

# TP SPRING



**ITg**  
IT GROUPE

STÉPHANE RIGAUT & ROMAIN NOEL

Montreuil, le 31/08/2020



**BNP PARIBAS**

La banque d'un monde qui change

- Prérequis
- Application fil rouge
  - Spring IO
  - Intégration IDE
  - Vérification du squelette
  - Swagger
  - Lancement de l'application
  - On développe ↻
    - Caractéristiques d'une bibliothèque
    - Ecriture de l'objet bibliothèque
    - Ecriture de l'API
    - Ecriture de la persistance
    - Ecriture du service (use-case)
    - Test et vérification manuel de l'API
- Architecture en couche par ordre de priorité (DDD)
  - Couche Domaine
  - Couche Application
  - Couche Exposition
  - Couche Infrastructure
- Gestion des exceptions
- Relation OneToMany
- Amélioration de la tolérance au changement
  - Mapping IHM
  - Isolation du domaine - Inversion des dépendances
- Validation
- Let's finish !



# Prérequis

---

- IntelliJ
- JDK 8
- Maven



- Référentiel de bibliothèques
  - Créer une bibliothèque
    - En retournant l'identité de la nouvelle bibliothèque
  - Afficher le détail d'une bibliothèque
    - Gérer la non existence de la bibliothèque avec une exception (*Runtime dans un premier temps*)
  - Lister l'ensemble des bibliothèques
    - La totalité
    - Par type
    - Par prénom de directeur
  - Mettre à jour une bibliothèque
    - A travers la méthode de création précédemment créée
    - A travers une nouvelle méthode qui contrôle d'existence de la bibliothèque avant la mise à jour
  - Supprimer une bibliothèque
    - En utilisant deleteById



# MISE EN PLACE

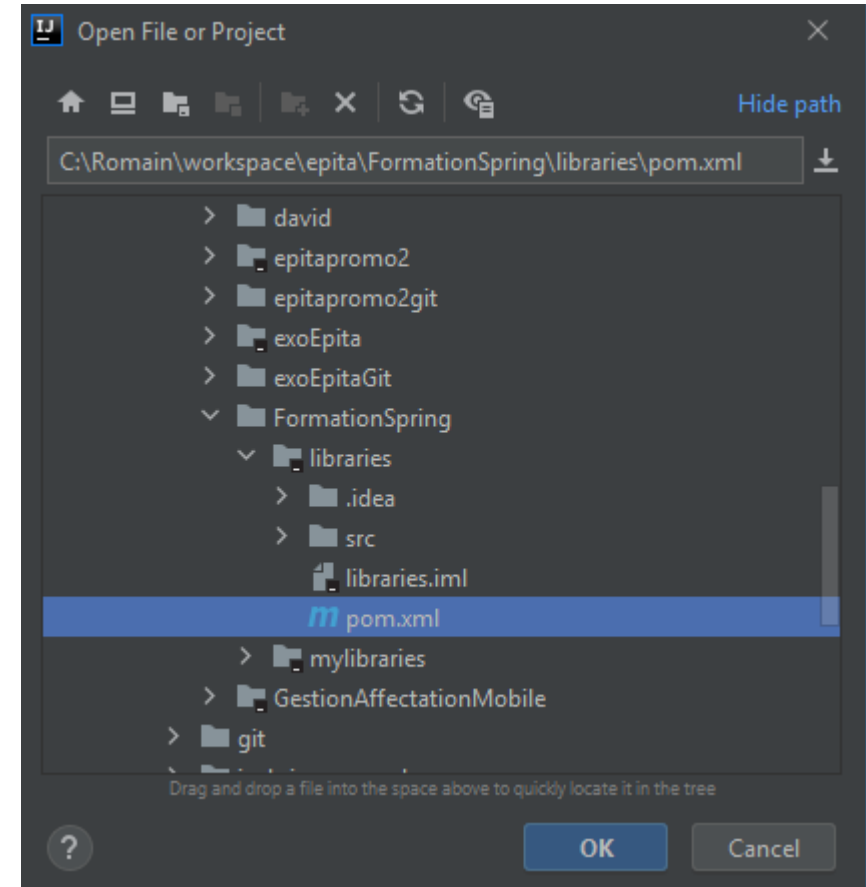


- Initialiser l'application sur <https://start.spring.io/>
- Choisir les critères
  - Project Maven
  - Langage JAVA
  - Spring Boot 2.3.3 (ou dernière version stable)
  - Group com.bnpparibas.itg.mylibraries
  - Artifact libraries
- Choisir les dépendances
  - Spring Web
  - Spring Data JPA
  - H2 Database

*NB : Il existe un plugin IntelliJ Spring initializr permettant d'éviter d'avoir à aller sur le site*



- Extraire dans un répertoire le contenu
- Ouvrir l'IDE
- File -> Open
- Choisir le dossier contenant le squelette
- Double-cliquez sur le pom



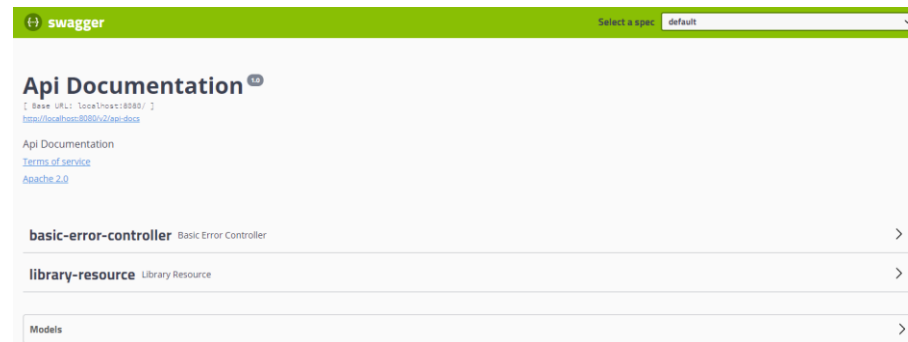
- Vérifier que le fichier pom.xml existe
  - Les dépendances sont correctes
  - Le plugin spring-boot-maven-plugin est présent. Il permet de créer un livrable exécutable (jar, war ...). Il se trouve par défaut dans le répertoire du projet dans /target
  - Tentez un « mvn clean install »
- Vérifier le fichier de configuration de l'application existe
  - Répertoire -> main/resources/application.properties
  - Ajoutez la propriété suivante : `spring.h2.console.enabled=true`

*Cela va nous permettre de visualiser une base de données mémoire (H2) pour le reste de l'exercice*





- Dans le squelette
  - Ajouter les dépendances de swagger dans le pom.xml du projet
    - springfox-swagger2
    - springfox-swagger-ui
  - Copier la classe SwaggerConfig de github et la mettre au même niveau que la classe LibraryApplication
- Ca sert à quoi ?
  - Offre une interface web permettant d'explorer et tester les différentes APIs
  - Permet de générer une documentation des APIs



*NB : Il existe d'autres outils pour effectuer des tests d'api (e.g. : Postman)*



- Lancer l'application
  - Aller dans la classe principale (LibrariesApplication)
  - Cliquez droit -> debug (c'est une action que vous allez faire souvent, pensez à vous mettre un raccourci IntelliJ...)
- Ouvrir un navigateur et aller sur les endpoints (interfaces disponibles)
  - <http://localhost:8080/swagger-ui.html>
  - <http://localhost:8080/h2-console>





# TP-SPRING-0 DOMAINE



- Caractéristiques (attributs)

- Identifiant unique [Long]
- Type (Associative, nationale, publique, scolaire, universitaire) [enum]
- Adresse (numéro, rue, code postal, ville) [int, String, int, String]
- Directeur (prénom, nom) [String, String]

- Bonnes pratiques

- Penser à découper en plusieurs objets plutôt que de mettre tous les attributs dans une seule classe
- Encapsulation
  - Les attributs d'une classe ne peuvent pas être directement manipulés de l'extérieur de la classe
    - Utiliser le mot clé `private` pour définir les attributs de la classe
    - Définir des méthodes d'accès à ces attributs (getter) avec le mot clé `public`



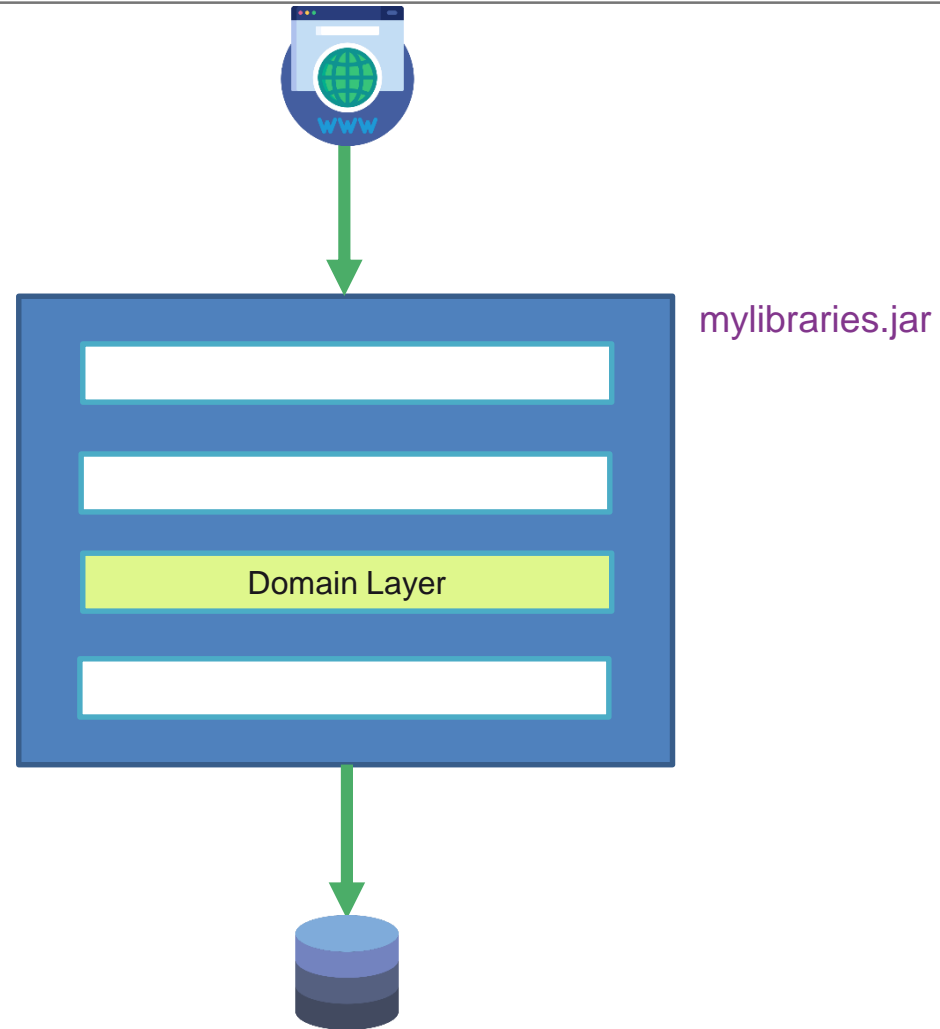
- Penser objet et encapsulation

```
public class Library {  
    public String id;  
    public Type type;  
  
    public int numberAddress;  
    public String streetAddress;  
    public int postalCodeAddress;  
    public String cityAddress;  
  
    public String surnameDirector;  
    public String nameDirector;  
}
```



```
public class Library {  
    private String id;  
    private Type type;  
    private Address address;  
    private Director director;  
  
    public Library(String id, Type type, Address address, Director director) {  
        this.id = id;  
        this.type = type;  
        this.address = address;  
        this.director = director;  
    }  
  
    public String getId() {  
        return id;  
    }  
  
    public Type getType() {  
        return type;  
    }  
  
    public Address getAddress() {  
        return address;  
    }  
  
    public Director getDirector() {  
        return director;  
    }  
}
```







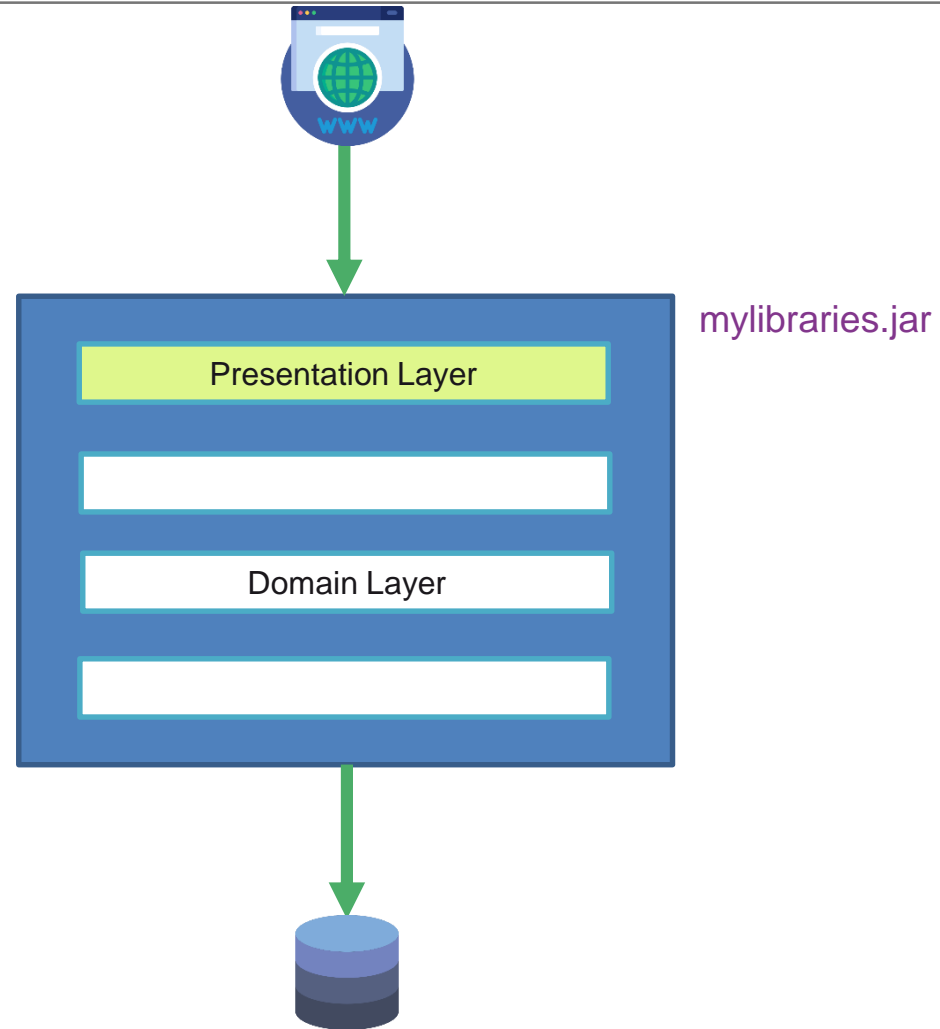
# TP-SPRING-0 PRÉSENTATION / EXPOSITION



- Bonnes pratiques : <https://blog.octo.com/designer-une-api-rest/>
  - KISS
  - Granularité moyenne
  - Noms > verbes
  - Pluriel > singulier
  - Casse (spinal-case)
  - CRUD (POST, GET, PUT, DELETE)
- Annotations
  - @RestController (@Controller + @ResponseBody)
  - @GetMapping, @PostMapping, @PutMapping, @DeleteMapping
  - @RequestBody, @PathVariable, @RequestParam
  - @ResponseStatus





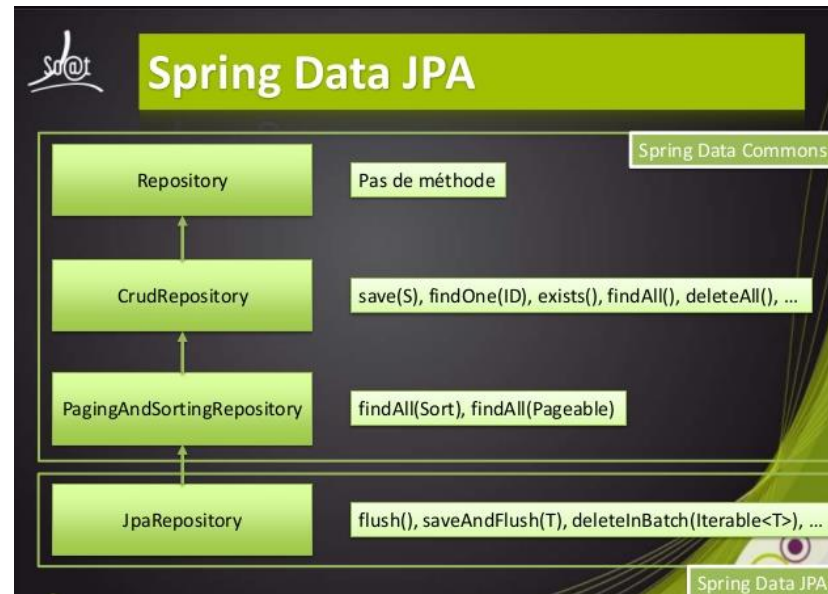




# TP-SPRING-0 INFRASTRUCTURE



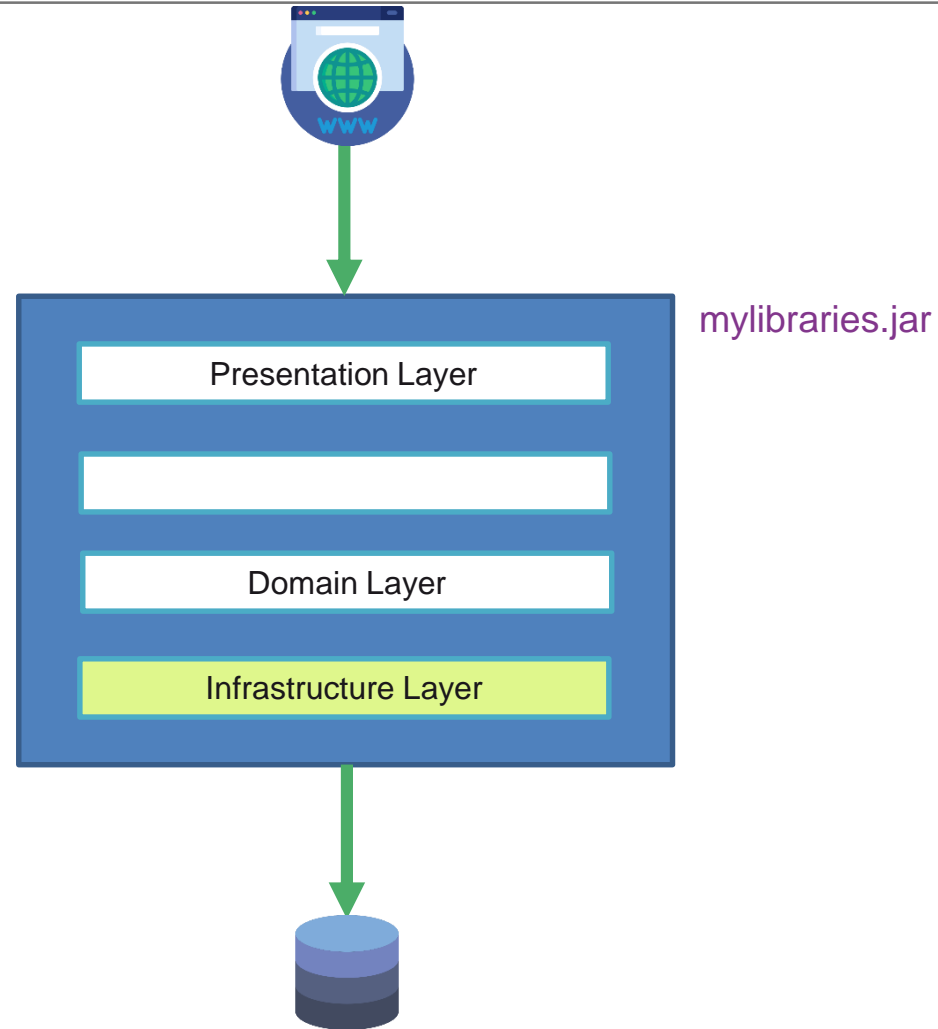
- Utilisation de spring-data-jpa
  - CrudRepository (CRUD opérations)
  - PagingAndSortingRepository (fournit en plus la possibilité de paginer et d'ordonner)
  - JpaRepository (fournit en plus la possibilité de forcer la synchronisation avec la base de données -> flush)



- Choisir le nom de la table associée à la classe avec @Entity
- Utiliser l'attribut @Column pour définir le nom de la colonne dans la table associé à l'attribut
- Identifier l'attribut unique et choisir le type de génération
  - @Id
  - @GeneratedValue
- Pour notre type de bibliothèque qui utilise un enum
  - @Enumerated(EnumType.STRING)
- Définir les relations avec les autres objets
  - Si les données des objets en relations sont stockées dans la même table
    - @Embedded sur l'attribut
    - @Embeddable sur la classe fille
  - Si les données des objets en relations sont stockées dans une table à part
    - @OneToOne
    - @OneToMany
    - @ManyToOne

*NB : Pour le moment on les stockera dans la même table, nous verrons les relations dans le TP3*







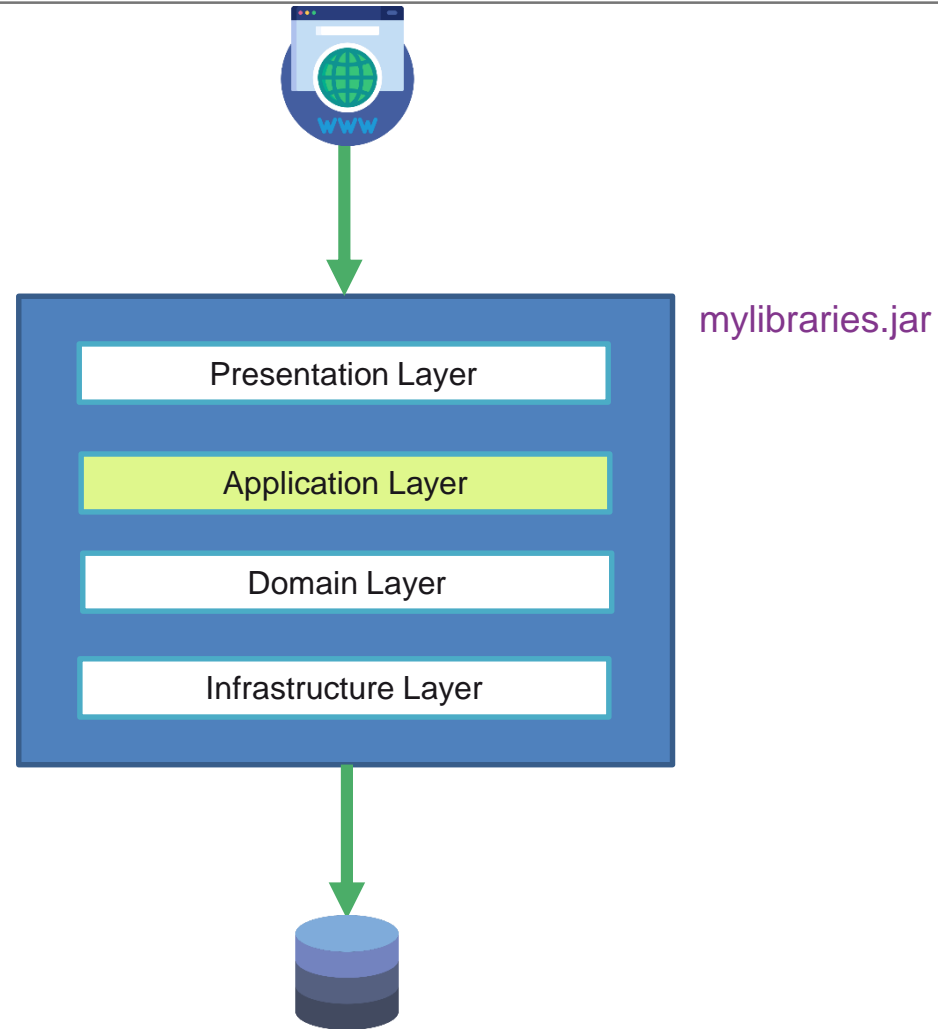
# TP-SPRING-0 APPLICATION



# Ecriture du service (use-case)

- Implémentation de/des la fonctionnalité(s) métier
- Déclare les méthodes qui doivent s'exécuter dans un contexte transactionnel
- Reçoit les requêtes/appels des entrées publiques (APIs, web, etc)
- Annotations
  - @Service
  - @Transactional







# Test et vérification manuelle de l'API (tp-spring-0) ↺

---

- <http://localhost:8080/swagger-ui.html>
- <http://localhost:8080/h2>



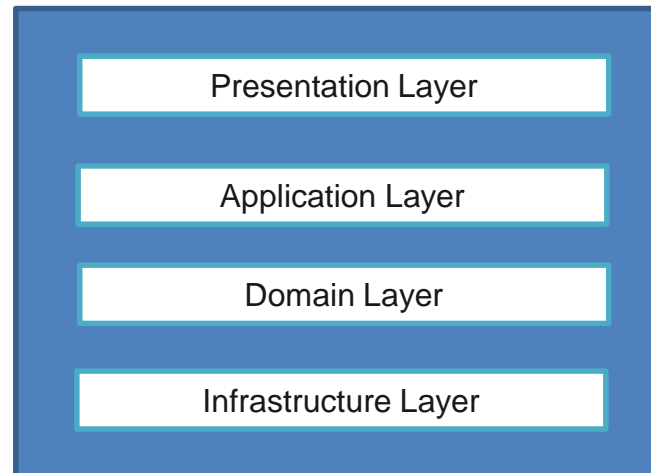
- Définir une classe de test et l'annoter avec l'annotation suivante :
  - @SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM\_PORT)
  - Cela permet de lancer les tests avec un contexte Spring initialisé (ainsi que la base H2)
- Pour effectuer des tests d'api, vous aurez besoin du composant suivant :
  - @Autowired private TestRestTemplate restTemplate;
  - Plus d'infos sur son utilisation ici : <https://www.baeldung.com/spring-boot-testresttemplate>
- Pensez que vos tests doivent être FIRST (Fast/**Indépendant**/Replayable/Self-validating/Thorough)  
-> Vous aurez certainement besoin de remettre à zéro votre base entre chaque test





# Architecture en couche (tp-spring-1)

- Chaque couche à une responsabilité
- Permet d'isoler le code métier du reste du code
- Le code métier est représenté par la couche application et la couche domaine
- La définition de l'API se retrouve dans la couche présentation
- L'implémentation du code d'accès à la base de données se trouve dans la couche infrastructure





- Protégée du reste du monde (aucune dépendance vers d'autres modules)
- Contient la logique métier (règles métier)
- Utilisation de pattern tactiques
  - Entities (Objets ayant une identité qui reste la même au cours de l'application indépendamment de leurs attributs)
  - Value Objects (Objets sans identité définis seulement par leurs attributs)
  - Repositories (abstraction de la persistance – agnostique des détails techniques du stockage)
  - ...







- Implémentation des use cases (fonctionnalités métier)
- Coordinateur (appelle d'autres application services si besoin)
- S'occupe du contexte transactionnel
- Reçoit les requêtes/appels des entrées publiques (APIs, web, etc) via la couche exposition
- Annotations
  - @Service
  - @Transactional





# Exposition

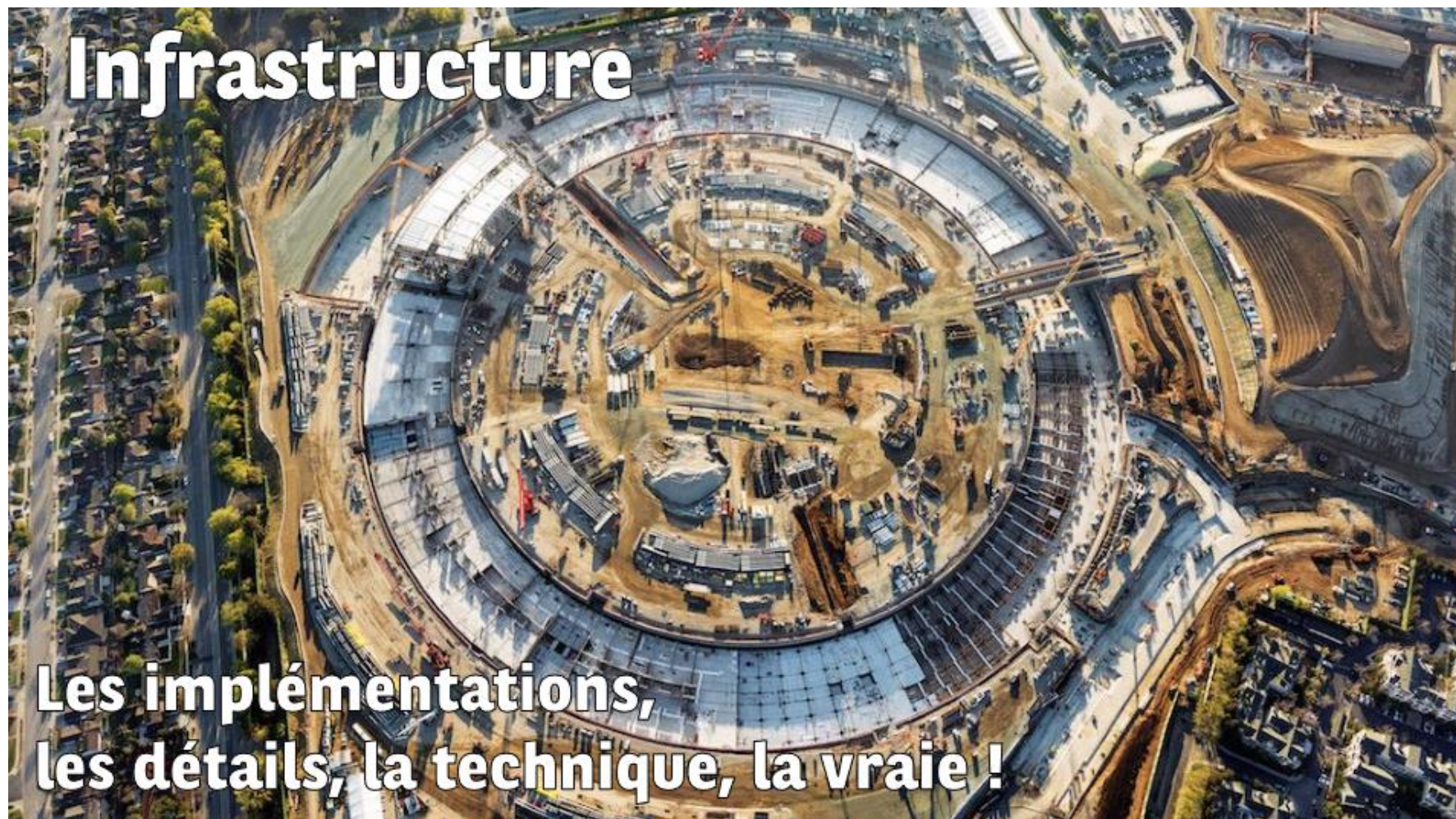
**La porte ouverte  
sur le monde extérieur**



- Ouvert au monde extérieur
- Exposition des APIs REST
- Adaptateurs
- DTO (Data Transfer Object)
- Annotations
  - @RestController (@Controller + @ResponseBody)
  - @RequestMapping (GET, POST, PUT, DELETE ...)
  - @RequestBody, @PathVariable, @RequestParam
  - @Valid
  - ResponseEntity







- Implémentation des détails techniques
  - Configuration et accès aux stockages de données
  - Envoi de mails
  - Audit
  - Appels à des services externes à l'application
- Couche d'anti-corruption
  - Adapter les résultats des appels externes aux objets du domaine



- Refactorer vos classes en 4 packages java spécifiques

*N.B. : Dans vos projets, vous utiliserez des modules maven différents*



# TP-SPRING-2



- Pouvoir gérer les exceptions de manière centralisé
  - Utilisation d'exception de type unchecked (runtime)
  - Créer sa propre exception qui hérite de RuntimeException
  
- Il existe plusieurs méthodes pour gérer les exceptions de contrôleurs :
  - 1 - Dans les contrôleurs :
    - A - avec l'annotation `@ExceptionHandler`
    - B - avec un try/catch, en lançant une `ResponseStatusException`
  - 2 - Dans une classe à part, grâce aux l'annotations `@ControllerAdvice` et `@ExceptionHandler` (quasiment identique au 1A, mais peut s'appliquer à plusieurs contrôleurs)
  - 3 - Dans les Exceptions, avec l'annotation `@ResponseStatus`

Documentation ici : <https://www.baeldung.com/exception-handling-for-rest-with-spring/>



# TP-SPRING-3





- Nous voulons pouvoir créer une bibliothèque avec des livres

- Créer l'objet livre

- Caractéristiques

- Id [Long]
- Titre [String]
- Auteur [String]
- Nombre de page [int]
- Genre littéraire [enum]

- Définir le mapping avec la base de données

- @Entity

- @Id

- @GeneratedValue

- @Column

- @Enumerated(EnumType.STRING)

- Mettre en relation la bibliothèque avec les livres

- @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)

- *Optionnel* : @JoinColumn(name="LIBRARY\_ID", referencedColumnName = "ID")

<https://www.baeldung.com/hibernate-one-to-many>

OU

- Mettre en relation la bibliothèque avec les livres

- @ManyToMany(cascade = {CascadeType.MERGE, CascadeType.PERSIST}, fetch = FetchType.LAZY)

- *Optionnel* : @JoinTable( name = "lib\_book", joinColumns = @JoinColumn(name = "lib\_id"), inverseJoinColumns = @JoinColumn(name = "book\_id"))

<https://www.baeldung.com/jpa-many-to-many>



- L'objet Library :
  - Représente notre objet métier ainsi que sa logique (règles métier)
  - Mapping avec le monde extérieur (REST / JSON)
  - Mapping avec la base de données (JPA)



Que se passe-t-il si nous modifions le modèle Library ?

- Impact direct sur l'IHM et sur la base de données



Solution ?

- L'idée est d'isoler l'objet métier Library et de créer deux nouveaux objets
  - Un objet LibraryDTO (représentant les données de l'IHM) (tp-spring-4)
  - Un objet LibraryJPA (représentant les données à persister en base) (tp-spring-5)





- Créer un objet LibraryDTO qui correspond aux données entrante reçu de l'IHM
  - Utiliser l'annotation @JsonProperty pour le mapping JSON vers un objet JAVA
  - Veillez à respecter les relations entre les objets pour une meilleure lisibilité
- Créer une classe permettant de transformer les données de l'IHM vers le domaine et du domaine vers l'IHM (LibraryAdapter)

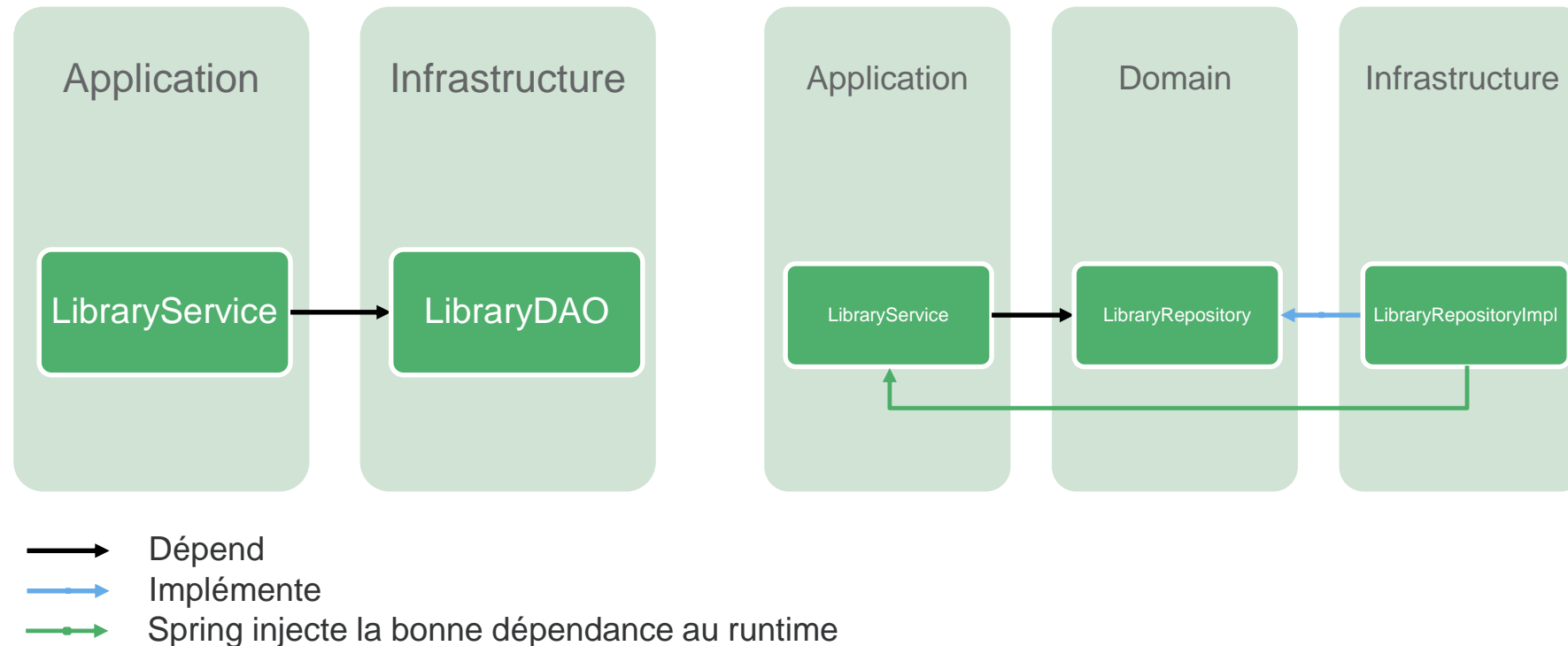


# TP-SPRING-5



# Isolation du domaine – Inversion des dépendances (tp-spring-5)

- Pour isoler complètement le domaine et le rendre agnostique de toute technologie sous-jacente
  - Inversion des dépendances
  - Détacher le mapping de l'entité métier avec la base de données
- Inversion des dépendances





- Nous voulons vérifier qu'un directeur est bien présent
- Où mettre le code associé à cette règle ?
- Plusieurs possibilités
  - Au niveau des adaptateurs avec la vérification des contraintes
    - On utilise les annotations de `javax.validation.constraints.*` et `@Valid` dans la signature des APIs
    - Permet d'effectuer les contrôles de surface simple
    - Ne permet pas de protéger le domaine, on peut toujours créer une bibliothèque sans contrôle
  - Au niveau du domaine, on effectue les différents contrôles souhaités
    - Le domaine est ainsi protégé de l'extérieur





# TP-SPRING-7



# Let's finish ! (tp-spring-7)

---

- DDD références pratiques
  - @DDD.Entity
  - @DDD.ValueObject
  - @DDD.Repository
  - @DDD.RepositoryImpl
  - @DDD.ApplicationService
- Egalité entre les entités
- Egalité entre les objets valeurs
- Test unitaire pour vérifier ces égalités

