tomcat. En effet, si vous la sautez, vous recevrez forcément des rapports de nombreux utilisateurs de votre application qui se plaindraient de tomber sur la page d'erreur 404 lorsqu'ils refraîchissent une page web de l'application ; ce qui est pour le moins très désagréable. Pour vous décrire un peu le scénario, supposez que vous êtes sur le menu principal de l'application (cf. section **XIV-D**), vous cliquez par exemple sur le bouton *Book Management* qui vous emmene sur la page de gestion des livres (cf. *http://localhost:8082/library-ui/book-page*). Une fois sur cette page, si vous la rafraîchissez via le bouton *Actualiser* de votre navigateur, vous verez apparaître l'erreur suivante :

Page d'erreur 404 après rafraîchissement

Pour résoudre ce problème, on va devoir ajouter l'instruction ***useHash : true*** dans le fichier *app-routing.module.ts* comme le montre la figure ci-dessous.

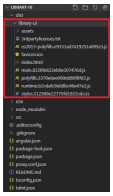
Configuration de l'ancre

Pourquoi faisons nous cette configuration ? Simplement pour forcer le serveur à toujours suffixer l'url de base de l'application par **#/**. On obtiendrait alors *http://localhost:8082/library-ui/#/*. Souvenez vous que dans le principe du SPA, nous avions dit que tous les composants angular étaient en fait incluses dans la seule et unique page *index.html* (qui est servie par défaut sur l'url *http://localhost:8082/library-ui/*) et qu'il y avait un jeu d'affichage de ces composants pour vous donnez l'impression de naviguer sur différentes pages web. Pour forcer le serveur à rediriger sans erreur depuis *index.html* vers un composant contenu dans cette page, on utilise donc l'ancre **#/** imposer par la commande *useHash : true*. Ainsi donc pour notre exemple, en rafraîchissant indéfiniment la page *http://localhost:8082/library-ui/#/book-page*, le problème 404 est résolu.

Étape 3 : ouvrir un terminal et se positionner à la racine du projet et saisir l'instruction suivante :

```
npm run-script build
```

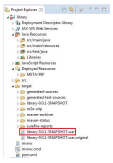
Si tout se passe bien, un dossier *dist/**library-ui*** est généré dans le projet Library-ui. C'est ce dossier *library-ui* que nous utiliserons pour le déploiement dans le serveur Tomcat.



Livrables Library-ui

XV-B - Préparation des livrables du back-end

Étape 1 : packager le projet Library à l'aide de la commande maven ***mvn clean package***.



Livrables du projet Library

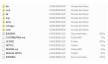
Un fichier ***library-0.0.1-SNAPSHOT.war*** est généré dans le dossier target. Ce fichier représente le principal livrable côté back-end.

Étape 2 : renommer le fichier ***library-0.0.1-SNAPSHOT.war*** avec le nom suivant ***library.war***. C'est ce fichier *library.war* que nous utiliserons pour le déploiement dans le serveur Tomcat.

XV-C - Déploiement dans le serveur Tomcat

Étape 1 : télécharger la version la plus à jour de Tomcat et bien entendu celui correspondant au système d'exploitation Windows. Il s'agira nécessairement d'un fichier .zip. L'adresse de téléchargement est la **suivante**. L'installation

est simple, il suffit de déposer le fichier zip à un endroit sur votre poste et de le dézipper. Vous obtenez un dossier comme celui-ci :

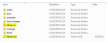


Dossier d'installation

Étape 2 : modifier le fichier conf/server.xml pour changer le port d'écoute du serveur Tomcat. En effet nous souhaitons que notre Back-end réponde sur le port 8082.

Configuration port d'écoute

Étape 3 : copier/coller les livrables dans le dossier *webapps* de Tomcat. Les livrables sont le dossier *library-ui* contenant les artefacts du projet front-end obtenu à la **section XV-A** et *library.war* obtenu à la **section XV-B**.



Livrables webapps

Étape 4 : démarrer Tomcat afin qu'il effectue le déploiement de nos deux applications aux fins de leur exploitation effective. Pour ce faire, allez dans le répertoire bin de Tomcat, ouvrez-y un terminal et saisissez la commande ***startup***.

Si tout se passe bien, on verra un nouveau terminal qui s'ouvre et qui se présentera comme sur les figures ci-dessous.

Déploiement Library

Déploiement Library

Le déploiement de notre application est terminé. Il ne nous reste plus qu'à naviguer sur notre site web pour effectuer des actions qu'il propose. C'est ce que nous allons vous montrer dans la vidéo de la section suivante.

XVI - Présentation vidéo

Cliquer sur ce lien pour lancer l'animation

XVII - Accéder au code source de l'application

Le code source de l'application Library côté back-end se trouve ici : **https://gitlab.com/gkemayo/library**.

Le code source de l'application Library-ui côté front-end se trouve ici : **https://gitlab.com/gkemayo/library-ui**.

XVIII - Conclusion, perspectives et remerciements

XVIII-A - Conclusion

Au terme de cet article, nous vous avons présenté comment il est possible de construire une application Java web en s'appuyant fondamentalement sur deux frameworks majeurs que sont Spring Boot et Angular. *Spring Boot* nous ayant permis de construire la partie back-end (serveur) et *Angular* pour la partie front-end (client). Nous avons pris comme exemple de projet, le développement d'une application de gestion de livres pour une bibliothèque. Même si nous ne sommes pas entrés en profondeur sur les aspects HTML/CSS et donc de design des pages web que nous avons développées, notre objectif était de vous proposer une application complète qui ne s'arrête pas à la présentation de deux ou trois concepts techniques. Vous pourrez consulter le code source de l'application pour aller plus loin. Enfin, nous vous avons montré comment déployer notre application en contexte de production dans un serveur Tomcat. L'article s'achève par une présentation vidéo présentant la navigation sur les différentes pages web et les actions qu'elles proposent.

XVIII-B - Perspectives

Dans cet article nous n'avons pas abordé les notions suivantes :

- connexion et sécurisation de l'application ;
- gestion des exceptions dans nos services et contrôleurs Rest ;
- gestion du CORS permettant à deux applications situées sur deux serveurs différents de communiquer en HTTP ;
- tests unitaires.

Ces différents concepts sont très importants à mettre en place dans une application afin de la rendre sécurisée, robuste, maintenable et non régressive suite à une éventuelle évolution. Ce n'était pas l'axe de cet article et nous vous laissons donc le soin de les mettre en œuvre dans l'application Library.

Voici quelques références utiles autour des notions abordées dans cet article :

- **Spring Boot et Restemplate : https://bnguimgo.developpez.com/tutoriels/spring/services-rest-avec-springboot-et-spring-resttemplate/**
- **Spring Boot guide : https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/**
- **Spring Boot initialzr : https://start.spring.io/**
- **Spring Data JPA : https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html**
- **Appendix A. Common application properties**
- **Spring Boot starter parent : https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-parent/pom.xml**
- **Swagger 2 documentation : https://swagger.io/docs/**
- **Angular Material : https://material.angular.io/**
- **Guide Angular : https://angular.io/start**
- **Single Page Application : https://www.arolla.fr/blog/2018/09/single-page-applications/**
- **Single Page Application : https://blog.octo.com/a-la-decouverte-des-architectures-du-front-3-4-les-single-page-applications/**

XVIII-C - Remerciements

Je voudrais dire un grand merci à tous ceux qui m'ont aidé à rendre ce cours propre. Notamment :
- **Mickael Baron** pour sa relecture technique et pour ses remarques
- **ClaudeLeLoup** pour sa relecture orthographique

- **Winjerome** pour son intervention sur les outils de redaction de ce cours