

# Bibliothèque municipale

Conception du système d'information

# Sommaire

1 – Démonstration

2 – Solutions techniques

2 – 1 Architecture logicielle globale

2 – 2 Architecture logicielle de l'API REST

3 – Implémentations

3 – 1 Implémentation de la DB

3 – 2 Implémentation de l'API REST

3 – 2 Points particuliers

3 – 3 Implémentation de l'UserWebSite

3 – 4 Implémentation du Batch

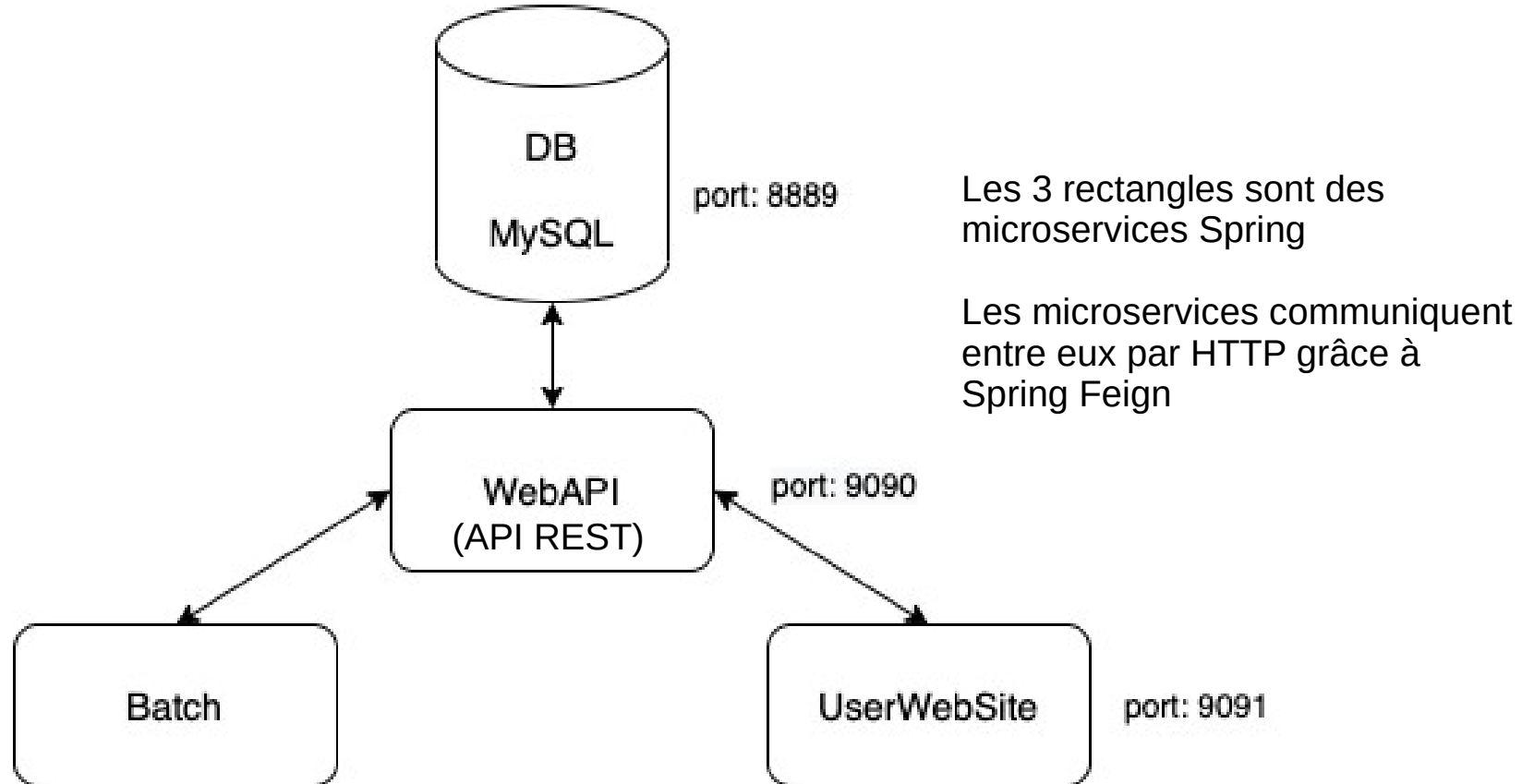
5 – Questions

# 1 - Démonstration

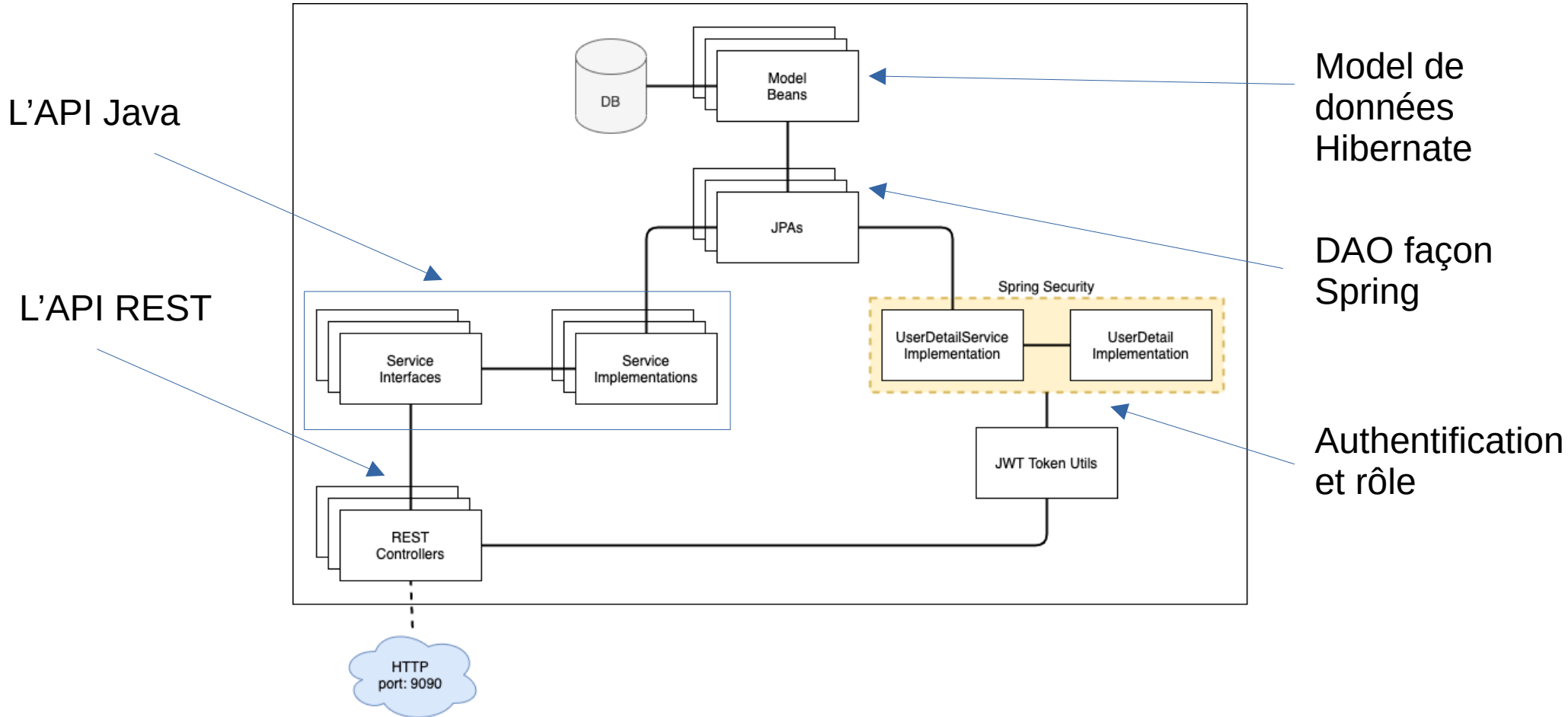
<http://localhost:9091/>

Postman (exemple du REST)

# 2 – 1 Architecture globale



# 2 – 2 Architecture de l'API REST



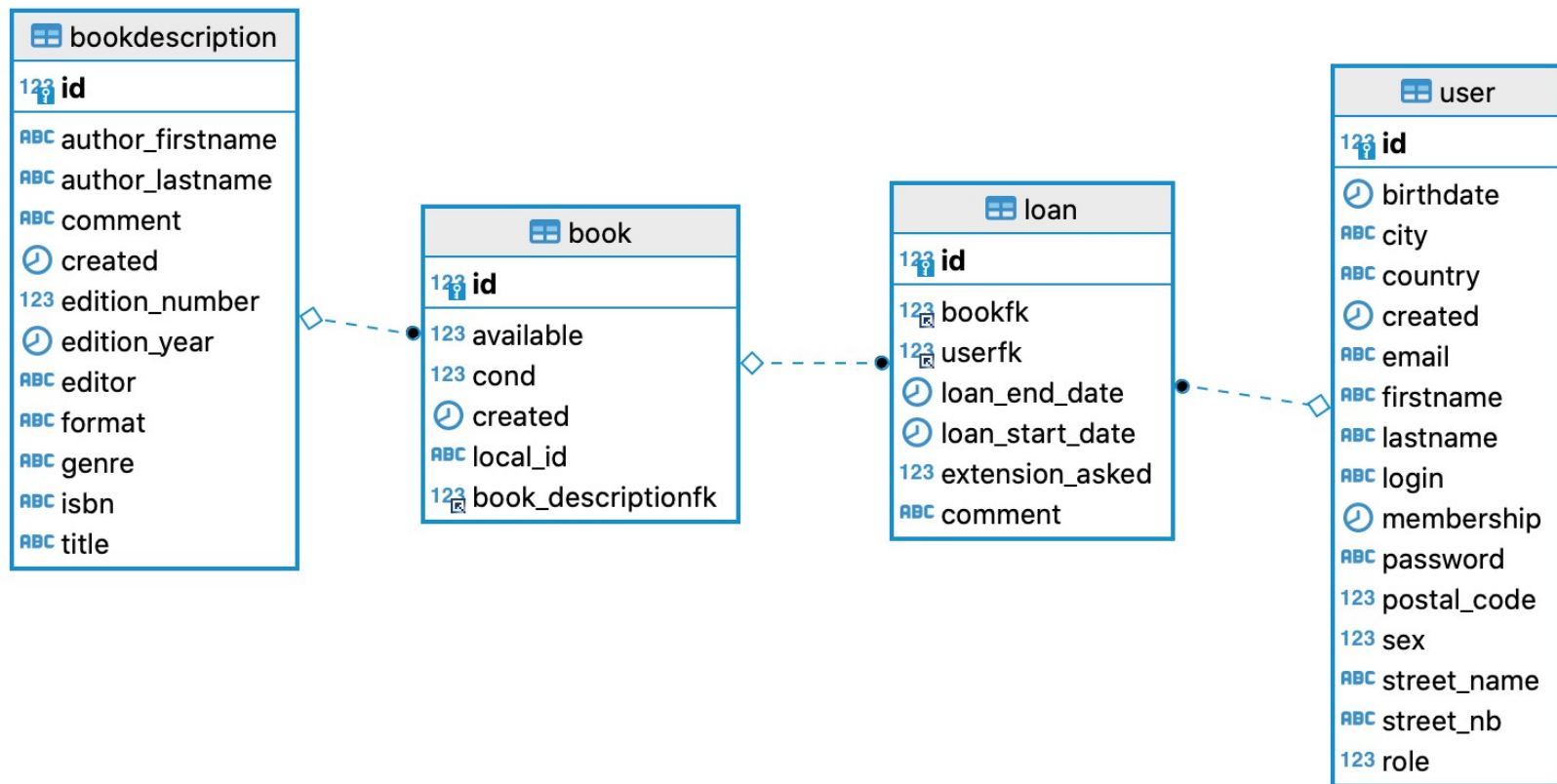
# 2 – 1 Architecture globale

- Gestion des dépendances :
  - Maven (faciliter la gestion des dépendances (les JARs) et des versions)
- Langage :
  - Java 12
- Conteneur de servlet :
  - Apache Tomcat 9
- Framework MVC :
  - Spring et Pebble (moteur de template ~TWIG) avec Spring Boot 2.4.2

# 2 – 1 Architecture globale

- Model :
  - SGBD-R utilisé pour le développement
  - MariaDB / MySQL : SGBD-R utilisé pour une petite production
  - Pour une plus grande production (non testés) : PostgreSQL ou Oracle (payant)
  - Hibernate : gestion de la DB (création des tables, persistance des données et des requêtes SQL) d'une façon orientée JAVA (entités JAVA dans les requêtes SQL)
  - Spring ORM (Object Relational Mapping) basé sur Hibernate (interface entre Hibernate et Spring)
    - Permet de représenter les entités SQL dans des classes JAVA
  - Bibliothèque Lombok : déclare les méthodes usuelles rencontrées dans les beans Java (constructeur, getter/setters, toString)
- Vues :
  - Pebble : moteur de templates pour générer le HTML
  - Bootstrap 5 pour le rendu responsif

# 3 – 1 Implémentation de la DB





# 3 – 2 Implémentation de l'API REST

## Revue générale de code

- Configuration de Tomcat, MySQL et JWT (Tokens) et mailer (simplejavamail) ([.../resources/application.properties](#)) , les valeurs par défaut surchargeables en ligne de commande
- Beans mappés sur la DB avec les annotations javax.persistence.\* et org.springframework.data.annotation.\* ([.../WebAPI](#)) :
- DAOs (Data Access Object) / Repositories basés sur Hibernate, Spring et :
  - 4 interfaces classiques: [BookDAO](#), [BookDescriptionDAO](#), [LoanDAO](#), [UserDAO](#)
  - Les [custom JpaRepositories](#) pour la recherche de livre : interface [SearchJpaRepository](#) + [SearchJpaRepositoryImpl](#)
- Services (l'API java en tant que telle : interfaces et impls) basés sur les beans et les DAOs ci-dessus ([.../webapi/services](#))
- Contrôlers basés sur Spring, et les services ci-dessus ([.../webapi/controllers](#))
- Exceptions gérées par 5 classes : [APIInvalidValueException](#), [APINotAuthorizedException](#), [APIDeletedException](#), [APIFoundException](#), [APIModifiedException](#) et une classe dérivant de [ResponseEntityExceptionHandler](#)

# 3 – 2 Implémentation de l'API REST

Un peu plus de détails....

- security :
  - [jwt](#) (par système de création de tokens) :
    - [AuthEntryPointJwt](#) : Gestion d'exception (voir [.../webapi/security/WebSecurityConfig.java](#))
    - [AuthTokenFilter](#) : filtre de servlet permettant la prise en compte du token par Spring Security
    - [JwtUtils](#) : générateur de tokens, conversion d'un token vers un userName, validateur de token et gestion des possibles erreurs
    -
  - [services](#) :
    - [UserDetailsServiceImpl](#) : récupère l'utilisateur (le bean [.../webapi/models/User.java](#)), par son userName, dans la base de données et le retourne à Spring Security
    - [UserDetailsImpl](#) : implémente les variables qui définissent l'utilisateur dans Spring Security (id, login, password, email et droits) à partir du bean [.../webapi/models/User.java](#).../webapi/models/User.java fourni par [UserDetailsServiceImpl](#)
  - [WebSecurityConfig](#) : configuration de Spring Security et définitions des paths HTTP qui sont exemptés d'authentification et d'autorisation

# 3 – 2 Implémentation de l'API REST

Encore un peu plus de détails....

- services :
  - **Impl** : implémentation de chaque API Java liée aux entités (Book, BookDescription, Loan, Search, User)
    - BookDescriptionServiceImpl, BookServiceImpl, LoanServiceImpl, SearchImpl, UserServiceimpl (plus les **Interfaces** associées)
    - Utilisés par les controllers
- Un controller REST par service : ex. [.../webapi/controllers/BookDescriptionController.java](#)
- autres :
  - **MyApplication** : point d'entrée du microservice Spring
  - **EmailSender** : envoi de mails
  - **MyExceptionHandler** : gestion des exceptions pour les controllers REST (dérive de la classe ResponseEntityExceptionHandler)

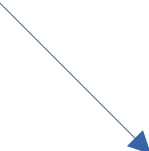
# 3 – 2 Points particuliers

- Tout les beans ont des annotations `@Valid` sur leurs champs pour indiquer des contraintes (non-vide, longueurs minimales, ...)
- Les exceptions levées dans les controllers annotés avec `@Valid` sont traitées dans la classe `ResponseEntityExceptionHandler`, qui surcharge la méthode `handleMethodArgumentNotValid` :

`@Override`

```
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException e, HttpHeaders headers, HttpStatus status, WebRequest request)
{
    Map<String, String> errors = e.getBindingResult().getAllErrors().stream().collect(Collectors.toMap(
        x -> ((FieldError) x).getField(),
        x -> ((FieldError) x).getDefaultMessage()
    ));

    return new ResponseEntity(errors, headers, status);
}
```



Retournera un dictionnaire de noms de champs et de description de l'erreur associée dans le JSON du REST

# 3 – 3 Implémentation de l' UserWebSite

## Revue de code

- Configuration de Tomcat et de VirtualBookcase ([...resources/application.properties](#))
- Beans différenciés :
  - Entités principales : Book, BookDescription, Loan, User
  - Beans liés aux formulaires (multipart) : Credentials (login+pwd), Email FullUserInfo, Search et SearchResult
- Pas de service, mais des proxies Spring Feign vers le code métier via l'API REST
- Vues basées sur HTML + Pebble [UserWebSite/src/main/resources/templates](#))
- Controllers : un controller par fonctionnalité accessible à l'user. Pas de controller sur les loans puisque le controller User a accès aux loans personnels de l'utilisateur.

# 3 – 3 Implémentation de l'UserWebSite

- security : `TokenUtils`, classe utilitaire pour récupérer les informations user stockées dans le token, et de gestion des cookies (qui stocke le token tant que la session est ouverte)
- `MyFeignProxy` : intermédiaire (qui transite via une connexion http) entre l'API et l'UserWebSite.
  - Façon dont le token est propagé et dont les infos utilisateur sont récupérées :

```
@RequestMapping(value = "/loan/{id}/extend", method = RequestMethod.GET)
```

```
public String extendLoan(@CookieValue(TokenUtils.TOKEN_COOKIE_NAME) String token,  
@PathVariable("id") int id, Model model)
```

```
{
```

```
    TokenUtils.UserInfo userInfo = tokenUtils.getUserInfoFromJwtToken(token);
```

```
    model.addAttribute("userInfo", userInfo);
```

```
    feignProxy.extendLoan(token, id);
```

On récupère le token dans un cookie

On récupère les info dans le token

On propage le token vers l'API REST

# 3 – 4 Implémentation du Batch

- 3 beans : `Book`, `BookDescription`, `Loan`
- 1 service : `TaskService`
  - Tache CRON : envoi de mails personnalisés et quotidiens, à heure précise (ici 10h), pour tout les utilisateurs ayant un prêt en retard
  - `EmailSender` : classe pour envoyer des emails
- `MyFeignProxy` : intermédiaire (qui transite via une connexion http) entre l'API et le batch.
  - Permet au batch de s'authentifier et de lister les prêts en retard utilisés par le `TaskService`
- `application.properties` : configuration du port Tomcat et du batch

Questions