



# Framework Spring

Introduction

Par Richard BONNAMY

# Introduction

# Objectifs de ce support de cours

- Présentation de **Spring**
- Présentation de **Spring Boot**
- Le principe d'injection de dépendances (**IoC**)

# Introduction

# Qu'est-ce que Spring ?

## **Spring est un framework**

- Boîte à outils Java, socle facilitateur pour le développement d'applications.
- Aujourd'hui principalement utilisé pour le développement côté serveur avec **Spring MVC** ou **Spring Web**.
- **Spring** permet de développer n'importe quel type d'applications. Par exemple **Spring Batch** pour le traitement type batch et la mise en base de fichiers.
- **Spring Framework** est souvent abrégé simplement **Spring**
- **Spring** a quelques concurrents comme **Jersey pour le développement web**

# Qu'est-ce qu'apporte Spring ?

- Fournit des modules, des « briques » communes à la plupart des applications Java modernes
- Favorise les bonnes pratiques (design patterns, architecture...)
- Spring est modulaire : <https://spring.io/projects>

# Bref historique

- **Octobre 2002** : Rod Johnson publie son livre «Expert One-on-One J2EE Design and Development», dans lequel il propose du code, qui va devenir plus tard le framework Spring. Il sort la 1<sup>ère</sup> version en 2003.
- **Mars 2004** : Spring 1.0 sort sous licence Apache 2.0
- **Octobre 2006** : Spring 2.0
- **2007** : Spring 2.5, avec support des annotations
- **Décembre 2009** : Spring 3.0
- **2014** : Spring 4 & Spring Boot
- **Juin 2016** : Spring 4.3 – Dernière versions qui supporte Java 6
- **Novembre 2022** : Spring 6.0 avec support Java 17+ et écosystème « Jakarta »



## Spring Boot



# Qu'est-ce que Spring Boot ?



## Spring Boot :

- facilite la configuration des applications reposant sur le framework Spring.
- préconfigure de nombreuses choses
- permet d'aller à l'essentiel, à savoir développer les besoins du client.

Avec **Spring Boot** et des modules additionnels, nous allons construire une application côté serveur très rapidement :

- Configuration très simple.
- Permettant d'exposer une API "REST" grâce au module **Spring Web**
- Permettant de gérer la persistance de nos données grâce au module **Spring Data JPA**

# Créer une application Spring Boot avec Spring Initializr

- Plus simple avec **Spring Boot**
- Contient un serveur intégré – ne nécessite donc pas le déploiement sur un serveur type **Tomcat**.
- On peut initialiser un projet **Spring Boot** avec **Spring Initializr** qui génère une archive à dézipper sur votre ordinateur : [Spring Initializr](https://start.spring.io)

The screenshot shows the Spring Initializr web application interface. The browser address bar displays 'start.spring.io'. The page features the Spring logo and the text 'spring initializr'. The main content area is divided into several sections:

- Project:** Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'.
- Language:** Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
- Spring Boot:** Includes radio buttons for '2.3.0 M4' (selected), '2.3.0 (SNAPSHOT)', '2.2.7 (SNAPSHOT)', '2.2.6', '2.1.14 (SNAPSHOT)', and '2.1.13'.
- Dependencies:** Includes a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'.
- Project Metadata:** Includes input fields for 'Group' (com.example), 'Artifact' (demo), and 'Name' (demo), and a 'Description' field (Demo project for Spring Boot).

At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. Social media icons for GitHub and Twitter are visible on the left side.

# Spring Boot – Gestion des dépendances

- **Spring Boot** fournit des **starters**.
- Un starter est une dépendance qu'on intègre à MAVEN et qui va installer automatiquement toutes les librairies nécessaires.



**Spring Boot**

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

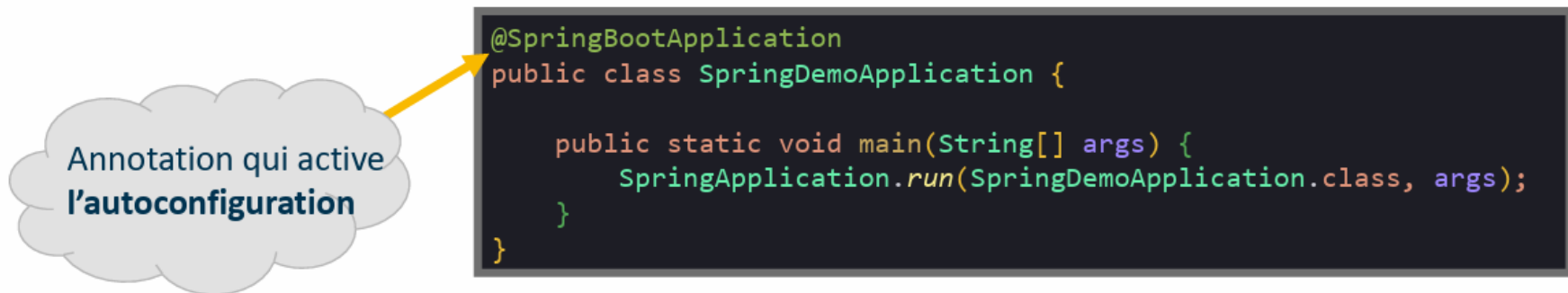


**Spring**

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>5.3.5</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>5.3.5</version>
</dependency>
```

# Spring Boot – Autoconfiguration

- **Spring Boot préconfigure** les modules inclus
- Exemple : la couche d'accès aux données avec JPA si vous avez le module Spring Data JPA.



- Le développeur se concentre sur les fonctionnalités demandées par le client

# Spring Boot – properties

- un fichier de configuration unique : **application.properties**
- A mettre dans **src/main/resources**
- **Spring Boot** facilite l'accès aux propriétés depuis le code

# Spring Boot – déploiement

- **Spring Boot** se déploie sous la forme d'une archive Java (fichier .jar) exécutable
- Avec **Spring Web**, cette archive contient un serveur d'application Tomcat embarqué (embedded).
- Lorsque vous démarrez votre application (classe avec une méthode main), cette dernière lance automatiquement Tomcat et expose vos services REST sur <http://localhost:8080> par défaut.
- En **quelques minutes** vous disposez d'une application **Spring Boot**...
- **Mais place à la preuve...à vous de jouer... !!!**

## **TP n°1: Créer une application Hello avec Spring Initializr**

## Injection de dépendances



# Spring – Injection de dépendance

- **Spring** est un framework construit autour de la notion d'**injection de dépendances**.
- Spring crée les instances de classes pour nous et ensuite on lui demande de les **injecter** là où on en a besoin avec des **annotations**

```
@RestController
@RequestMapping("/hello")
public class HelloControleur {

    @Autowired
    private UtilisateurService service;

    @GetMapping
    public String getDonnee(){

        return service.extractDonnee();
    }
}
```

Plus besoin d'écrire  
UtilisateurService service =  
new UtilisateurService();

# Spring – Notion de bean

- **Spring** peut gérer le cycle de vie de vos instances de classes.
- Par exemple, pour que Spring crée une instance de la classe **Configuration** au démarrage de l'application, il suffit que je positionne l'annotation **@Component** au-dessus.
- **Les étapes du démarrage de l'application:**
  - **Spring** scanne tous les packages pour détecter les classes annotées avec certaines annotations comme **@Component**, **@Service**, **@Configuration**.
  - **Spring** crée une instance de chacune de ces classes
  - Ces instances sont stockées dans un espace mémoire géré par **Spring** appelé **conteneur IoC** (Inversion Of Control)
  - Puis **Spring** injecte ces instances partout où elles sont demandées

# Spring – Que signifie Inversion of Control ?

- **Inversion of Control** est un principe de conception logiciel.
- Ce n'est pas un design pattern au sens strict du terme.
- Il exprime que le contrôle de la création et de la gestion des objets est inversé :
  - **Sans IoC** : Le programmeur est responsable de créer les objets et de gérer leurs dépendances manuellement (par exemple vous êtes obligé d'utiliser un setter pour créer une dépendance entre 2 instances)
  - **Avec IoC** : Le framework (dans ce cas, le conteneur Spring) est responsable de la création des objets et de la gestion de leurs dépendances.

# Spring – Notion de bean

- **Spring** va créer **une instance** de la classe **Configuration** et la stocker en mémoire dans son **conteneur** qu'on appelle le **conteneur IoC** (Inversion Of Control)

```
@Component
public class Configuration {

    public Configuration(){
        System.out.println("Coucou je suis vide pour l'instant");
    }
}
```

```

[_____] .___[ ] [ ] [ ] \__, | / / / /
=====|_|=====|___/=//_/_/_/
:: Spring Boot ::                      (v2.7.18)

2023-11-23 11:31:41.104 INFO 19876 --- [
2023-11-23 11:31:41.109 INFO 19876 --- [
Coucou je suis vide pour l'instant
2023-11-23 11:31:42.002 INFO 19876 --- [

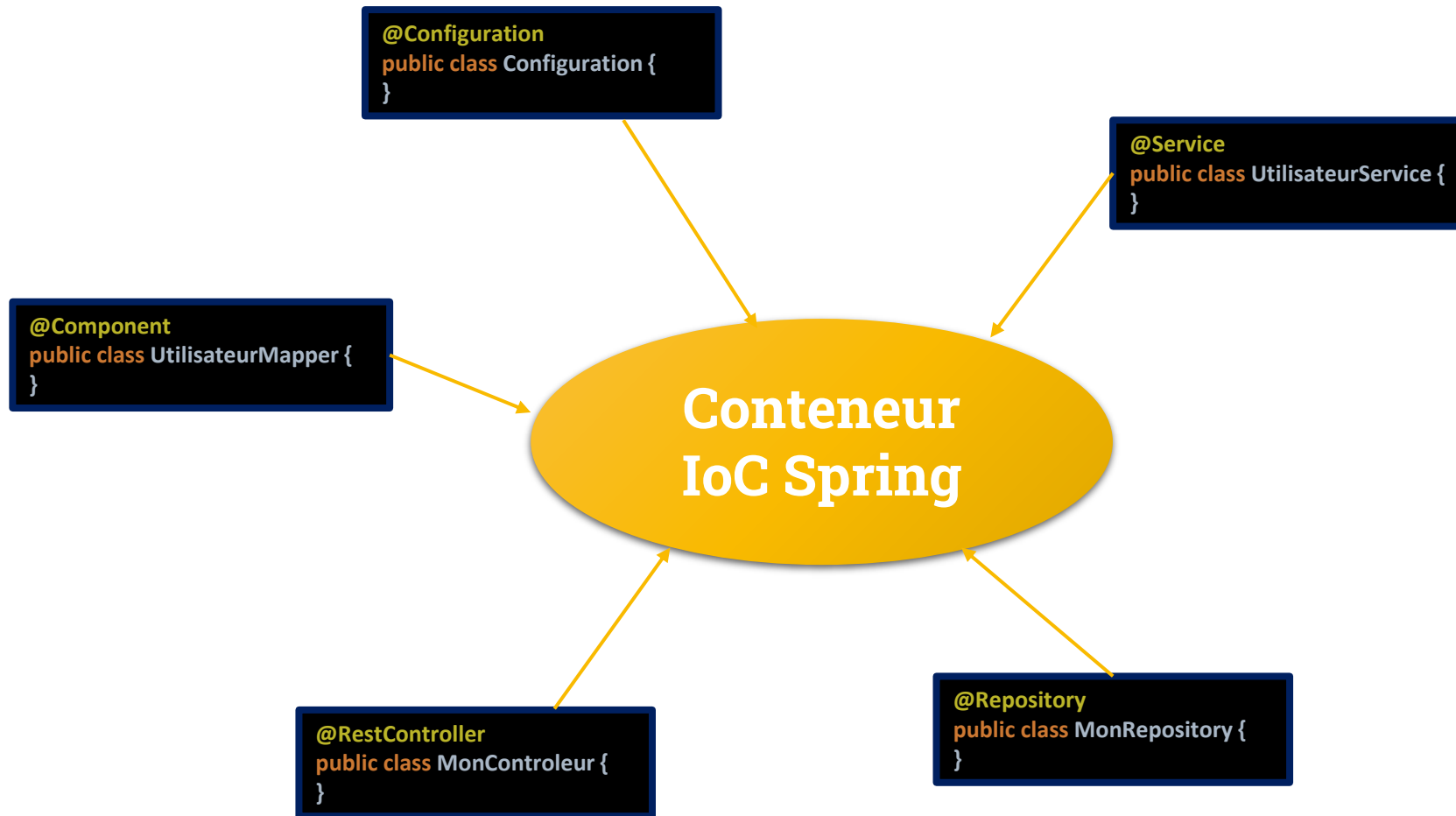
Process finished with exit code 0

```

# Spring – Liste des annotations générant des beans

- Un **bean Spring** est un objet dont le cycle de vie est géré par **Spring**.
- Toutes les classes annotées avec une des **annotations suivantes** vont devenir des **beans Spring**.
  - @Component
  - @Service
  - @Configuration
  - @Repository
  - @RestController
- Lorsque **Spring** tombe sur une classe annotée avec l'une des annotations ci-dessus, il en crée une instance et la stocke en mémoire dans son **conteneur IoC**.

# Spring – En résumé



- Au démarrage de l'application, **Spring scanne les packages** et crée une instance de chaque classe annotée pour les mettre dans le **conteneur**.
- Chacune de ces instances est appelée un **bean Spring**

# Spring – Utiliser un bean Spring

```
@Configuration
public class Configuration {

    public Configuration(){
        System.out.println("Coucou je suis vide pour l'instant");
    }
}
```

- Si je souhaite injecter l'instance du **bean Configuration** dans une de mes classes, j'utilise **@Autowired**

```
@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

    @Autowired
    private Configuration conf;

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(conf);
    }
}
```

# Spring – Exemple sans Spring

- Exemple pour créer une instance d'EntityManager **sans Spring** :
  - Mettre en place un fichier **persistence.xml** dans src/main/resources/**META-INF**
  - Créer une instance de EntityManagerFactory, puis créer une instance d'entityManager :

```
@RestController
@RequestMapping("/utilisateurs")
public class UtilisateurControleur {

    @GetMapping
    public List<Utilisateur> extraireUtilisateurs(){

        EntityManagerFactory emf = Persistence.createEntityManagerFactory("unit1");
        EntityManager em = emf.createEntityManager();

        // Requête ici

        return utilisateurs;
    }
}
```



# Spring – Exemple avec Spring

## ➤ Exemple de la classe précédente avec Spring :

- Les paramètres d'accès à la base sont renseignés dans le fichier **application.properties**
- L'annotation **@PersistenceContext** permet de demander à **Spring** de créer une instance d'entityManager
- Les informations utiles (URL de la base, user, password) sont contenues dans le fichier **application.properties**

```
@RestController
@RequestMapping("/utilisateurs")
public class UtilisateurControleur {

    @PersistenceContext
    private EntityManager em;

    @GetMapping
    public List<Utilisateur> extraireUtilisateurs(){

        // Requête ici

        return utilisateurs;
    }
}
```

# Spring – Utiliser un de mes composants

- Attention, **@Autowired** ou **@PersistenceContext** ne fonctionne que si la classe dans laquelle se trouve cette annotation est elle-même un **bean** Spring.
- Dans l'exemple précédent, la classe est annotée avec **@RestController** qui fait de cette classe bean Spring.

```
@RestController
@RequestMapping("/hello")
public class HelloControleur {

    @PersistenceContext
    private EntityManager em;

    @GetMapping
    public String direHello(){

        // Requête ici

        return "Hello";
    }
}
```

# Spring – Les 3 manières de faire de la DI

- **DI = Dependency Injection** / Injection de dépendance
- Il y a 3 manières d'injecter une dépendance :
  - Via un attribut d'instance
  - Via un constructeur
  - Via une méthode
- Dans tous les cas, l'attribut d'instance, le constructeur ou la méthode doit appartenir à un bean Spring.

# Spring – DI sur attribut d'instance

- **Premier cas** : sur un attribut d'instance
- C'est le cas que nous avons vu précédemment en plaçant un **@Autowired** sur un attribut d'instance d'une classe :

```
@RestController
@RequestMapping("/hello")
public class HelloControleur {

    @Autowired
    private Configuration config;

}
```

- Ici **@Autowired** désigne à Spring l'attribut qu'il doit instancier, c'est donc Spring qui va valoriser cet attribut avec l'instance qu'il a dans son conteneur

# Spring – DI sur constructeur

- **Second cas** : sur les paramètres du constructeur
- Dans ce cas on demande à Spring d'injecter un bean Spring en paramètre de ce constructeur:

```
@Service
public class UtilisateurService {

    private Configuration config;

    @Autowired
    public UtilisateurService(Configuration config){
        this.config = config;
    }
}
```

- Attention, les paramètres du constructeur doivent tous être des beans

# Spring – DI sur un paramètre de méthode

- **Troisième cas** : sur un paramètre de méthode, par exemple un setter
- Dans ce cas on place **@Autowired** sur la signature de la méthode :

```
@Service
public class UtilisateurService {

    private Configuration config;

    @Autowired
    public void setConfig(Configuration config) {
        this.config = config;
    }
}
```

# Spring – Récupérer une valeur dans le fichier de configuration

- Il est possible de créer vos propres propriétés dans le fichier **application.properties**
- Pour quoi faire ? Pour influencer sur le comportement d'une application
- Exemple :
  - Fichier application.properties :

```
init.base=false
```

- Injection dans un bean spring :

```
@Value("${init.base}")  
private boolean initBase;
```

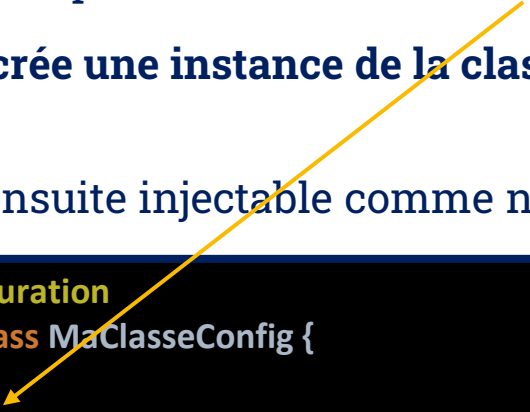
# Spring – Créer un bean Spring via une méthode

- Il est également possible de créer un **bean** Spring via une méthode annotée de @Bean
- **La méthode crée une instance de la classe Configuration qui est placée dans le container IoC.**
- Ce **bean** est ensuite injectable comme n'importe quelle autre bean.

```
@Configuration
public class MaClasseConfig {

    @Bean
    public Configuraton genererConfig(){

        return new Configuration();
    }
}
```





## **TP n°2: Créer des beans Spring et vérifier qu'ils sont bien instanciés**



Spring Boot  
Fournisseur de  
données

# Spring Boot – Créer des APIs avec Spring Web

- L'utilisation principale de **Spring Boot** est la création d'**APIs**.
- Le but d'une API est d'échanger des données avec le monde extérieur.
- Pour créer une API, Spring Boot ne suffit pas, il faut le module **Spring Web**.

# Spring Boot – Protocole HTTP

- Les échanges reposent sur le **protocole HTTP**.
- L'utilisation du protocole HTTP est transparente, c'est le mode d'échange des informations sur le web.
  - Lorsque vous cliquez sur un lien href, une requête HTTP GET est générée par votre navigateur vers le serveur.
  - Lorsque vous cliquez sur le bouton valider d'un formulaire, une requête HTTP POST est générée par votre navigateur vers le serveur.

# Spring Boot – Les 4 verbes HTTP

- Les **4 verbes HTTP** suivants permettent de réaliser toutes les opérations sur les données :
- **GET**: permet de demander au serveur de retourner des données.
  - **POST** : permet de demander au serveur la création d'une donnée.
  - **PUT** : permet de demander au serveur la modification d'une donnée.
  - **DELETE** : permet de demander au serveur la suppression d'une donnée.

store Access to Petstore orders		
GET	/store/inventory	Returns pet inventories by status
POST	/store/order	Place an order for a pet
GET	/store/order/{orderId}	Find purchase order by ID
DELETE	/store/order/{orderId}	Delete purchase order by ID

# Spring Boot – Fournisseur de données

- Un serveur peut renvoyer 2 types de données :
  - Une page HTML construite côté serveur : avec le module **Spring MVC** et **Thymeleaf** par exemple
  - Ou des données au format JSON (standard actuel) : avec le module **Spring Web**
  - Dans le second cas, un framework front type **Angular**, **ReactJS** ou **VueJS**, doit être mis en place pour exploiter les données (i.e. générer du HTML à partir des données).

# Spring Boot – Création du serveur

- Lorsque vous démarrez une application **Spring Boot** dans laquelle vous avez inclus le module **Spring Web** :
  - l'application démarre automatiquement un serveur Tomcat sur le port 8080
- Pour accéder à ce serveur local : <http://localhost:8080>
- A partir de là, l'application Spring Boot traite toutes les requêtes HTTP qui arrivent sur cette adresse.
  - On peut utiliser **Postman** par exemple.
- **Il est alors possible de créer vos propres routages en renvoyant les requêtes entrantes vers vos méthodes.**

# Spring Boot – Exemple de routage

GET : <http://localhost:8080/hello>

→ Classe PolitesseControleur, méthode direHello()

GET : <http://localhost:8080/bye>

→ Classe PolitesseControleur, méthode direBye()

GET : <http://localhost:8080/villes>

→ Classe DataControleur, méthode getVilles()

- Avec **Spring Web**, vous créez vos routes et les classes qui fournissent des données : message ou données plus complexes comme une ArrayList de Ville par exemple.
- **Notion de Contrôleur** : classe dont les méthodes sont mappées sur des routes (URL+verbe HTTP) particulières.



# Spring Boot – relier un contrôleur Spring à une route

## ➤ Les contrôleurs Spring :

- traitent les requêtes HTTP envoyées par le client sur une route donnée (exemple : <http://localhost:8080/inventory> avec verbe **GET**)
- Renvoient des données ou une page HTML selon le module utilisé

```
@RestController
@RequestMapping("/store")
public class StoreControleur {

    @Autowired
    private PetService service;

    @GetMapping(path = "/inventory")
    public List<Pet> getPets(){

        List<Pet> pets = service.extractAll();
        return pets;
    }
}
```

# Spring Boot – relier un contrôleur Spring à un verbe HTTP

- Un **contrôleur Spring** peut contenir plusieurs méthodes **GET, PUT, POST** et **DELETE** à condition que les URL soient différentes:
- Exemple avec 2 méthodes mappées sur des routes **GET**

```
@GetMapping(path = "/inventory")
public List<Pet> getPets(){

    List<Pet> pets = service.extractAll();
    return pets;
}

@GetMapping(path = "/inventory/specialOffer")
public List<Pet> getPetsPromotions(){

    List<Pet> pets = service.extractSpecialOffer();
    return pets;
}
```

# Spring Boot convertit automatiquement en JSON les données renvoyées

- Dans l'exemple ci-dessous les méthodes renvoient des listes d'animaux familiers (Pet).
- La librairie Jackson incluse dans Spring Boot convertit automatiquement les données en JSON
- **Attention s'il y a des cycles dans les attributs !!**

```
@GetMapping(path = "/inventory")
public List<Pet> getPets(){

    List<Pet> pets = service.extractAll();
    return pets;
}

@GetMapping(path = "/inventory/specialOffer")
public List<Pet> getPetsPromotions(){

    List<Pet> pets = service.extractSpecialOffer();
    return pets;
}
```

# Exemple de cycle provoquant une boucle infinie dans Jackson

- Dans l'exemple ci-dessous, le renvoi par Spring Boot d'une liste de villes ou de départements provoque une boucle infinie au niveau de la couche de transformation en JSON.

```
@Entity
public class Departement {

    @OneToMany
    private List<Ville> villes = new ArrayList<>();
}
```

```
@Entity
public class Ville {

    @ManyToOne
    @JoinColumn(name="id_dept")
    private Departement departement;
}
```

- La présence de cycles est très fréquente dans les entités JPA.

## **TP n°3: Créer un contrôleur et échanger des données avec lui**



## Spring Boot les requêtes

# Spring Boot – route paramétrée

- On peut définir une **URL contenant un paramètre (placeholder) dans l'URL même**:
  - Structure d'une URL paramétrée : /monUrl/{nomParametre}
  - La méthode doit dans ce cas posséder un paramètre avec le même nom (id dans l'exemple) et le paramètre doit avoir l'annotation **@PathVariable**
  - **@PathVariable** permet de demander à Spring de valoriser le paramètre.

```
@GetMapping(path = "/inventory/{id}")  
public Pet getPet(@PathVariable Long id){  
  
    Pet pet = dao.extractById(id);  
    return pet;  
}
```

- Dans l'exemple ci-dessus si j'appelle <http://localhost/store/inventory/128> alors id vaut 128

# Spring Boot – requête avec paramètre

- On peut également définir une **requête paramétrée**:
  - Structure d'une requête avec paramètre : `/monUrl?id=valeur`
  - La méthode doit dans ce cas posséder un paramètre avec le même nom (**id** dans l'exemple) et le paramètre doit avoir l'annotation **@RequestParam**

```
@GetMapping(path = "/inventory")
public Pet getPet(@RequestParam Long id){

    Pet pet = dao.extractById(id);
    return pet;
}
```

- Dans l'exemple ci-dessus si j'appelle <http://localhost/store/inventory?id=128> alors l'id récupéré en paramètre de méthode vaut 128



# Spring Boot – requête avec paramètre

## ➤ Exemple avec plusieurs requêtes GET

```
@RestController
@RequestMapping("/store")
public class StoreControleur {

    @Autowired
    private PetDao dao;

    @GetMapping(path = "/inventory")
    public List<Pet> getPets(){

        List<Pet> pets = dao.extractAll();
        return pets;
    }

    @GetMapping(path = "/inventory/{id}")
    public Pet getPet(@PathVariable Long id){

        Pet pet = dao.extractById(id);
        return pet;
    }
}
```

# Spring Boot – envoyer des données au serveur

- Suivant qu'on veut créer ou modifier une donnée, on va utiliser **POST** ou **PUT**.
- Les données doivent être envoyées au serveur au **format JSON**.
- Spring va transformer les messages reçus au format JSON en instance de vos classes.
  - Attention pour cela il faut que votre classe ait des attributs portant le même nom que dans le message

```
{
```

```
  "nom": "Tours",
```

```
  "nbHabs": 136000
```

```
}
```

```
public class Ville {  
    private String nom;  
    private int nbHabs;  
  
    public Ville(String nom, int nbHabs) {  
        this.nom = nom;  
        this.nbHabs = nbHabs;  
    }  
  
    // + GETTERS et SETTERS  
}
```

# Spring Boot – La méthode du contrôleur

- Dans le contrôleur on va créer par exemple une méthode appeler **insertPet** qui prend en paramètre une instance de Pet.
- Cette méthode possède l'annotation **@PostMapping**
- Pour appeler cette méthode il faudra donc lui **envoyer un message JSON** avec les attributs name et id, dans une requête **HTTP POST**.

```
{  
  "name": "Suki",  
  "id": 127  
}
```

**POST**

Pour réaliser cette  
requête on utilise  
POSTMAN ou  
javascript

```
@PostMapping  
public ResponseEntity<String> insertPet(@RequestBody Pet nvPet){  
  
    dao.insertPet(nvPet);  
    return ResponseEntity.ok("Success !");  
}
```

# Spring Boot – A quoi sert @RequestBody ?

- L'annotation **@RequestBody** est indispensable sinon Spring ne transformera pas le JSON en instance de votre classe.

Pour réaliser cette  
requête on utilise  
POSTMAN ou  
javascript

```
{  
  "name": "Suki",  
  "id": 127  
}
```

**POST**

```
@PostMapping  
public ResponseEntity<String> insertPet(@RequestBody Pet nvPet){  
  
    dao.insertPet(nvPet);  
    return ResponseEntity.ok("Success !");  
}
```

# Spring Boot – A quoi sert ResponseEntity ?

- Une **ResponseEntity** correspond à une réponse HTTP.
- Vos méthodes peuvent retourner au choix :
  - Des données. Dans ce cas c'est Spring qui génère la réponse HTTP en y incluant vos données.
  - Une ResponseEntity contenant les données et un statut.
- On retourne en générale une ResponseEntity si on souhaite associer les données avec un statut.
- Dans le cas ci-dessous, je retourne un message Success ! Associé au statut HTTP 200 (méthode ok)

```
@PostMapping
public ResponseEntity<String> insertPet(@RequestBody Pet nvPet){

    dao.insertPet(nvPet);
    return ResponseEntity.ok("Success !");
}
```

# Spring Boot – Retourner une erreur au client.

- Avec la **ResponseEntity** il est possible de retourner une erreur au client.
- Dans l'exemple ci-dessous, on réalise un test afin de vérifier si le pet existe déjà en base de données ou non.
- S'il existe déjà on retourne une réponse HTTP avec code erreur 400 et un message d'erreur !

```
@PostMapping
public ResponseEntity<String> insertPet(@RequestBody Pet nvPet){

    if (dao.extractById(nvPet.getId())!=null){
        return ResponseEntity.badRequest().body("A pet with this id already exists !");
    }

    dao.insertPet(nvPet);
    return ResponseEntity.ok("Success !");
}
```

## **TP n°4 et 5: Envoyer des données au serveur et mise en place des méthodes CRUD**



## L'API Bean Validation



# API Bean Validation - Présentation

- Il s'agit d'un module Spring Boot qui permet de réaliser des contrôles simples sur les objets de l'application.
  - Sur les objets reçus du front par exemple.
- Il faut ajouter le starter dans le pom.xml

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>
```

# API Bean Validation – annotations

- Cette API permet de définir des contraintes sur les attributs d'une classe via des annotations :
  - @NotNull : l'attribut doit être non nul
  - @Size : permet de définir une taille min et/ou max sur un attribut de type String
  - @Range : permet de définir une valeur min et/ou max sur un attribut de type Number
  - @Min: permet de définir une valeur min sur un attribut de type Number
  - @Max: permet de définir une valeur max sur un attribut de type Number

# API Bean Validation– Exemple

➤ Voici un exemple :

```
public class Client {  
  
    @NotNull  
    @Size(min=1, max=100)  
    private String nom;  
  
    @NotNull  
    @Size(min=1, max=100)  
    private String prenom;  
  
}
```

# API Bean Validation – déclenchement du contrôle

- Le déclenchement du contrôle peut se faire de plusieurs manières
- **Premier cas** : dans le contrôleur, dans la signature de la méthode avec **@Valid**.
  - Attention dans ce cas il faut injecter un paramètre de type **BindingResult** qui va recevoir automatiquement le résultat des contrôles :

```
@PostMapping
public void insertVille(@Valid @RequestBody Client client, BindingResult result) {

    if (result.hasErrors()){
        return ResponseEntity.badRequest().body(result.getAllErrors().get(0).getDefaultMessage());
    }
    // ...
}
```

*Une exception est jetée avec le message d'erreur du premier contrôle non passant.*

- Dans l'exemple ci-dessus on retourne une "bad request" mais on peut aussi jeter une exception comme on le verra par la suite.

# API Bean Validation – déclenchement du contrôle

- **Second cas** : dans le corps de la méthode avec une instance de **Validator** et sa méthode `validateObject` qui retourne un objet `Errors` contenant ou non des erreurs.

```
@RestController
@RequestMapping("/clients")
public class ClientControleur {

    @Autowired
    private Validator validator;

    @PostMapping
    public void insertVille(@RequestBody Client client) {

        Errors errors = validator.validateObject(client);

        if (errors.hasErrors()){
            return ResponseEntity.badRequest().body(result.getAllErrors().get(0).getDefaultMessage());
        }

        // ...
    }
}
```

*Une exception est jetée avec le message d'erreur du premier contrôle non passant.*

# API Bean Validation – contrôle sur le dto ?

- On peut mettre en place les contrôles, soit sur un DTO soit sur l'objet métier.
- Si les contrôles **ne dépendent pas d'un contexte** il est préférable de les centraliser au niveau de l'objet métier.
- Si les contrôles **dépendent d'un contexte** il est préférable de les positionner sur un DTO. Rappelons qu'un objet métier peut avoir plusieurs représentations.
- Exemple :
  - application logistique avec une classe Cargaison (avec attribut poids) dont le contrôle sur le poids dépend du contexte.
  - Dans ce cas il ne serait pas pertinent de mettre une annotation sur l'attribut poids.
- Question :
  - le contrôleur reçoit un DTO mais je veux faire les contrôles sur l'objet métier. Comment faire ?

## Structure d'une API

# Qu'est-ce qu'une API ?

- **API** : ensemble de définitions et de protocoles qui facilite la création et l'intégration de logiciels d'applications (source : RedHat)
- Une API peut prendre différentes formes :
  - un ensemble d'interfaces qui représente un contrat,
  - un ensemble de librairies (API LocalDate de java 8),
  - **des données exposées sur le web au format JSON.**



# Qu'est-ce que REST ?

- **REST** est un ensemble de contraintes que l'API doit respecter
- Voici les contraintes principales :
  - Les échanges sont gérés via HTTP
  - Les échanges sont stateless.
    - Entre 2 requêtes GET aucun état/historique n'est conservé.
    - Les requêtes sont indépendantes les unes des autres.
  - L'API fournit pour chaque ressource une liste de liens définissant toutes les actions possibles sur cette ressource.
- Les vraies architectures REST sont rares et souvent on parle d'API REST de manière abusive.
- "La plupart des soi-disant API REST sont en réalité du RPC (Remote Procedure Call)" – Roy Fielding.

# Bien concevoir les routes de l'application

- Une **bonne définition des routes** d'une application est un **aspect stratégique** de la conception d'une API
- La **sémantique** d'une API doit être **intuitive**.
- On peut par exemple mettre en place des **routes hiérarchiques** de manière à bien structurer son API
- Exemple : routes fictives d'un site de vols internationaux

HTTP	Routes	Données fournies
GET	/pays	Fournit la liste des pays desservis
GET	/pays/{codePays}	Fournit des infos sur un pays spécifique.
GET	/pays/{codePays}/aeroports	Fournit la liste des aéroports d'un pays donné
GET	/aeroport/{code}	Fournit des infos sur un aéroport donné
GET	/aéroport/{code}/vols	Fournit la liste des vols d'un aéroport donné
GET	/aéroport/{code}/vols/{date}	Fournit la liste des vols d'un aéroport donné À une date donnée
POST	/aeroport/	Ajout d'un nouvel aéroport
DELETE	/aeroport/{code}	Suppression d'un aéroport d'un code donné

# A éviter

- Eviter les routes qui font la même chose au sein de l'API :

HTTP	Routes
GET	/pays/FR
GET	/paysByCode/FR
GET	/pays?code=FR

- Eviter les termes pas clairs (intuitivité) : éviter les termes génériques ou les abréviations.

HTTP	Routes	Données fournies
GET	/liste	Fournit la liste des aéroports
GET	/liste/{code}/infos	Fournit la liste des vols de l'aéroport
GET	/clis	Fournit la liste des clients

## Accès à la base de données

# Spring JDBC vs Spring Data JPA.

- Nous avons vu comment échanger des données avec le serveur
- Nous n'avons pas encore vu comment relier le serveur à une base de données
- Il existe 2 manières de le faire :
  - Avec le module **Spring JDBC** : bas niveau, repose sur JDBC, utilisation de requêtes SQL natives.
  - Avec le module **Spring Data JPA** : plus haut niveau, mapping ORM, utilisation de JPA/Hibernate
- Dans les 2 cas ne pas oublier le **module d'accès à la base de données** (MySQL par exemple)

# Spring Boot – Spring Data JPA.

- Pas de fichier ~~persistence.xml~~
- Les données d'accès à la base de données sont renseignées dans le fichier **application.properties**.

**spring.jpa.hibernate.ddl-auto=update**

**spring.datasource.url=jdbc:mysql://localhost:3306/mabase**

**spring.datasource.username=root**

**spring.datasource.password=password**

**spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver**

# Spring Boot – Spring Data JPA.

- Les instances d'**EntityManager** sont créées automatiquement dans le **conteneur IoC**.
- Il est possible d'**injecter** une **instance d'EntityManager** directement dans le code avec l'annotation **@PersistenceContext**
- Inutile de gérer le cycle de vie des EntityManager. C'est géré par le conteneur IoC.

```
@Service
public class PetDao {

    @PersistenceContext
    private EntityManager em;

    public List<Pet> extractPets() {
        TypedQuery<Pet> query = em.createQuery("SELECT p FROM Pet p", Pet.class);
        return query.getResultList();
    }
}
```

# Spring Boot – Gestion des transactions.

- Les ouvertures et fermetures de transaction sont également gérées par Spring
- Il suffit juste de préciser quelles méthodes ont besoin d'une transaction avec l'annotation **@Transactional**

```
@Service
public class PetDao {

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public void insertPet(Pet nvPet){
        em.persist(nvPet);
    }
}
```



# Spring Boot – Création avec POST.

- La requête **POST** est utilisée généralement pour créer une donnée
- Le **body** contient l'objet à créer.

```
@PostMapping
public void insertPet(@RequestBody Pet pet){

    dao.insertPet(pet);
}
```

# Spring Boot – Modification avec PUT.


- La requête **PUT** est utilisée généralement pour modifier une donnée à partir de son id.
- Une requête paramétrée peut être utilisée pour transmettre l'**id**
- Le **body** contient l'objet modifié.

```
@PutMapping("/{id}")  
public void modifPet(@PathVariable int id, @RequestBody Pet pet){  
  
    dao.modifPet(id, pet);  
}
```

# Spring Boot – Modification avec PUT.

➤ Avec une DAO :

```
@PostMapping("/{id}")  
public void modifPet(@PathVariable int id, @RequestBody Pet pet){  
  
    dao.modifPet(id, pet);  
}
```




```
@Transactional  
public void modifPet(int id, Pet pet) {  
    Pet petFromDB = em.find(Pet.class, id);  
    if (petFromDB!=null) {  
        petFromDB.setNom(pet.getNom());  
    }  
}
```

# Spring Boot – Suppression avec DELETE.

- La requête **DELETE** est utilisée pour supprimer une donnée à partir de son id.
- Une requête paramétrée est utilisée pour transmettre l'id

```
@DeleteMapping("/{id}")  
public void deletePet(@PathVariable int id){  
  
    dao.deletePet(id);  
}
```



```
@Transactional  
public void deletePet(int id) {  
  
    Pet petFromDB = em.find(Pet.class, id);  
    em.remove(petFromDB);  
}
```

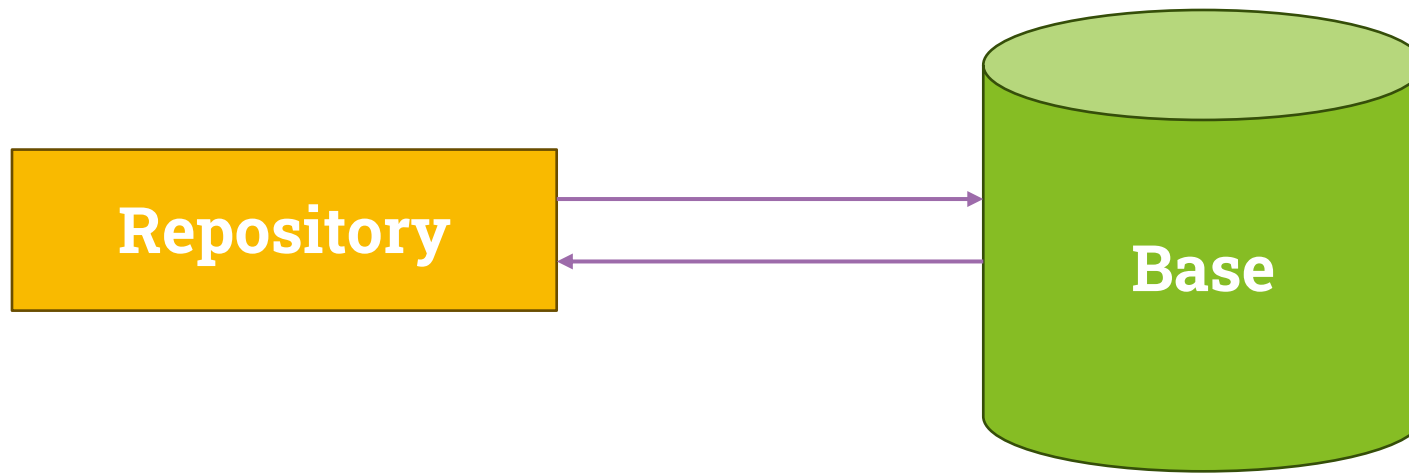
## **TP n°6: Réalisation d'une API complète avec base de données**



## Les Repository avec Spring Data JPA

# Spring Data JPA et Repository

- Semblable au concept de **DAO** (Data Access Object)
- Rôle = exécuter les actions pour communiquer avec la BDD
- Un repository fournit des méthodes prêtes à l'emploi



# Création d'un repository

- Le concept de **Repository** est un concept original dans l'univers Java car ce sont des interfaces ne comportant que des signatures de méthodes.
- Il faut créer une interface qui hérite de **CrudRepository**
- Cet héritage doit préciser 2 types génériques :
  - l'entité concernée
  - le type d'identifiant de l'entité
- **Exemple :**

```
public interface VilleRepository extends CrudRepository<Ville, Integer> {  
}
```



# Création d'un repository

## ➤ Liste des méthodes disponibles par défaut

Nom de la méthode	A quoi elle sert ?
<code>findAll()</code>	Extrait toutes les entités de la base de données
<code>findById(int id)</code>	Extrait une entité de la base de données en fonction de son id
<code>existsById(int id)</code>	Retourne si oui ou non l'entité existe avec cet id
<code>saveAll(Iterable iter)</code>	Sauve toutes les entités passées en paramètres
<code>save(...)</code>	Sauve l'entité passée en paramètre
<code>delete(...)</code>	Supprime l'entité passée en paramètre
<code>deleteById(int id)</code>	Supprime l'entité dont l'id est passé en paramètre

# Dérivation de requêtes

- La **dérivation de requêtes** permet de **générer automatiquement** des requêtes **JPQL** à partir des noms des méthodes.
- Les noms de méthodes sont interprétés et traduits en requêtes JPQL selon les conventions de nommage spécifiques qui prennent en compte les propriétés des entités

```
public interface VilleRepository extends CrudRepository<Ville, Integer> {  
  
    Ville findByNom(String nom);  
}
```

- Lorsque j'invoque la méthode **findByNom(String nom)** avec par exemple "Pau", **Spring Data JPA** génère automatiquement la requête JPQL:

SELECT v FROM Ville v WHERE v.nom='Pau'

# Dérivation de requêtes : règles de nommage

- Les règles de nommage – préfixes de noms de méthodes :
  - findBy..., readBy..., queryBy..., getBy... qui fonctionnent de manière identique. Ce n'est qu'une question de préférence
  - findFirstBy...
  - findTopNBy... qui permet de trouver les N premiers résultats d'une requête.
    - Exemple : findTop3By...
  - countBy... qui permet de compter un nombre d'entités pour un critère donné
  - existsBy... qui permet de contrôler l'existence de données, retourne un boolean
  - deleteBy..., removeBy...

# Exemples d'extractions

## ➤ Autres exemples :

```
public interface VilleRepository extends CrudRepository<Ville, Integer> {  
  
    Ville findByNom(String nom);  
  
    Ville findByNomAndPopulation(String nom, int pop);  
  
    Ville findByNomAndDepartement(String nom, Departement dept);  
  
    List<Ville> findBy(); // équivalent de findAll()  
  
    List<Ville> findByOrderByPopulation(); // findAll() avec tri par population  
  
    // Recherche les 10 plus grandes villes pour un code  
    // département donné. Tri par population descendante.  
    List<Ville> findTop10ByDepartementOrderByPopulationDesc(Departement dept);  
}
```

# Exemples de requêtes diverses

## ➤ Autres exemples :

```
public interface VilleRepository extends CrudRepository<Ville, Integer> {  
  
    Optional<Ville> findByNom(String nom);  
    Map<Departement, Integer> countByDepartementCode(String codeDept);  
    boolean existsByNomAndDepartementCode(String nom , String codeDept);  
  
}
```

# Mise en place de méthode spécifique

- La technique précédente fonctionne pour des requêtes simples.
- Parfois il n'est pas possible de trouver le nom de méthode qui conviendra à votre besoin.
- Il est donc possible d'associer une requête JPQL à une méthode.
- On utilise **@Query** au-dessus du nom de la méthode et avec la **requête JPQL**

```
public interface VilleRepository extends CrudRepository<Ville, Integer> {  
  
    Ville findByNom(String nom);  
  
    List<Ville> findByOrderByNom();  
  
    Ville findByNomAndNbHabs(String nom, int nbHabs);  
  
    @Query("SELECT v FROM Ville v WHERE v.nom in (:nom1, :nom2)")  
    Ville maMethode(String nom1, String nom2);  
}
```

# JpaRepository vs CrudRepository

- Spring Data JPA fournit d'autres interfaces comme **JpaRepository**
- **JpaRepository** hérite de **CrudRepository** et fournit donc quelques méthodes complémentaires

Nom de la méthode	A quoi elle sert ?
findAll(Pageable pageable)	Extrait toutes les entités de la base de données avec une pagination
flush()	Pour synchroniser le contexte de persistance avec la base de données
saveAndFlush(T entity)	Sauvegarde immédiatement l'entité en base
saveAllAndFlush(Iterable<T> iter)	Même chose mais pour un ensemble
deleteAllInBatch(Iterable<T> iter)	Supprime toutes les entités
deleteAllByIdInBatch(Iterable<T> iter)	Supprime toutes les entités par id

# Exemple de pagination

- La pagination vous permet de récupérer côté front des résultats page par page.
- Il vous suffit de préciser le numéro de la page courante souhaitée ainsi que sa taille. Le principe est de faire correspondre ces données à votre tableau d'affichage paginé.
- La première page est 0 et la méthode à utiliser est `findAll(Pageable pageable)`
- **Exemple :**

```
@GetMapping("/pagination")
public Page<Ville> getVilles(@RequestParam int page, @RequestParam int size){

    PageRequest pagination = PageRequest.of(page, size);
    return villeRepository.findAll(pagination);
}
```

- Page : numéro de la page souhaitée
- Size: nombre d'éléments à extraire (i.e. afficher côté front)

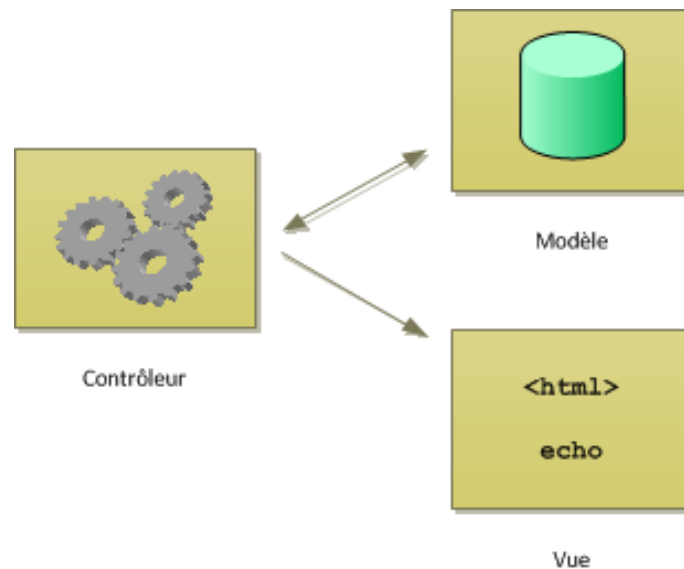


## **TP n°7: Modification de l'API précédente avec des repositories**

## Notion de DTO

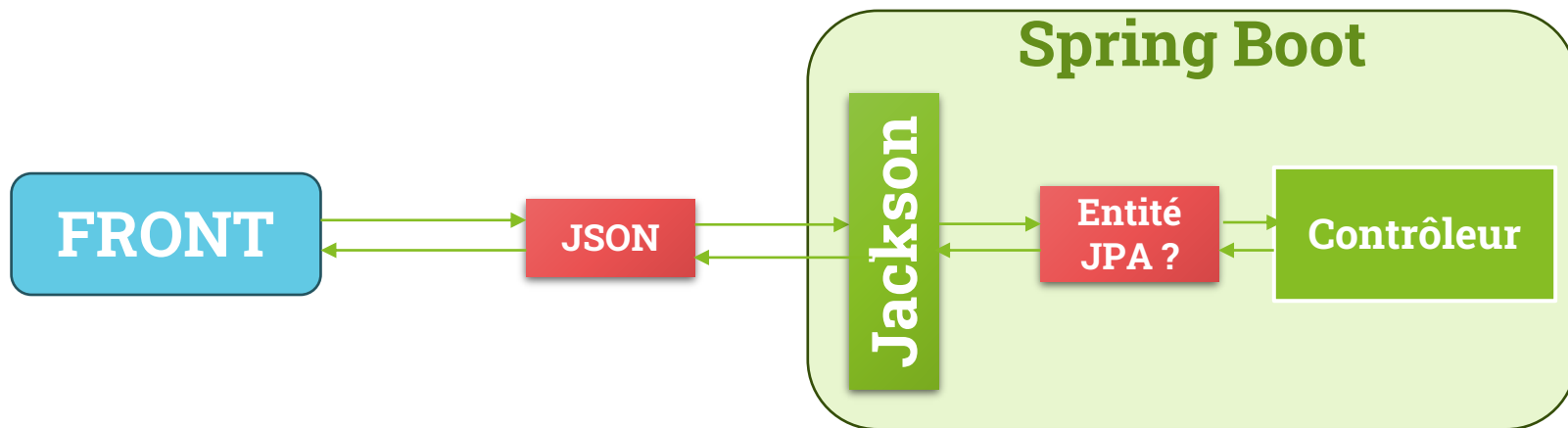
# Le pattern MVC

- **MVC:** Modèle, Vue, Contrôleur
- Dans ce pattern, la "Vue" représente l'IHM, mais peut aussi représenter un objet exploité par la vue: même si un DTO ne fait pas spécifiquement partie de la Vue il peut être considéré comme un pont entre le modèle et la vue. Il peut contenir également des éléments de présentation.



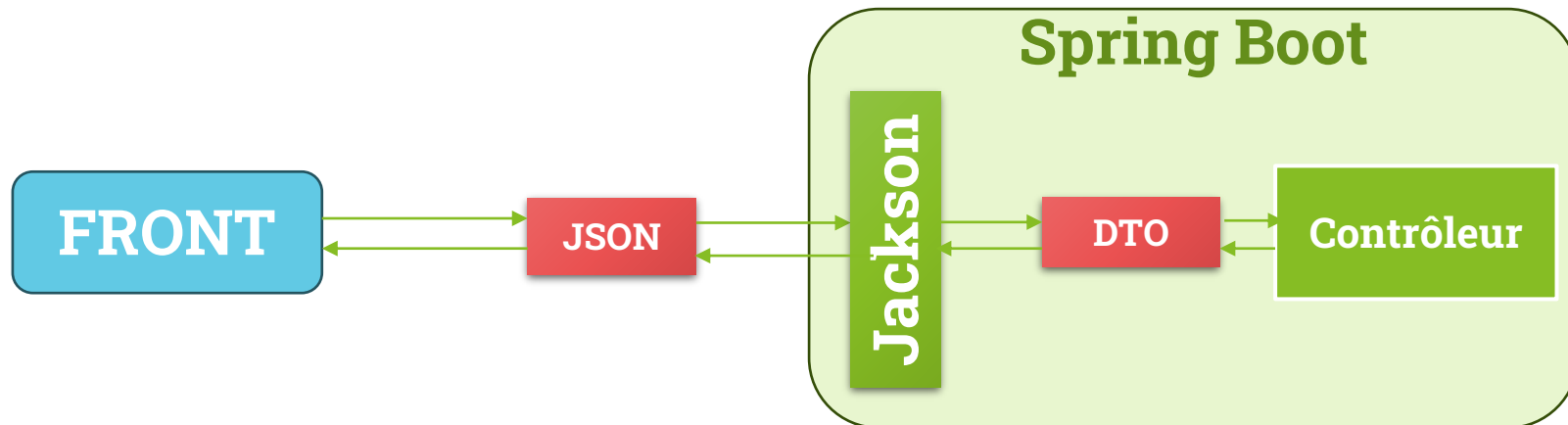
# La sérialisation / désérialisation entre front et back

- La sérialisation et la désérialisation est assurée par Jackson
  - Sérialisation: transformation d'un objet en message JSON
  - Désérialisation : transformation d'un message JSON en objet
- Quel type d'objets ?
  - Des instances d'entités JPA ?
  - Le problème des entités JPA :
    - Les boucles infinies au niveau de Jackson à cause des cycles OneToMany/ManyToOne
    - Côté vue je peux avoir besoin de données formatées d'une certaine manière (ex: date) ou transformées (ex: date => age, ou, nom et prénom => identité avec nom en majuscules).



# Le concept de DTO

- Le concept de DTO est rendu indispensable dès lors qu'on veut:
  - Séparer le métier de sa représentation (formatage et transformation de données)
  - Eviter les boucles infinies dans la couche de transformation en JSON.
- Les entités JPA ne doivent pas être utilisées comme objets de transits :
  - Une méthode de contrôleur ne doit pas prendre en paramètre une entité JPA
  - Une méthode de contrôleur ne doit pas renvoyer une liste d'entités JPA



# Pattern MVC et bonnes pratiques (1/2)

- Mauvaise pratique : ajoutez des **attributs** dans l'entité métier car on en a besoin dans la vue
- Imaginons le besoin suivant :

```
@Entity
public class Client {
    @Id
    private int id;
    private String nom;
    private String prenom;
    private LocalDate dateNaissance;
}
```

```
@Entity
public class Client {
    @Id
    private int id;
    private String nom;
    private String prenom;
    private LocalDate dateNaissance;

    @Transient
    private String identite;
    @Transient
    private String dateFormatee;
    @Transient
    private int age;
}
```

**Pas  
bien**

- Besoins en HTML :

Identité	Date de naissance	Age
Jean MARTIN	12 avril 1990	29
Ian LIU	11 août 1982	36

# Pattern MVC et bonnes pratiques (2/2)

- La classe métier représente le métier tel qu'exprimé par le client dans le cahier des charges.
- Si la vue a des besoins, on crée une classe qui couvre **les besoins de la vue**
- Une classe métier peut avoir **n** classes type "Dto" en fonction des besoins de différentes vues.

```
@Entity
public class Client {
    @Id
    private int id;
    private String nom;
    private String prenom;
    private LocalDate dateNaissance;
}
```



```
public class ClientDto {
    private String identite;
    private String dateNaissance;
    private int age;
}
```

**Bien**

- Appellations :
  - Classe de type "Dto" pour Data Transfer Object
  - On les appelle aussi parfois View, comme ClientView par exemple.

# La transformation d'une classe métier en DTO

- ❑ Cette étape de transformation n'est pas considérée comme du métier
- ❑ Cela peut se faire dans le contrôleur avec une classe type mapper.
- ❑ **Exemple : ClientMapper a la responsabilité de transformer des instances de Client en ClientDto et vice versa.**

```
@Component
public class ClientMapper {

    public ClientDto toDto(Client client){
        ClientDto dto = new ClientDto();
        dto.setIdentite(client.getPrenom()+" "+client.getNom().toUpperCase());
        dto.setAge(DateUtils.calculerAge(client.getDateNaissance()));
        // etc.
        return dto;
    }

    public Client toBean(ClientDto client){
        Client bean = new Client();
        // etc.
        return bean;
    }
}
```



## TP n°8: Mise en place des DTO



## Gestion des exceptions avec ExceptionHandler

# La gestion des exceptions

- Vous pouvez générer des exceptions côté serveur afin de renvoyer directement un message d'erreur au front associé à un code erreur HTTP.
- Pour cela il faut créer un **ExceptionHandler** qui va être chargé de capter les exceptions renvoyées par le contrôleur et de transmettre les messages au client.
- Exemple qui capte toutes les exceptions de type **FunctionalException** et renvoie au client une réponse avec le message d'erreur:

```
@ExceptionHandler({FunctionalException.class})  
public ResponseEntity<String> traiterErreurs(FunctionalException e) {  
  
    return ResponseEntity.badRequest().body(e.getMessage());  
}
```

- Attention, un **ExceptionHandler** est lié à un contrôleur.

# ExceptionHandler pour plusieurs contrôleurs

- Vous pouvez créer un **contrôleur parent** : AbstractControleur
- Ce contrôleur parent contient l'ExceptionHandler
- Tous vos contrôleurs doivent hériter de AbstractControleur

```
@ExceptionHandler({FunctionalException.class})  
public ResponseEntity<String> traiterErreurs(FunctionalException e) {  
  
    return ResponseEntity.badRequest().body(e.getMessage());  
}
```

# Autre solution: ControllerAdvice

- L'annotation **@ControllerAdvice** permet de définir un **exception handler** attaché non pas à un contrôleur mais à l'ensemble de l'application
- Il faut créer une classe indépendante portant l'annotation **@ControllerAdvice** et qui va posséder la méthode précédente :

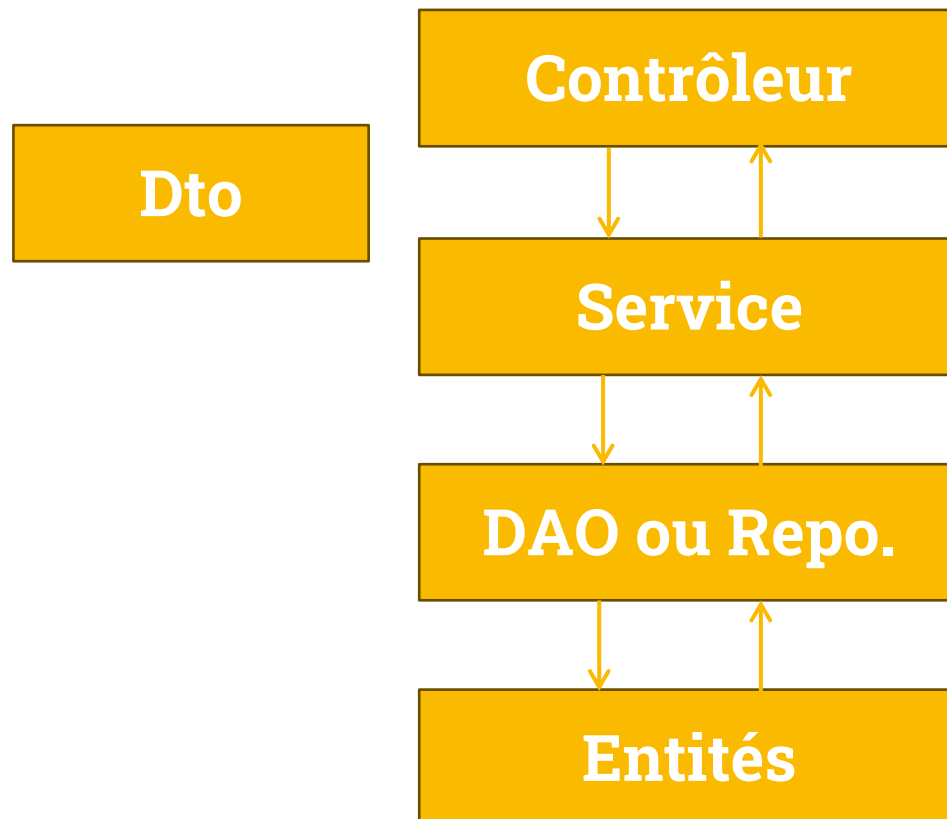
```
@ControllerAdvice  
public class RestResponseEntityExceptionHandler {  
  
    @ExceptionHandler({ FunctionalException.class})  
    protected ResponseEntity<String> traiterErreurs(FunctionalException ex) {  
        return ResponseEntity.badRequest().body(ex.getMessage());  
    }  
}
```

## **TP n°9: Mise en place d'un gestionnaire d'exceptions**

## Découpage en couches

# Le découpage en couches

- Le **découpage en couches** est un **aspect stratégique** de votre **conception**.
- Vous devez affecter les responsabilités au mieux au sein de votre application.
- Exemple de découpage basique :



*Interception des requêtes et renvoi des réponses*  
*Le contrôleur n'est que le point d'entrée/sortie. Il ne contient pas de code excepté l'appel de la couche de service*

*Mise en place du code du "Cas d'Utilisation"*  
*Mise en place des contrôles métier*

*Accès à la base de données*

*Entités JPA*



# Contrôleur avant

- Exemple de contrôleur qui ne **respecte pas les bonnes pratiques**

```
@Transactional
@PostMapping
public List<VilleDto> insertVille(@RequestBody Ville nvVille) throws FunctionalException {
    if (nvVille.getNom()==null || nvVille.getNom().isEmpty()){
        throw new FunctionalException("Le nom de la ville est obligatoire");
    }
    if (nvVille.getNbHabitants() <=0){
        throw new FunctionalException("La population de la ville doit être renseignée.");
    }
    if (nvVille.getId()!=0){
        throw new FunctionalException("L'identifiant de la ville ne doit pas être renseigné.");
    }
    villeDao.insert(nvVille);
    List<Ville> villes = villeDao.extractAll();
    return villeMapper.toDtos(villes);
}
```

- Ce n'est pas de la responsabilité du contrôleur de faire autant de choses.

# Contrôleur après

- Exemple de contrôleur qui **respecte les bonnes pratiques**

**@PostMapping**

```
public Iterable<Ville> insertVille(@RequestBody Ville nvVille) throws FunctionalException {  
    villeService.insertVille(nvVille);  
    return villeService.extraireToutes();  
}
```

- Exemple de méthode insertVille dans VilleService :

**@Transactional**

```
public void insertVille(@RequestBody Ville nvVille) throws FunctionalException {  
    if (nvVille.getNom()==null || nvVille.getNom().isEmpty()){  
        throw new FunctionalException("Le nom de la ville est obligatoire");  
    }  
    if (nvVille.getNbHabitants() <=0){  
        throw new FunctionalException("La population de la ville doit être renseignée.");  
    }  
    if (nvVille.getId()!=0){  
        throw new FunctionalException("L'identifiant de la ville ne doit pas être renseigné.");  
    }  
    villeRepository.save(nvVille);  
}
```



## Documenter votre API avec Swagger

# Qu'est-ce que Swagger ?

- Un outil qui publie automatiquement la documentation de votre API
- Se nomme désormais **OpenAPI 3.0**

The screenshot shows the Swagger UI interface for an API named 'API Recensement'. The top bar features the Swagger logo and a search bar containing '/v3/api-docs'. Below the title, there are links for 'Terms of service', 'Nom du contact - Website', 'Send email to Nom du contact', and 'Nom de la licence'. A 'Servers' dropdown menu is set to 'http://localhost:8080 - Generated server url'. The main section, titled 'ville-controller', lists three endpoints for the '/villes' path: a GET method (blue bar), a PUT method (orange bar), and a POST method (green bar). Each endpoint has a dropdown arrow on its right side.

# Mettre en place Swagger

- Dans votre **pom.xml**

```
<dependency>  
  <groupId>org.springdoc</groupId>  
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>  
  <version>2.5.0</version>  
</dependency>
```

# Récupérer la documentation de votre API au format JSON

- Extraire la documentation de votre API au format JSON
- URL par défaut : <http://localhost:8080/v3/api-docs>
- Possibilité de modifier cette URL par défaut via une propriété dans **application.properties** :
  - **springdoc.api-docs.path=/api-docs**

➤ Exemple :

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "API Recensement",
    "description": "Cette API fournit des données de recensement de population pour la France.",
    "termsOfService": "OPEN DATA",
    "contact": {
      "name": "Nom du contact",
      "url": "URL du contact",
      "email": "email@exemple.com"
    },
    "license": {
      "name": "Nom de la licence",
      "url": "URL de la licence"
    },
    "version": "1.0"
  },
  "servers": [
    {
      "url": "http://localhost:8080",
      "description": "Generated server url"
    }
  ],
  "paths": {
    "/villes": {
      "get": {
        "tags": [
          "ville-controller"
        ],
        "operationId": "extraireVilles",
        "responses": {
          "200": {
            "description": "OK",
            "content": {
              "*/*": {
                "schema": {
                  "type": "array",
                  "items": {
                    "$ref": "#/components/schemas/VilleDto"
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

# Récupérer la documentation de votre API au format JSON

- Extraire la documentation de votre API au format HTML
- URL par défaut : <http://localhost:8080/swagger-ui/index.html>

The screenshot displays the Swagger UI interface. At the top, the Swagger logo is visible, followed by a search bar containing '/v3/api-docs' and an 'Explore' button. Below this, a box highlights the 'API Recensement' section, which includes version information (1.0, OAS 3.0), a description ('Cette API fournit des données de recensement de population pour la France.'), and links for 'Terms of service', 'Nom du contact - Website', 'Send email to Nom du contact', and 'Nom de la licence'. An orange arrow points from the text 'D'où viennent ces informations ?' to this highlighted box. Below the API description, a 'Servers' dropdown menu shows 'http://localhost:8080 - Generated server url'. At the bottom, a section titled 'ville-controller' lists three endpoints: 'GET /villes', 'PUT /villes', and 'POST /villes', each with a corresponding colored button and a dropdown arrow.

**API Recensement** 1.0 OAS 3.0  
/v3/api-docs

Cette API fournit des données de recensement de population pour la France.

[Terms of service](#)  
[Nom du contact - Website](#)  
[Send email to Nom du contact](#)  
[Nom de la licence](#)

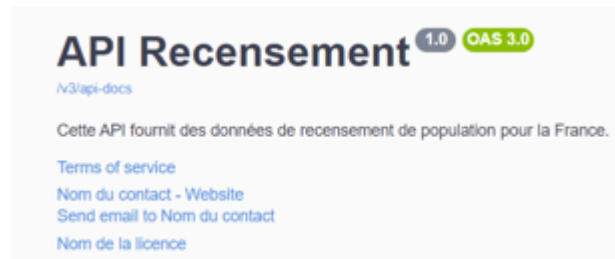
**D'où viennent ces informations ?**

**Servers**  
http://localhost:8080 - Generated server url

**ville-controller**

- GET /villes
- PUT /villes
- POST /villes

# Modifier les données d'introduction à l'API



- Pour faire apparaître ces données il faut injecter un **bean** spécial dans le **container IoC** grâce à une méthode annotée avec **@Bean**.
- Exemple :

```
@Configuration
public class SwaggerConfig {

    @Bean
    public OpenAPI customOpenAPI() {
        return new OpenAPI()
            .info(new Info()
                .title("API Recensement")
                .version("1.0")
                .description("Cette API fournit des données de recensement de population pour la France.")
                .termsOfService("OPEN DATA")
                .contact(new Contact().name("Nom du contact").email("email@exemple.com").url("URL du contact")))
            .license(new License().name("Nom de la licence").url("URL de la licence")));
    }
}
```



# Documenter les différentes méthodes

- Grâce à un jeu d'annotations à positionner sur les points d'entrée (méthodes mappées sur des routes) de vos contrôleurs, vous pouvez documenter votre API.
- Exemple :

```
/**
 * GET /villes : permet d'extraire la liste de toutes les villes présentes dans
 * le système
 *
 * @return
 */
@Operation(summary = "Extraire la liste des villes")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
        description = "Liste des villes au format JSON",
        content = { @Content(mediaType = "application/json", schema = @Schema(implementation = VilleDto.class)) }
    )
})
@GetMapping
public ResponseEntity<List<VilleDto>> extraireVilles() {
    return ResponseEntity.ok(villeService.findAll());
}
```

# Documenter les différentes méthodes

➤ Vous pouvez également documenter les **erreurs**

➤ Exemple :

```
/**
 * GET villes/nom/{nom} : permet d'extraire toutes les villes dont le nom commence
 * par nom
 *
 * @param debutNom début du nom des villes recherchées
 * @return ResponseEntity avec un statut 200 si ok ou 400 si une règle métier n'a pas été respectée
 */
@Operation(summary = "Extraire la liste des villes dont le nom commence par la chaîne de caractères {debutNom}")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
        description = "Liste des villes dont le nom commence par {debutNom}",
        content = { @Content(mediaType = "application/json",
            schema = @Schema(implementation = VilleDto.class)) },
    @ApiResponse(responseCode = "400", description = "Si debutNom ne contient pas au moins 1 caractère",
        content = @Content)})
@GetMapping("/nom/{debutNom}")
public ResponseEntity<List<VilleDto>> extraireVilleParNom(@PathVariable String debutNom) throws AnomalieException {
    return ResponseEntity.ok(villeService.findByNomStartsWith(debutNom));
}
```

**TP non numéroté: Mettre en place la documentation Swagger de votre API en suivant les instructions de ce cours.**

## Interfaces et faible couplage

# Couplage fort vs couplage faible

- Le **couplage fort** c'est quand une classe a un attribut d'instance qui est une autre classe.
- Exemple de **couplage fort**

```
@RestController
@RequestMapping("/store")
public class StoreControleur {

    @Autowired
    private PetDao dao;

    @GetMapping(path = "/inventory")
    public List<Pet> getPets(){

        List<Pet> pets = dao.extractPets();
        return pets;
    }
}
```

```
@Component
public class PetDao {

    @PersistenceContext
    private EntityManager em;

    public List<Pet> extractPets(){
        TypedQuery<Pet> query = em.createQuery("SELECT p
        FROM Pet p", Pet.class);
        return query.getResultList();
    }
}
```

# Couplage fort vs couplage faible

- Le **couplage faible** c'est quand une classe a un attribut d'instance qui est une **interface**.
- Exemple de couplage faible

```
@RestController
@RequestMapping("/store")
public class StoreControleur {

    @Autowired
    private PetDao dao;

    @GetMapping(path = "/inventory")
    public List<Pet> getPets(){

        List<Pet> pets = dao.extractPets();
        return pets;
    }
}
```

```
@Component
public class PetDaoJpa implements PetDao {

    @PersistenceContext
    private EntityManager em;

    @Override
    public List<Pet> extractPets(){
        TypedQuery<Pet> query = em.createQuery("SELECT p
FROM Pet p", Pet.class);
        return query.getResultList();
    }
}
```

```
public interface PetDao {

    List<Pet> extractPets();
}
```

# Avantages du couplage faible (loose coupling )

- L'interface agit comme un contrat qu'une classe doit respecter
- Il est facile de remplacer une implémentation (i.e. classe qui implémente l'interface) par une autre si ces 2 implémentations respectent le même contrat : celui de l'interface
- Les applications sont plus modulaires.
- Les applications sont plus faciles à faire évoluer.

# Fonctionnement de l'Autowired avec une interface

- L'interface n'est pas instanciable donc elle ne peut pas devenir un bean Spring.
- La classe qui implémente l'interface est un bean Spring. C'est elle qui porte @Component ou @Service
- Lorsque Spring rencontre un @Autowired sur une interface, il recherche dans le conteneur IoC un bean qui implémente cette interface.
- Attention, si plusieurs beans implémentent la même interface, il y a ambiguïté et @Autowired ne peut pas être utilisé.



# Utilisation de @Qualifier

- Lorsque plusieurs beans implémentent la même interface, il y a ambiguïté et **@Autowired** ne peut **pas être utilisée seule**.
- **L'application ne démarre pas en l'état !**

```
@RestController
@RequestMapping("/villes")
public class VilleControleur {

    @Autowired
    private VilleService service;

    @GetMapping
    public List< Ville > getVilles(){

        List<Ville> villes = service.extraireToutes();
        return pets;
    }
}
```

```
public interface VilleService {

    Iterable<Ville> extraireToutes();

    void insertVille(Ville nvVille) throws AnomalieException;

    void modifyVille(int id, Ville nvDonnees);
}
```

```
@Service
public class VilleServiceImpl1 implements VilleService {

}
```

```
@Service
public class VilleServiceImpl2 implements VilleService {

}
```

# Utilisation de @Qualifier

- Dans ce cas chaque bean doit être nommé et l'annotation @Autowired doit être complétée avec @Qualifier("nomBean")

```
@RestController
@RequestMapping("/villes")
public class VilleControleur {

    @Autowired
    @Qualifier("service1")
    private VilleService service;

    @GetMapping
    public List< Ville > getVilles(){

        List<Ville> villes = service.extraireToutes();
        return pets;
    }
}
```

```
public interface VilleService {

    Iterable<Ville> extraireToutes();

    void insertVille(Ville nvVille) throws AnomalieException;

    void modifyVille(int id, Ville nvDonnees);
}
```

Non utilisé

```
@Service("service1")
public class VilleServiceImpl1 implements VilleService {

}
```

```
@Service("service2")
public class VilleServiceImpl2 implements VilleService {

}
```



Appeler une API  
externe

# Appel d'une API externe depuis une application Spring

- Utilisation de RestTemplate et méthode getForEntity

```
RestTemplate restTemplate = new RestTemplate();  
MaDto response = restTemplate.getForObject("http://api.data.org/...", MaDto.class);
```

- **Attention votre classe MaDto doit avoir la même structure que le JSON.**
- Comment voir à quoi ressemble le JSON associé ?


```
RestTemplate restTemplate = new RestTemplate();  
ResponseEntity<String> response = restTemplate.getForEntity("http://api.data.org/...",  
String.class);
```

# Parser un fichier JSON avec Jackson

- ❑ Mapping implicite basé sur les noms des attributs.
- ❑ Annotations Jackson disponibles si les noms des attributs de classes et des clés JSON sont différents.
- ❑ Exemple pour mapper l'objet JSON **représentant un livre** avec la classe **Livre**

```
{ "titre": "Germinal",  
  "auteur": {"nom": "Zola", "prenom": "Emile"},  
  "ean": 9782246465614,  
  "collection": "Les Cahiers Rouges",  
  "editeur": "Grasset"  
}
```

```
public class Livre {  
  
    private String titre;  
    private Auteur auteur;  
    private BigInteger ean;  
    private String collection;  
    private String editeur;  
}
```



# Parser un fichier JSON avec Java

- ❑ Utilisation de **Jackson** pour **lire le fichier JSON**
- ❑ Le terme de **désérialisation** est également utilisé dans le vocabulaire Jackson.
- ❑ Code beaucoup plus simple qu'avec JAXB


```
ObjectMapper mapper = new ObjectMapper();  
Livre l = mapper.readValue(response.getBody(), Livre.class);
```

# Cas particulier: noms différents

- ❑ Supposons par exemple qu'une clé dans le fichier JSON ne respecte pas les conventions de nommage Java
- ❑ Dans ce cas il faut utiliser l'annotation **@JsonProperty** avec le nom de la clé JSON entre parenthèses.
- ❑ Vous pouvez utiliser également **@JsonGetter** sur la méthode getter

```
{ "titre-livre": "Germinal",  
  "auteur": {"nom": "Zola", "prenom": "Emile"},  
  "ean": 9782246465614,  
  "collection": "Les Cahiers Rouges",  
  "editeur": "Grasset"  
}
```

```
public class Livre {  
    @JsonProperty("titre-livre")  
    protected String titre;  
    protected Auteur auteur;  
    protected BigInteger ean;  
}
```



# Cas particulier: les attributs de type date

- ❑ Supposons que dans le fichier JSON vous ayez une date et que vous souhaitiez la parser en `LocalDate`.
- ❑ Vous allez devoir utiliser une annotation particulière `@JsonDeserialize` positionnée sur l'attribut de la classe Java.
- ❑ Cette annotation va préciser la classe que Jackson doit utiliser pour passer de `String` à `LocalDate`

```
{ "titre-livre": "Germinal",  
  "auteur": {"nom": "Zola", "prenom": "Emile"},  
  "ean": 9782246465614,  
  "collection": "Les Cahiers Rouges",  
  "editeur": "Grasset",  
  "dateParution": "12/05/2004"  
}
```

```
public class Livre {  
  
    protected String titre;  
    protected Auteur auteur;  
  
    @JsonDeserialize(using = LocalDateDeserializer.class)  
    protected LocalDate dateParution;  
}
```



# Exemple de Deserializer

- ❑ Cette classe doit hériter de la classe mère **StdDeserializer**<LocalDate> pour une **LocalDate**
- ❑ La méthode obligatoire à implémenter est **deserialize** avec 2 paramètres.
- ❑ **JsonParser** fournit la valeur issue dun fichier JSON au format désiré
- ❑ **DeserializationContext** sert s'il est nécessaire de récupérer des valeurs précédentes par ex.

```
public class LocalDateDeserializer extends StdDeserializer<LocalDate> {  
  
    protected LocalDateDeserializer() {  
        super(LocalDate.class);  
    }  
  
    @Override  
    public LocalDate deserialize(JsonParser parser, DeserializationContext context) throws IOException {  
        String dateAuFormatString = parser.readValueAs(String.class);  
        return LocalDate.parse(dateAuFormatString, DateTimeFormatter.ofPattern("dd/MM/yyyy"));  
    }  
}
```

# Cas particulier: lecture d'un tableau d'objets

- ❑ Supposons que dans le fichier JSON vous ayez une collection d'objets JSON

```
[{ "nom": "Paul",  
  "prenom": "Jean",  
},  
{ "nom": "Mex",  
  "prenom": "Tex",  
},  
]
```

- ❑ Il existe plusieurs moyens d'extraire ces valeurs
- ❑ Tout d'abord sous forme de tableau:

```
ObjectMapper mapper = new ObjectMapper();  
Person[] persons = mapper.readValue(new File("C:/Temp/persons.json"), Person[].class);
```

# Cas particulier: lecture d'un tableau d'objets


```
[{ "nom": "Paul",  
  "prenom": "Jean",  
},  
{ "nom": "Mex",  
  "prenom": "Tex",  
},  
]
```

- ❑ Pour une **List**, on ne peut pas utiliser **List.class** en second paramètre.
- ❑ Il faut utiliser une méthode spécifique, fournie par le mapper, qui va **calculer** le type d'une List de Person:

```
ObjectMapper mapper = new ObjectMapper();  
CollectionType collType = mapper.getTypeFactory().constructCollectionType(List.class, Person.class);  
List<Person> persons = mapper.readValue(new File("C:/Temp/persons.json"), collType);
```

# Cas particulier: les propriétés inconnues

- ❑ Ajouter la propriété suivante sur votre classe principale:



```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Livre {

    @JsonProperty(value = "titre-livre")
    protected String titre;
    protected Auteur auteur;
    protected String dateParution;
```



Renvoyer un  
fichier binaire

# Le back renvoie un fichier PDF dans la réponse HTTP

- Etape 1: injecter la réponse HTTP en paramètre de la méthode

```
@GetMapping("/{nom}/fiche")  
public void ficheVille(@PathVariable String nom, HttpServletResponse response)  
throws Exception {  
}
```

- Etape 2: déclarer l'attachement d'un fichier et donner un nom à ce fichier

```
@GetMapping("/{nom}/fiche")  
public void ficheVille(@PathVariable String nom, HttpServletResponse response) throws IOException,  
DocumentException, FunctionalException {  
  
    response.setHeader("Content-Disposition", "attachment; filename=\"fichier.pdf\"");  
  
}
```

# Le back renvoie un fichier PDF dans la réponse HTTP avec la librairie iText

- Etape 3: Créer le document PDF et le writer avec **iText**

```
@GetMapping("/{nom}/fiche")
public void ficheVille(@PathVariable String nom, HttpServletResponse response) throws IOException,
DocumentException, FunctionalException {

    response.setHeader("Content-Disposition", "attachment; filename=\"fichier.pdf\"");

    Document document = new Document(PageSize.A4);
    PdfWriter.getInstance(document, response.getOutputStream());

}
```

- C'est le PdfWriter qui permet d'écrire les données binaires dans la réponse HTTP.

# Le back renvoie un fichier PDF dans la réponse HTTP

## ➤ Etape 4: l'étape de création

```
@GetMapping("/{nom}/fiche")
public void ficheVille(@PathVariable String nom, HttpServletResponse response) throws IOException,
DocumentException, FunctionalException {

    response.setHeader("Content-Disposition", "attachment; filename=\"fichier.pdf\"");

    Document document = new Document(PageSize.A4);
    PdfWriter.getInstance(document, response.getOutputStream());

    document.open();
    document.addTitle("Fiche");
    document.newPage();
    BaseFont baseFont = BaseFont.createFont(BaseFont.HELVETICA, BaseFont.WINANSI, BaseFont.EMBEDDED);
    Phrase p1 = new Phrase("COUCOU", new Font(baseFont, 32.0f, 1, new BaseColor(0, 51, 80)));
    document.add(p1);
    document.close();
}
```

## ➤ Ne pas oublier open() et close()



# Le back renvoie un fichier PDF dans la réponse HTTP

- Etape 5: on pousse les données

```
@GetMapping("/{nom}/fiche")
public void ficheVille(@PathVariable String nom, HttpServletResponse response) throws IOException,
DocumentException, FunctionalException {

    response.setHeader("Content-Disposition", "attachment; filename=\"fichier.pdf\"");

    Document document = new Document(PageSize.A4);
    PdfWriter.getInstance(document, response.getOutputStream());

    document.open();
    document.addTitle("Fiche");
    document.newPage();
    BaseFont baseFont = BaseFont.createFont(BaseFont.HELVETICA, BaseFont.WINANSI, BaseFont.EMBEDDED);
    Phrase p1 = new Phrase("COUCOU", new Font(baseFont, 32.0f, 1, new BaseColor(0, 51, 80)));
    document.add(p1);
    document.close();

    response.flushBuffer();
}
```

**TP n°10: Mettre en place une méthode d'export qui renvoie un fichier CSV et une seconde qui renvoie un fichier PDF.**



## Tester avec Spring Test et MockMvc

# Spring Test - Présentation

- Spring Test est un module permettant de démarrer l'environnement applicatif pour réaliser des tests unitaires.
- Spring Test est complémentaire à Junit
- Les dépendances à ajouter dans le pom.xml

```
<<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-test</artifactId>  
  <scope>test</scope>  
</dependency>  
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-engine</artifactId>  
  <scope>test</scope>  
</dependency>  
<dependency>  
  <groupId>org.junit.jupiter</groupId>  
  <artifactId>junit-jupiter-api</artifactId>  
  <scope>test</scope>  
</dependency>
```

# Spring Test - Présentation

- Une classe de tests unitaires réalisées avec Spring Test est annotée avec **@SpringBootTest**
- **Spring Test initialise tous les modules** de manière à ce que le test bénéficie du même environnement que l'application.
- Avec cette annotation la classe de tests unitaires devient un bean Spring
- Ce faisant, il est possible d'**injecter des beans Spring** dans cette classe

# Spring Test - Exemple

## ➤ Exemple :

```
@SpringBootTest
class VilleServiceImplTest {

    @Autowired
    private VilleService villeService;

    @Test
    void extraireToutes() {
        Iterable<Ville> villes = villeService.extraireToutes();
        assertTrue(villes.iterator().hasNext());
    }
}
```

- **Problème** : que faire si le test s'exécute sur une plateforme qui n'a pas d'accès à la base de données ?

# Spring Test – Mettre en place une configuration de tests

- Il est possible d'associer une configuration de tests à **Spring Test**
- L'idée est de créer un fichier **application-*test*.properties**
- Ce fichier doit être placé dans le répertoire **src/test/resources**
- Ensuite au niveau du test unitaire, ajoutez l'annotation :

**@ActiveProfiles("test")**

- Vous pouvez mettre un autre nom que test.

# Spring Test – Mettre en place une base mémoire h2

- Dans le fichier **application-*test*.properties**, on configure une base de test

```
spring.datasource.url=jdbc:h2:mem:test;MODE=MySQL;  
spring.datasource.driver-class-name=org.h2.Driver  
spring.datasource.username=  
spring.datasource.password=  
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect  
spring.jpa.hibernate.ddl-auto=update
```

- Lors du démarrage du test unitaire, la base **h2** sera utilisée.



# Spring Test – Exemple

## ➤ Exemple :

```
@SpringBootTest
@ActiveProfiles("test")
class VilleServiceImplTest {

    @Autowired
    private VilleService villeService;

    @Test
    void extraireToutes() {
        Iterable<Ville> villes = villeService.extraireToutes();
        assertTrue(villes.iterator().hasNext());
    }
}
```

- Evidemment la base **h2** est vide mais il est possible de l'initialiser avec des données

# Spring Test – Mettre en place un fichier d'initialisation de h2

- On ajoute un fichier **data.sql** dans **src/test/resources**
- Le problème : le fichier d'initialisation est exécuté avant que les tables ne soient générées par JPA.
- Il faut donc ajouter une propriété dans **application-*test*.properties**, pour passer le script **data.sql** après l'initialisation des tables par JPA.

```
spring.jpa.defer-datasource-initialization=true
```

# Spring Test – Mocker les accès base avec @MockBean et Mockito

- Normalement un test unitaire doit être indépendant de ressources externes.
- On peut avoir besoin de "mock" (simuler, maquetter) la partie repository par exemple.
- Pour demander à Spring Test de "mock" un repository, il faut injecter le Repository dans la classe de tests unitaire et ajouter l'annotation @MockBean dessus

# Spring Test – Exemple de mock

## ➤ Exemple :

```
@SpringBootTest
@ActiveProfiles("test")
class VilleServiceImplTest {

    @Autowired
    private VilleService villeService;

    @MockBean
    private VilleRepository villeRepository;

    @Test
    void extraireToutes() {
        Iterable<Ville> villes = villeService.extraireToutes();
        assertTrue(villes.iterator().hasNext());
    }
}
```

- **Problème** : mon assertion comme quoi j'ai des villes en base ne fonctionne plus.

# Spring Test – Imposer un comportement au mock avec Mockito

- **Mockito** permet de donner du comportement à un mock.
- **Exemple:**

```
Mockito.when(villeRepository.findAll()).thenReturn(List.of(new Ville("Angers", 142000)));
```

- Avec **Mockito** j'impose de retourner une liste de Ville ne contenant qu'une ville lorsque la méthode findAll() est appelée.
- C'est très puissant mais rend l'écriture des tests assez longues.

# Spring Test – Exemple avec Mockito

## ➤ Exemple:

```
@SpringBootTest
@ActiveProfiles("test")
class VilleServiceImplTest {

    @Autowired
    private VilleService villeService;

    @MockBean
    private VilleRepository villeRepository;

    @Test
    void extraireToutes() {

        Mockito.when(villeRepository.findAll()).thenReturn(List.of(new Ville("Angers", 142000)));

        Iterable<Ville> villes = villeService.extraireToutes();
        assertTrue(villes.iterator().hasNext());
    }
}
```

# Spring Test – Tester un contrôleur avec MockMvc

- Il est possible de tester l'API avec **MockMvc**.
- C'est une API très riche qui est intégrée à **Spring Test**.
- Pour utiliser **MockMvc** dans un test unitaire il faut ajouter l'annotation **@AutoConfigureMockMvc** et injecter une instance de **MockMvc** dans la classe de tests

# Spring Test – Exemple complet avec MockMvc : les attributs

## ➤ Exemple:

```
@SpringBootTest
@ActiveProfiles("test")
@AutoConfigureMockMvc
class VilleControleurTest {

    @Autowired
    private ObjectMapper objectMapper;
    @MockBean
    private VilleRepository villeRepository;
    @MockBean
    private DepartementRepository departementRepository;
    @Autowired
    private MockMvc mockMvc;
}
```



# Spring Test – Exemple complet avec MockMvc : test de la méthode get

## ➤ Exemple:

```
@Test
void getAllVilles() throws Exception {

    Ville v = new Ville("Angers", 142000);
    v.setDepartement(new Departement("49", "Maine-et-Loire"));

    when(villeRepository.findAll()).thenReturn(List.of(v));

    this.mockMvc.perform(MockMvcRequestBuilders.get("/villes")).andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().string(containsString("Angers")))
        .andExpect(jsonPath("$.nom", is("Angers")))
        .andExpect(jsonPath("$.departement.code", is("49")));
}
```

# Spring Test – Exemple complet avec MockMvc : test de la méthode post KO

## ➤ Exemple:

```
@Test
void insertVilleKo() throws Exception {

    this.mockMvc.perform(MockMvcRequestBuilders.post("/villes")
        .content(objectMapper.writeValueAsString(new VilleDto()))
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isBadRequest());
}
```

# Spring Test – Exemple complet avec MockMvc : test de la méthode post OK

## ➤ Exemple:

```
@Test
void insertVilleOk() throws Exception {

    Departement departement = new Departement("49", "Maine-et-Loire");
    Ville nvVille = new Ville("Angers", 142000, departement);

    when(departementRepository.findByCode(anyString())).thenReturn(departement);
    this.mockMvc.perform(MockMvcRequestBuilders.post("/villes")
        .content(objectMapper.writeValueAsString(new VilleDto(nvVille)))
        .contentType(MediaType.APPLICATION_JSON)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk());
}
```

## **TP n°11: Mettre en place des tests unitaires**



## ANNEXE 1 :

Faire une  
application Spring  
Boot sans partie  
web

# 2 types d'applications au sein d'un même projet

➤ **Il est possible de faire coexister au sein d'un même projet :**

- une application qui lance un serveur web
- une application qui va par exemple réaliser un traitement de fichiers et qui n'a donc pas besoin d'un serveur web.

➤ **Avantages :**

- Ces 2 applications peuvent utiliser les mêmes services, DAO, repositories et entités JPA.

# CommandLineRunner

- **Pour créer une application qui traite par exemple un fichier :**
  - Créer une application Spring Boot qui implémente l'interface CommandLineRunner
  - Redéfinir la méthode **void run(String[] args)**.
- **Exemple :**

```
@SpringBootApplication
public class TraitementFichiersApplication implements CommandLineRunner {

    public static void main(String[] args) {

        SpringApplication.run(TraitementFichiersApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
    }
}
```

# Bloquer le démarrage de Tomcat

- Le problème est que par défaut, vous allez démarrer Tomcat sur le port 8080
- Il faut bloquer le démarrage de Tomcat en modifiant comme suit la méthode main :

```
@SpringBootApplication
public class TraitementFichiersApplication implements CommandLineRunner {

    public static void main(String[] args) {

        SpringApplication application = new SpringApplication(TraitementFichiersApplication.class);
        application.setWebApplicationType(WebApplicationType.NONE);
        application.run(args);
    }

    @Override
    public void run(String... args) throws Exception {
    }
}
```



# Injection de vos beans Spring

- Vous pouvez alors injecter vos beans Spring directement dans la classe :

```
@SpringBootApplication
public class TraitementFichiersApplication implements CommandLineRunner {

    @Autowired
    private VilleService villeService;

    public static void main(String[] args) {

        SpringApplication application = new SpringApplication(TraitementFichiersApplication.class);
        application.setWebApplicationType(WebApplicationType.NONE);
        application.run(args);
    }

    @Override
    public void run(String... args) throws Exception {
    }
}
```

# Désactiver l'accès à la base de données

- Il est possible de désactiver l'accès à la base de données en ajoutant l'annotation `@EnableAutoConfiguration` et en excluant les classes qui servent à configurer les accès à la base.

```
@SpringBootApplication
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class, HibernateJpaAutoConfiguration.class})
public class TraitementFichiersApplication implements CommandLineRunner {
    ...
}
```

## **TP n°12: Traitement du fichier recensement**

FIN

**MERCI DE VOTRE ATTENTION !**