

# Architecture Microservices avec Spring Boot

**Christophe Fontaine**  
[cfontaine@dawan.fr](mailto:cfontaine@dawan.fr)

16/01/2023

# Objectifs

---



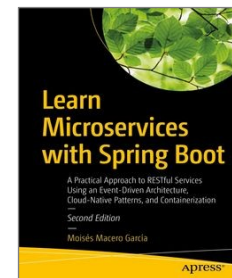
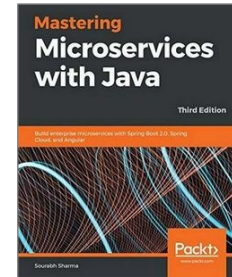
- Maîtriser l'utilisation de Spring Boot pour la construction de web services

**Durée :** 5 jours

**Pré-requis:** Maîtrise de la programmation orientée objet Java et de Spring Core

# Bibliographie

- **Mastering Microservices with Java**  
Sourabh Sharma  
Packt Publishing - Février 2019
- **Learn Microservices with Spring Boot**  
Moisés Macero García  
Apress - 2<sup>nd</sup> edition - Novembre 2020
- **Spring in Action**  
Craig Walls  
Manning - 6<sup>nd</sup> edition - Janvier 2022
- **Spring Microservices in Action**  
John Carnell, Illary Huaylupo Sánchez  
Manning - 2<sup>nd</sup> edition - Mai 2021



# Bibliographie



- **Spring Reference Documentation**

- Spring Framework

<https://docs.spring.io/spring-framework/docs/current/reference/html/>

- Spring Boot

<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>

- Spring Data JPA

<https://docs.spring.io/spring-data/jpa/docs/current/reference/html>

- Spring Security

<https://docs.spring.io/spring-security/site/docs/current/reference/html5/>

- **Maven: The Complete Reference**

<https://books.sonatype.com/mvnref-book/reference/index.html>

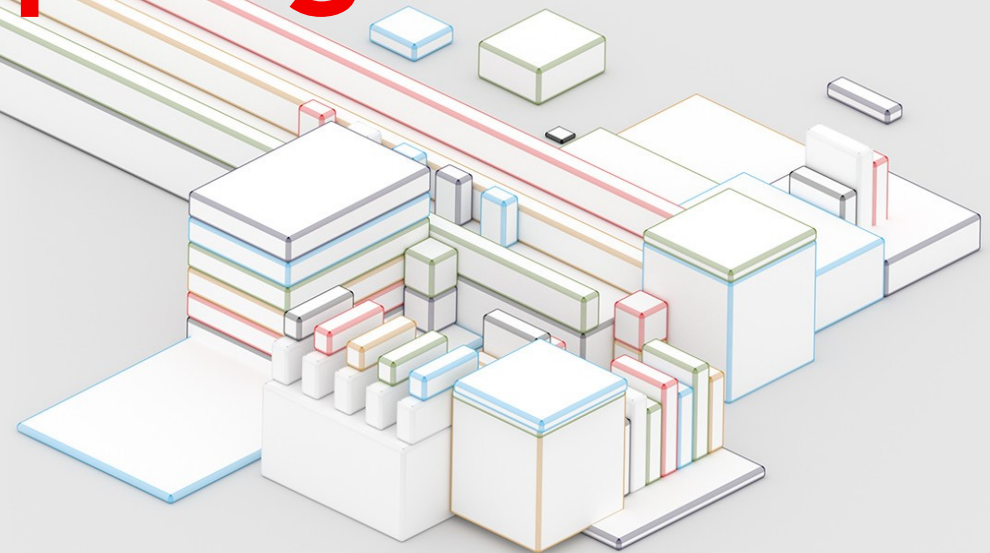
# Plan

---



- Découvrir Spring Boot
- Configurer Spring Boot
- Mapping d'entités avec Spring Data JPA
- Implémenter des requêtes sur les données de la base
- Développer des micro-services avec Spring Web
- Sécuriser un service web
- Tester une application Spring Boot
- Mettre en production un service
- Écrire des clients de services web

# Découvrir Spring Boot



# Historique

2002

**Rod Johnson** publie son livre  
**ExpertOne-on-One J2EE Design and Development**  
qui explique les raisons de la création de Spring



2004

**Spring 1.0** sort sous licence Apache 2  
Création de l'entreprise **interface21**

2006

**Spring 2.0** → Java 5, Groovy

2009

**Spring 3.0** → Java EE 6, configuration java  
Achat de SpringSource par VMWare (420 M\$)

2011

**Spring Boot 1.0** → gain en temps de développement  
configuration aisée et ajout de fonctionnalités

# Historique



**2013**

**Spring 4.0** → Java 8 , Java EE 7 et inclut Spring Boot  
Création de **Pivotal**, joint venture entre VMWare et EMC

**2017**

**Spring 5.0** → au minimum Java 8, Kotlin, Reactive programming

**2018**

**Spring Boot 2.0** → au minimum Java 8, amélioration d'actuator

**2019**

Acquisition de Pivotal Software par VMWare (2,7 milliards)

**2022**

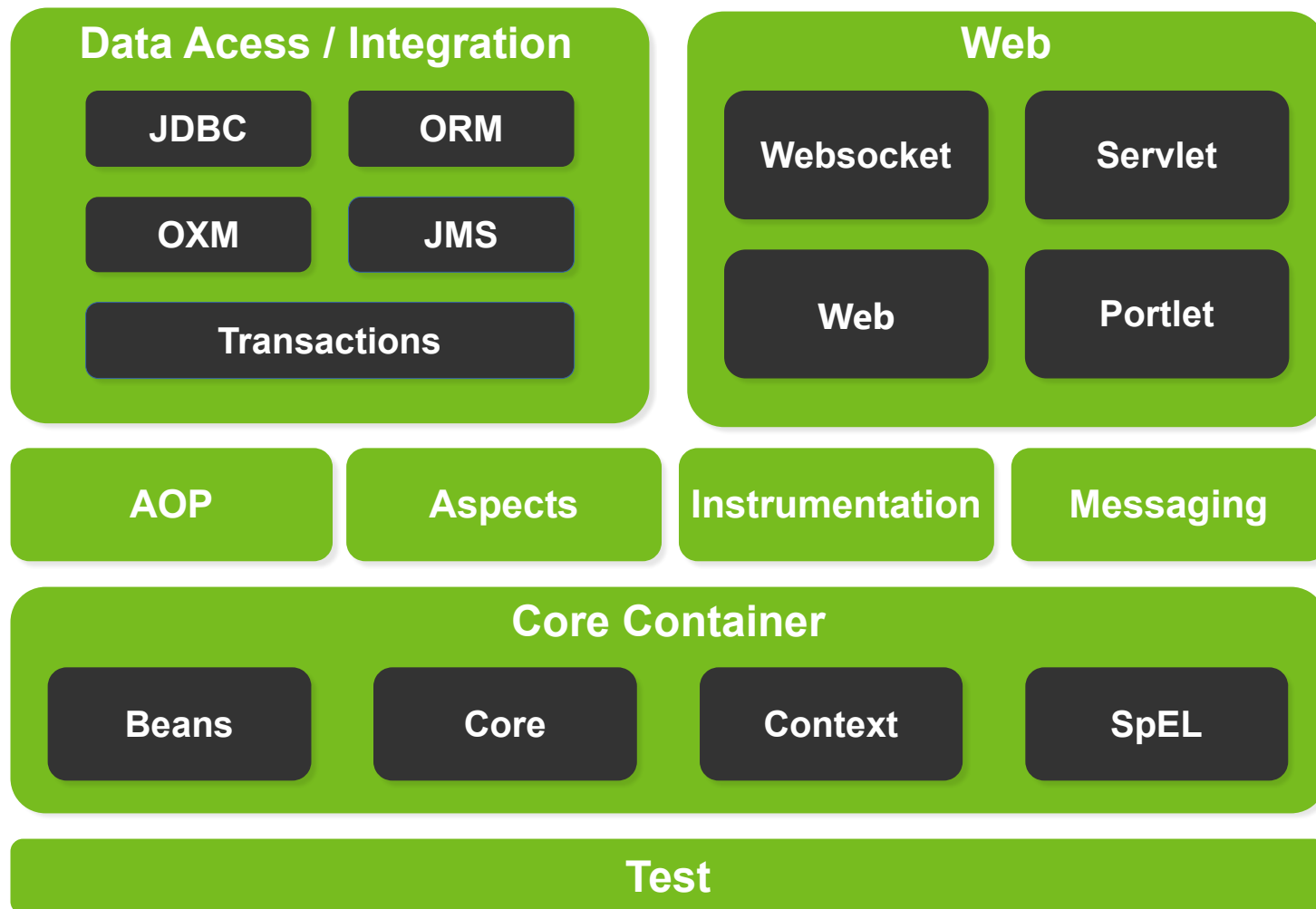
**Spring 6.0** → au minimum Java 17 , Jakarta EE 9  
**Spring Boot 3.0** compilation native, observabilité



# Structure du framework



## Spring Framework Runtime



# Galaxie Spring

Le cadre de Spring consistait à apporter un conteneur léger servant à l'loc  
Aujourd'hui, Spring représente un grand nombre de modules logiciels :



**Spring Framework** → contient les fonctionnalités de base de Spring  
(représente la version 1 de spring) **version: 6.0.4**



**Spring boot** → pour simplifier le démarrage et le développement  
de nouvelles applications Spring **version: 3.0.1**



**Spring Security** → sécurité au niveau d'une application JEE  
(authentification et habilitation des utilisateurs) **version: 6.0.1**



**Spring Data** → a pour but de faciliter l'utilisation de solutions de type  
No SQL. Il est composé de plusieurs sous-projets, un pour les  
différentes solutions supportées **version: 2022.0.1**

# Galaxie Spring



**Spring Batch** → plan de production pour l'enchaînement de traitements par lots liés par des dépendances **version: 5.0.0**



**Spring Cloud** → fournit des outils permettant aux développeurs de créer rapidement certains des modèles courants dans les systèmes distribués **version: 2022.0.0**



**Spring Web Flow** → développement d'interfaces web riches (ajax, jsf,...), utilise Spring MVC **version: 2.5.1**



**Spring Web Service** → permet de développer des services web de type SOAP **version: 4.0.1**



**Spring LDAP** → a pour but de simplifier l'utilisation d'annuaires de type LDAP **version: 2.4.0**

# Galaxie Spring



## Autre module :

 **Spring Shell (2.1.5)**

 **Spring Cloud Data Flow (2.10.0)**

 **Spring HATEOAS (2.0.0)**

 **Spring CredHub (2.3.0)**

 **Spring REST Docs (3.0.0)**

 **Spring Flo (0.8.8)**

 **Spring AMQP (3.0.0)**

 **Spring for Apache Kafka (3.0.1)**

 **Spring Integration (6.0.1)**

 **Spring Vault (3.0.0)**

 **Spring Statemachine (3.2.0)**

 **Spring GraphQL (1.1.1)**

 **Spring Session (2021.2.0)**

 **Spring Authorization Server (1.0.0)**

# Concept d'inversion de contrôle



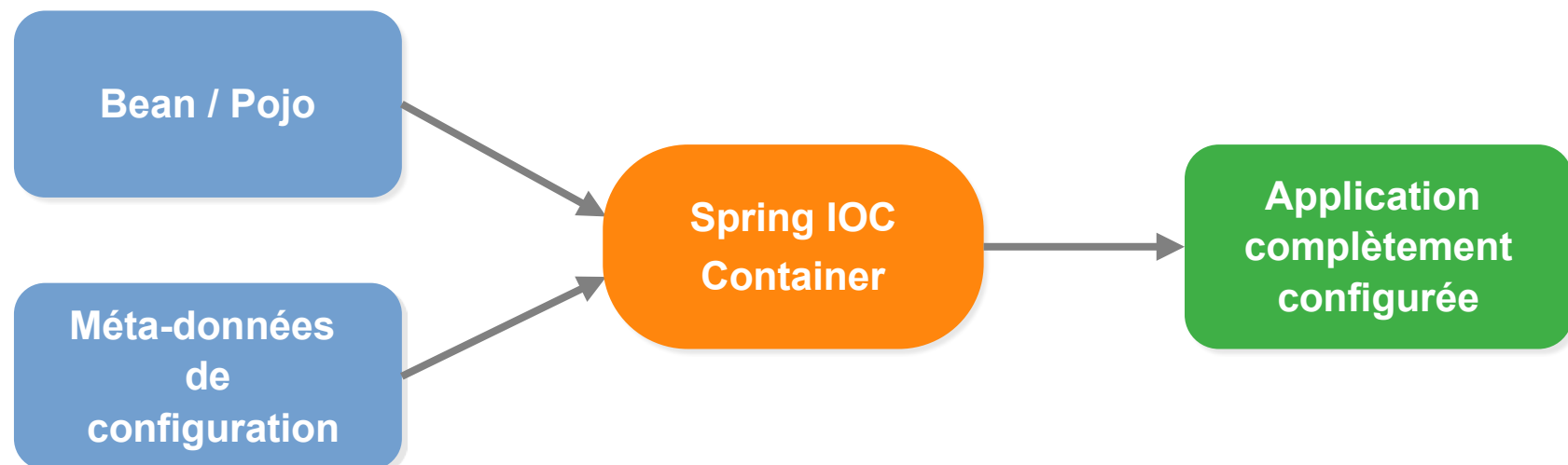
- Patron d'architecture qui fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application mais du framework
- Avec l'IoC, le framework prend en charge l'exécution principale du programme, il coordonne et contrôle l'activité de l'application
- IoC permet de découpler les dépendances entre objets (Couplage → degré de dépendance entre objets)
- Utilisation la plus connue : l'inversion des dépendances décrit dans l'article :

**dependency inversion principle de Robert C. Martin en 1994**

# Le conteneur d'inversion de contrôle

## Le conteneur d'inversion de contrôle est le cœur de spring

- Le conteneur d'ioC utilise l'injection de dépendance pour gérer les composants qui constituent une application
- Les objets gérés par le conteneur d'ioC sont des **beans**
- Il reçoit ses instructions pour l'instanciation, la configuration et l'assemblage des beans en lisant les **métadonnées de configuration**



# Le conteneur d'inversion de contrôle



- L'interface **ApplicationContext** représente le conteneur d'inversion de contrôle
- Plusieurs implémentations sont fournies avec Spring
  - pour les applications autonomes
    - `ClassPathXmlApplicationContext`
    - `AnnotationConfigApplicationContext`, ...
  - pour les applications web
    - `WebXmlApplicationContext`, ...
- Les méta-données peuvent être fournies :
  - en XML
  - avec les **annotations java** (Spring 2.5)
  - avec du **code Java** (Spring 3.0)

} à privilégier  
avec spring boot

# Déclarer un Bean

- L'annotation **@Configuration** placée sur une classe indique qu'elle fournit des définitions de bean
- Pour déclarer un bean, il faut annoter une méthode avec l'annotation **@Bean**
  - le type de retour de la méthode, définit le type du bean
  - par défaut, le bean aura le même nom que la méthode on peut le modifier avec l'attribut **name** de **@Bean**

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

    @Bean(name = { "dataSource", "dataSourceA" })
    public DataSource dataSource() { // ... }
}
```



# Instancier un conteneur



- On utilise pour implémentation de `ApplicationContext`:  
**`AnnotationConfigApplicationContext`**

En paramètre du constructeur, on utilise les classes annotée avec **`@Configuration`** qui contiennent la définition des beans

- On récupère les instances des beans depuis le conteneur avec la méthode :

**`T getBean(String name, Class<T> requiredType)`**

```
public static void main(String[] args) {  
    ApplicationContext ctx =  
        new AnnotationConfigApplicationContext(AppConfig.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

# Composants



- L'annotation **@Component**, lorsqu'elle est placée sur une classe, permet de déclarer un bean (composant)

Spring va :

- scanner l'application pour détecter les composants
  - les instancier et injecter toutes les dépendances
  - les injecter là où ils sont utilisés
- **@Component** est utiliser pour un composant générique, mais on peut catégoriser les composants avec :
    - **@Controller** pour les contrôleurs
    - **@Service** pour les services
    - **@Repository** pour les gestionnaires des données (DAO)

# Composants



- L'attribut **value** de ces annotations permet de spécifier le nom du bean, sinon par défaut, c'est le nom de la classe

```
@Service("contactService1")  
public class ContactServiceImpl implements ContactService{ }
```

- L'annotation **@ComponentScan** est placée sur la classe de configuration

Elle permet d'indiquer avec l'attribut **basePackages**, les packages où sont recherchés les composants

- Si on ne précise rien, elle les cherchera dans le package de la classe de configuration et ses sous-packages

```
@Configuration  
@ComponentScan(basePackages = "fr.dawan.formation")  
public class AppConfig { }
```

# L'annotation @Import

- L'annotation **@Import** permet de charger des définitions de bean depuis une autre classe de configuration

```
@Configuration
public class ConfigA {
    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B b(A a) {
        return new B(a);
    }
}
```

# Dépendance des beans

- On peut matérialiser la dépendances à l'aide des paramètres de la méthode

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(AccountRepository accRepository){
        return new TransferServiceImpl(accRepository);
    }
}
```

- Lorsque les beans ont des dépendances les uns aux autres  
On exprime cette dépendance en ayant une méthode du bean qui en appelle une autre

```
@Bean
public Foo foo() {
    return new Foo(bar());
}

@Bean
public Bar bar() {
    return new Bar();
}
```

# L'annotation `@Autowired`



- L'annotation `@Autowired` permet de faire de l'injection automatique de **dépendances basée sur le type**
- Elle s'utilise sur une **propriété**, un **setter** ou un **constructeur**
- L'attribut `required` permet de préciser si l'injection d'une instance dans la propriété est obligatoire, par défaut à `true`
- Depuis Spring 4.3, `@Autowired` sur le constructeur n'est plus nécessaire si le bean ne définit qu'un seul constructeur (Si plusieurs constructeurs sont disponibles, on doit en annoter un pour indiquer celui qui doit être utilisé)

```
public class PersonneService {  
    @Autowired  
    private PersonneDao personneDao;  
    public void setPersonneDao(PersonneDao personneDao) {  
        this.personneDao = personneDao;  
    }  
}
```

# L'annotation @Qualifier



- L'annotation **@Qualifier** permet de qualifier le candidat à une injection automatique **avec son nom**  
C'est utile lorsque plusieurs instances sont du type à injecter
- Elle s'utilise avec l'annotation **@Autowired**
- Elle peut s'appliquer sur
  - sur un attribut

```
@Autowired  
@Qualifier("per1")  
private Personne personne;
```

- sur un setter

```
@Autowired  
public void setPersonne(@Qualifier("per1") Personne personne){ }
```

- sur un constructeur

```
@Autowired  
public Constructeur(@Qualifier("per1") Personne personne) { }
```

# Les annotations de gestion du cycle de vie des beans



- La méthode annotée avec **@PostConstruct** sera exécutée après la création d'une nouvelle instance

```
public class MonBean {  
    @PostConstruct  
    public void initialiser() {  
    }  
}
```

- On peut utiliser aussi l'attribut **initMethod** de l'annotation **@Bean**

```
@Configuration  
public class AppConfig {  
    @Bean(initMethod = "initialiser")  
    public Foo foo() {  
        return new Foo();  
    }  
}
```



# Les annotations de gestion du cycle de vie des beans



- La méthode annotée avec **@PreDestroy** sera exécutée avant la destruction d'une instance

```
public class MonBean {  
    @PreDestroy  
    public void detruire() {  
    }  
}
```

- On peut utiliser aussi l'attribut **destroyMethod** de l'annotation **@Bean**

```
@Configuration  
public class AppConfig {  
    @Bean(destroyMethod = "detruire")  
    public Foo foo() {  
        return new Foo();  
    }  
}
```

# Les annotations de gestion du cycle de vie des beans



- Pour pouvoir utiliser les annotations **@PostConstruct** et **@PreDestroy** à partir de Java 11, il faut ajouter la dépendance suivante :

```
<dependency>
  <groupId>jakarta.annotation</groupId>
  <artifactId>jakarta.annotation-api</artifactId>
  <version>2.1.1</version>
</dependency>
```

- Elles sont dans le package **jakarta**

# L'annotation @Scope



- L'annotation **@Scope** permet de préciser la portée du bean
- Les valeurs utilisables sont :
  - **singleton (par défaut)**  
↳ crée une instance unique pour chaque conteneur IoC
  - **prototype**  
↳ crée une instance à chaque demande
  - **request (web)**    @RequestScope  
↳ crée une instance par requête HTTP
  - **session (web)**    @SessionScope  
↳ crée une instance par session HTTP

# L'annotation @Scope



- **application** (web) @ApplicationScope  
↳ crée une instance dont la durée de vie est celle du ServletContext
- **websocket** (web) @Scope(scopeName="websocket", proxyMode=ScopedProxyMode.TARGET\_CLASS)  
↳ crée une instance par session websocket

```
@Controller
@Scope("prototype")
public class MonController { // ... }

@Bean
@Scope("prototype")
public Person personPrototype() {
    return new Person();
}
```

# L'annotation @Lazy

- Par défaut, Spring crée tous les beans singleton au démarrage du contexte d'application
- L'annotation **@Lazy** permet de retarder le chargement des singletons à leur première utilisation
- On peut la placer, au niveau
  - de la classe : tous les beans seront en lazy loadings

```
@Lazy
@Configuration
@ComponentScan(basePackages = "fr.dawan.formation")
public class AppConfig {
```

- du bean :

```
@Lazy
@Bean
public Formation getFormation(){
    return new Formation();
}
```

# Dépendance Circulaire



- Dépendance Circulaire : via l'injection de constructeur
    - Le bean A nécessite une instance du bean B
    - Le bean B nécessite une instance du bean A
  - Elle est détectée par le conteneur d'ioc à l'exécution et lance une exception (`BeanCurrentlyInCreationException`)
  - **Solution**
    - **modifier l'architecture du code** : une dépendance circulaire, indique qu'il y a quelque chose qui ne va pas
- Sinon temporairement on peut :
- Placer l' **@Autowired** sur un **setter** ou sur un **attribut**

# Dépendance Circulaire

## – Utiliser @Lazy

```
@Component
public class BeanA {
    private BeanB b;
    @Autowired
    public BeanA(@Lazy BeanB b) {
        this.b = b;
    }
}
```

## – Utiliser @PostConstruct

```
@Component
public class BeanA {
    @Autowired
    private BeanB b;
    @PostConstruct
    public void init() {
        b.set(this);
    }
    public BeanB getCircB() {
        return b;
    }
}
```

- Maven est une **framework** de gestion de projets regroupant :
  - Un ensemble de standards
  - Un **repository** d'un format particulier
  - Un outil pour gérer et décrire un projet
- Il fournit un cycle de vie standard pour **Construire**, **Tester** et **Déployer** des projets selon une logique commune
- Il s'articule autour d'une déclaration commune de projet que l'on appelle le **POM** (Project Object Model)
- Maven structure un projet à partir d'un « archétype » et permet de construire son propre archétype pour reproduire un schéma



# POM

## (Project Object Model)

---



- Descripteur d'un projet Apache Maven, au format XML
- Indique à Maven quel type de projet il va devoir traiter et comment il va devoir s'adapter pour transformer les sources et produire le résultat attendu en définissant plusieurs goals (tâches)
- Ces tâches ou goals, utilisent le **POM** pour s'exécuter correctement. Des plugins peuvent être développés et utilisés dans de multiples projets de la même manière que les tâches pré-construites
- Description complète :

<http://maven.apache.org/ref/3-LATEST/maven-model/maven.html>

# Entête d'un POM (GAV)

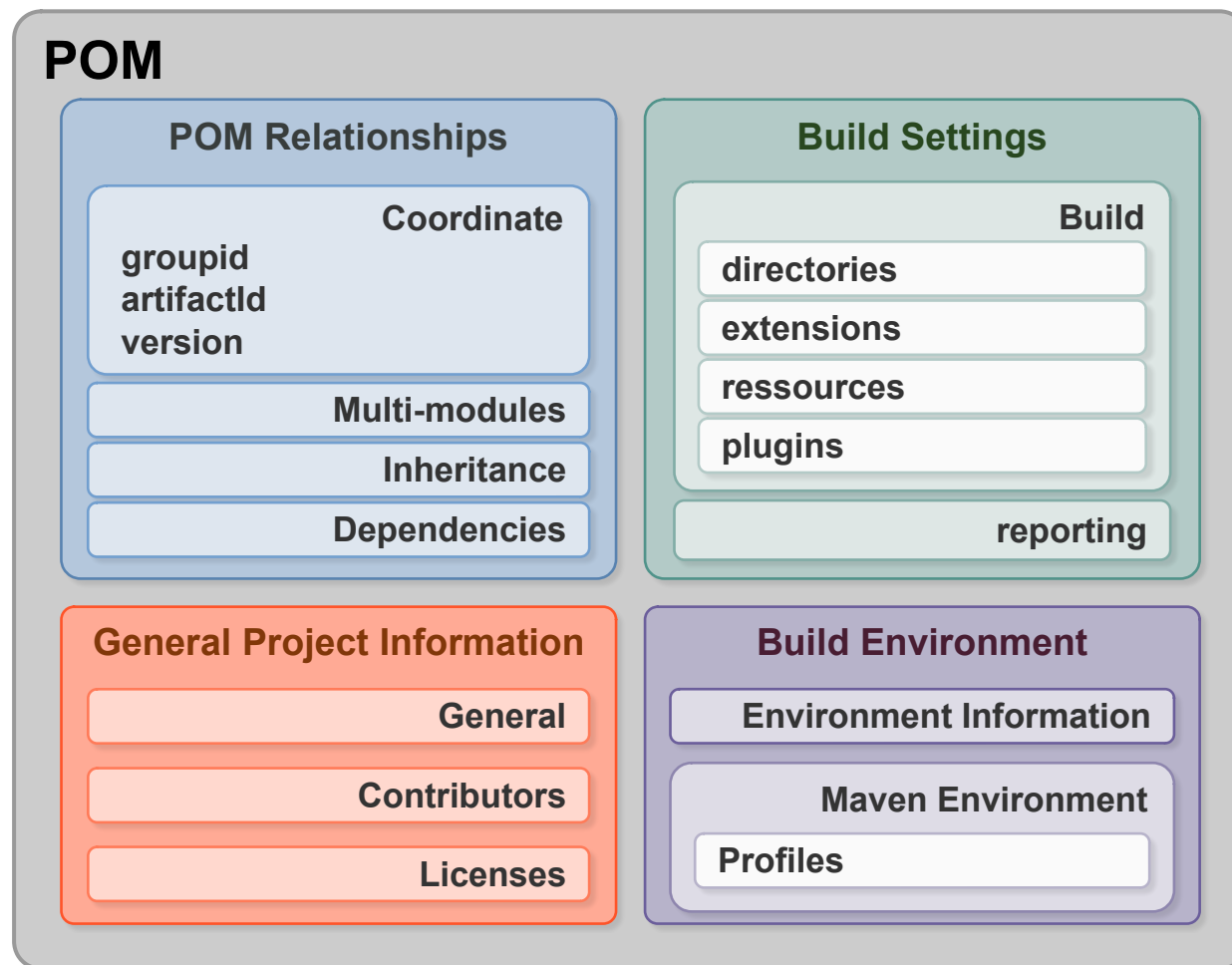


- Maven identifie de manière unique un projet avec :
  - **groupId**      identifiant arbitraire du groupe de projet habituellement basé sur le package Java ( sans espace, ni : )
  - **artifactId**    nom arbitraire du projet (sans espaces, ni :)
  - **version**        version du projet : **Major.Minor.Maintenance** si en développement, on ajoute **-SNAPSHOT**
- **GAV Syntaxe:**      groupId:artifactId:version

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dawan.training</groupId>
  <artifactId>maven-training</artifactId>
  <version>1.0</version>
</project>
```

# POM - Structure

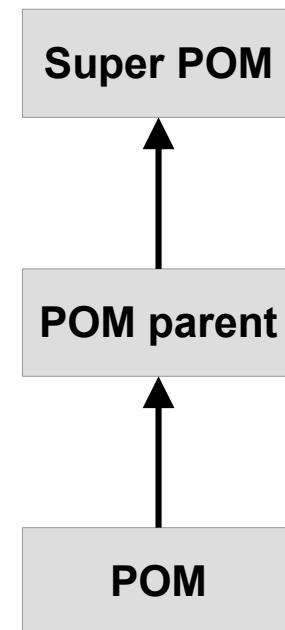
Le POM se compose de 4 catégories de description et de configuration



# POM - Héritage

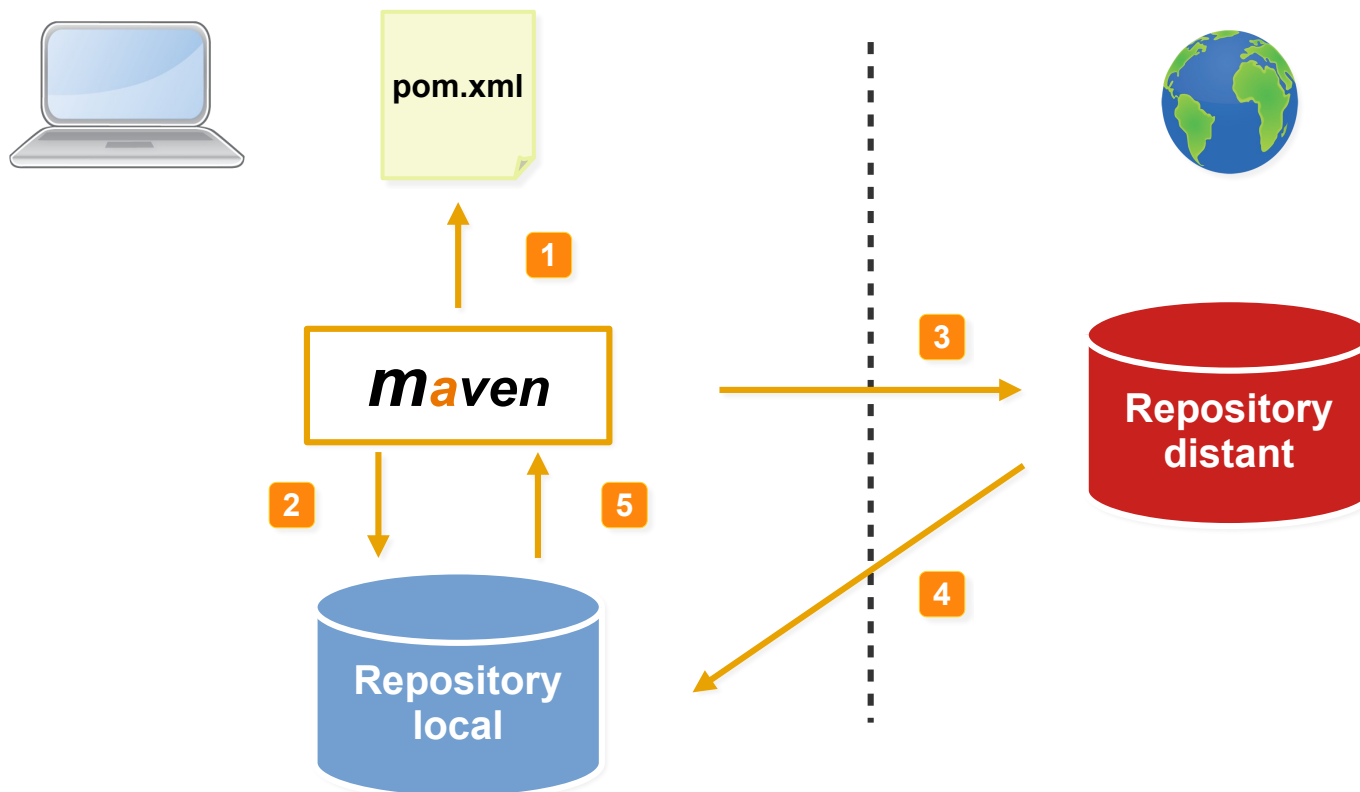
- Les fichiers POM peuvent hériter d'une configuration : groupId, version, configuration, dépendances ...

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <parent>
    <artifactId>maven-training-parent</artifactId>
    <groupId>com.dawan.training</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>maven-training</artifactId>
  <packaging>jar</packaging>
</project>
```



# Repository

- Emplacement des bibliothèques logicielles organisées selon une arborescence spécifique à Maven (settings.xml)
- 2 types de référentiels : local ou remote  
central Repository : <http://search.maven.org/>



# Cycle de Vie Maven



- **clean** nettoie le projet (supprimer les .class)
- **validate** valide le projet
- **compile** compile les sources du projet en java en .class
- **test** lancement des unitaire  
compile→test
- **package** empaquetage en archive jar, war...  
compile→test→package
- **verify** vérification du projet
- **install** installation  
compile→test→package→copie l'archive dans le repository local
- **site** intégration
- **deploy** déploiement

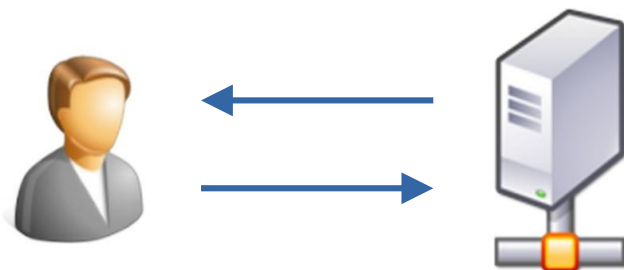
Spring Boot aide à créer des applications Spring autonomes

## Objectifs

- Démarrage plus rapide et plus accessible d'un projet Spring
- Fournir les valeurs par défaut dès le début que l'on pourra modifier par la suite
- Fournir des fonctionnalités non fonctionnelles communes à un grand nombre de projets (serveurs intégrés, sécurité, métriques, contrôles de santé et configuration externalisée)
- Aucune génération de code et pas de configuration XML requise

# Spring Boot

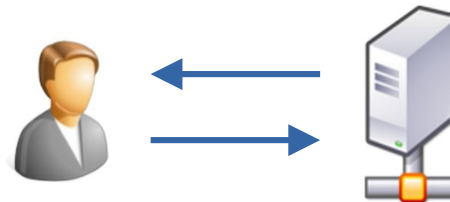
- Le développement d'application Web nécessite bien souvent des acteurs difficiles à mettre en place :
  - Un livrable
  - Un serveur applicatif
- Les configurations sont souvent lourdes, celles-ci bloquent :
  - Le déploiement à chaud de machines et services
  - Le déploiement rapide d'applications



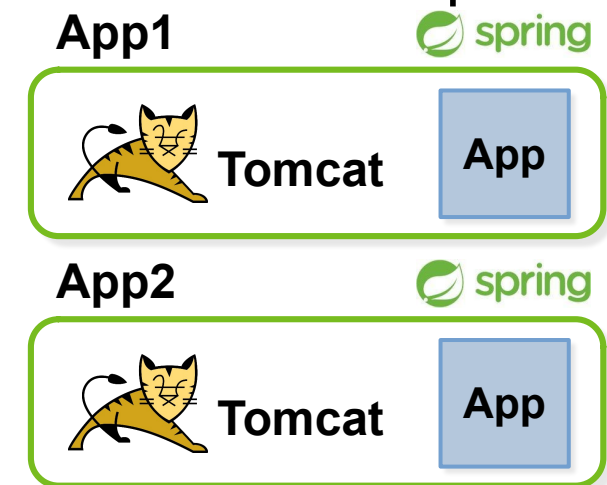


# Spring Boot

- Spring Boot va offrir un déploiement incluant le conteneur applicatif
- **Avantages :**
  - Le déploiement va pouvoir se faire rapidement sur plusieurs ordinateurs
  - Les Test d'intégrations sont simplifiés
  - Le besoin de Haute Disponibilité va pouvoir être satisfait plus rapidement



- **Inconvénients :**
  - Consommation mémoire plus importante dans le cas de beaucoup d'applications



# Spring Boot : Pré-requis

---



- Configuration requise pour Spring Boot 3.0.1
  - **java** : java 17 à java 19
  - **spring framework** : 6.0.3
  - **Maven** : 3.5+
- Spring Boot prend en charge les conteneurs de servlet intégrés
  - **Tomcat 10.0** (servlet 5.0)
  - **Jetty 11.0** (servlet 5.1)
  - **Undertow 2.2** (servlet 5.0)
- On peut déployer une application Spring Boot sur tous les conteneurs compatible avec les Servlet 5.0

# Les starters



- Les starters sont des descripteurs de dépendance simplifiés que l'on peut inclure dans l'application

on veut utiliser spring et jpa → spring-boot-starter-datajpa

- Tous les starters officielles se nomment suivant Le modèle **spring-boot-starter-\***

spring-boot-starter-web, spring-boot-starter-thymeleaf, spring-boot-starter-test, spring-boot-starter-security ...

- On a aussi les Spring Boot technical starters qui permet d'utiliser d'autre "moyen" technique

pour utiliser jetty au lieu de tomcat → spring-boot-starter-jetty

spring-boot-starter-undertow, spring-boot-starter-jetty, spring-boot-starter-logging, spring-boot-starter-log4j2 ...

# Les starters



- **spring-boot-starter-parent** est utilisé par tous les projets Spring Boot comme parent dans le pom.xml
- Il contient:
  - la version de java (par défaut Java 17)
  - les versions par défaut des dépendance de spring boot
  - configuration par défaut des plugins maven
- On peut redéfinir une version d'une dépendance en fournissant une propriété avec un nom correspondant à la dépendance dans le POM.xml

```
<properties>  
    <mockito.version>2.10.20</mockito.version>  
</properties>
```

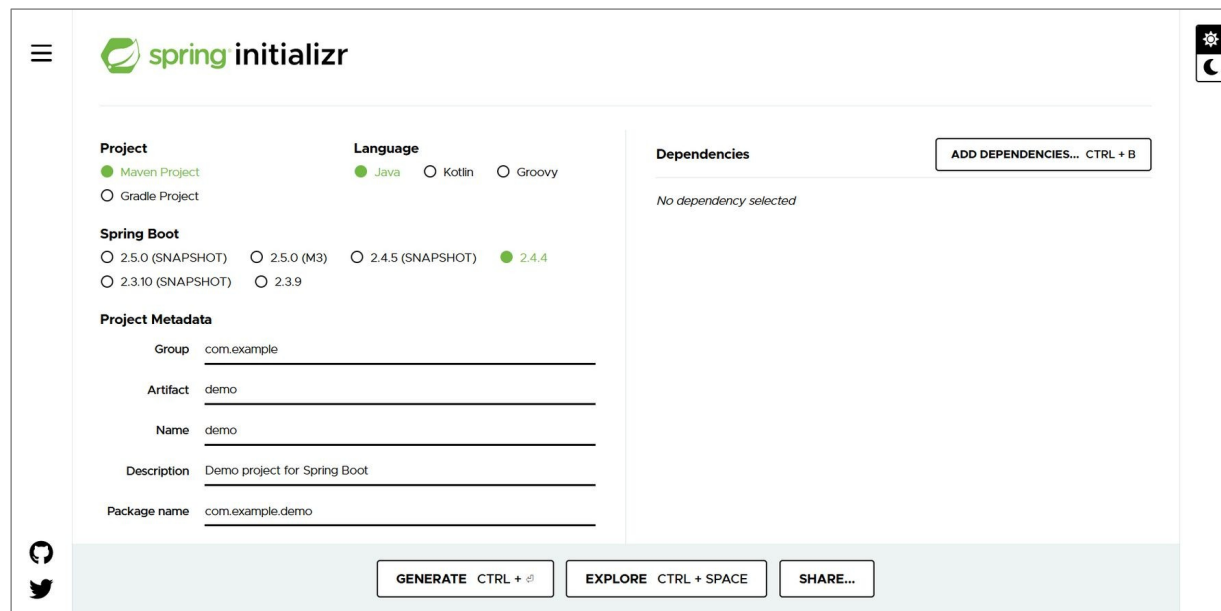
# Auto-configuration



- L'auto configuration de spring boot permet d'automatiquement configurer l'application à partir des dépendances des jars que l'on a ajouté  
si on a Hsqldb dans le classpath, Spring Boot auto-configures une base de donnée en mémoire)
- Si on définit un bean de configuration, il écrase la configuration par défaut
- L'auto configuration provient de :  
**spring-boot-autoconfiguration-{version}.jar**
- On peut afficher les auto-configuration qui sont appliquées avec les log en démarrant l'application en debug (avec **--debug**)

# Spring Initializer

- Outil permettant de choisir les frameworks Spring que l'on utilisera dans notre projet



The screenshot shows the Spring Initializer web interface. It features a sidebar with a hamburger menu and a Twitter icon. The main content area is divided into three sections: 'Project', 'Language', and 'Dependencies'. The 'Project' section has radio buttons for 'Maven Project' (selected) and 'Gradle Project'. The 'Language' section has radio buttons for '.Java' (selected), 'Kotlin', and 'Groovy'. The 'Dependencies' section has a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'. Below these sections is the 'Project Metadata' section with input fields for 'Group' (com.example), 'Artifact' (demo), 'Name' (demo), 'Description' (Demo project for Spring Boot), and 'Package name' (com.example.demo). At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. The Spring logo and 'spring initializr' text are at the top left of the main content area.

- **Spring initializer est :**
  - disponible au travers du site : <https://start.spring.io/>  
le projet est téléchargeable au format zip
  - intégré à Spring Tools Suite

# Spring Boot CLI



- Spring Boot CLI est un outil en ligne de commande qui peut être utilisé pour développer rapidement avec spring
- Il permet d'exécuter des scripts **Groovy**
- **Installation**
  - Téléchargement : [spring-boot-cli-3.0.1-bin.zip](#)
  - Variable d'environnement :

```
SPRING_HOME = Dossier d'installation  
Ajout au PATH =%SPRING_HOME%\bin
```

- Vérification

```
$ spring --version
```

# Spring Boot CLI



- **Exemple** : HelloWorld.groovy

```
@RestController
class HelloWorld {
    @RequestMapping("/")
    String home() {
        "Hello World!"
    }
}
```

- **Exécution** (uniquement spring boot CLI 2.x)

```
$ spring run HelloWorld.groovy
```

dans un navigateur : <http://localhost:8080>

↳ affiche: Hello World !



# Spring Boot CLI



- **Dépendance**

Spring Boot essaie de déduire les dépendances depuis le code

ex: `@RestController` → tomcat embedded et spring mvc

L'annotation groovy `@Grab` permet de définir les dépendances

ex : `@Grab('spring-boot-starter-freemarker')`

- **Commande**

← Uniquement spring boot CLI 2.x

**run** → exécuter les scripts groovy des applications Spring Boot

**jar** → créer un JAR exécutable à partir de scripts groovy

**shell** → démarrer un shell embarqué

**init** → initialiser un nouveau projet en utilisant Spring Initializr  
**--list** pour répertorier les capacités du service

...

# Environnements de développement

Tous les IDE supportant Java SE/Java EE

Les deux principaux :

- **Spring Tools Suite**  
(<https://spring.io/tools>)



↳ un IDE basé sur Eclipse fournit par Pivotal

- **IntelliJ IDEA**  
(<https://www.jetbrains.com/idea/>)

↳ existe en 2 versions :

- community edition (open-source et gratuit)
- ultimate (payante)



# Configuration de Spring Tool Suite 4



- À partir de eclipse **4.18**, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)

Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option **-vm**

```
-vm
```

```
C:\Program Files\Java\jdk1.8.0_351
```

- Dans windows → préférence

- filtre sur **jre**

Installed JREs → Add → choisir :

- Standard VM
  - JRE home : C:\Program Files\Java\jdk-17.0.5
  - JRE Name : jdk-17.0.5

# Configuration de Spring Tool Suite 4

---



- filtre sur **text editors**

  - cocher : Insert spaces for tabs

  - cocher : remove multiple spaces and backspace/delete

- filtre sur **spelling**

  - décocher : enable spelling

- filtre sur **formatter**

  - java → code style → Formatter

  - new → profil name : Eclipse

  - indentation : tab policy choisir **space only**

# Web Service REST : HelloWorld



- **pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.dawan</groupId>
  <artifactId>springboot</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.1</version>
  </parent>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
  </dependencies>
</project>
```

# Web Service REST : HelloWorld



- **HelloWorldController.java**

```
package fr.dawan.springboot.controllers;

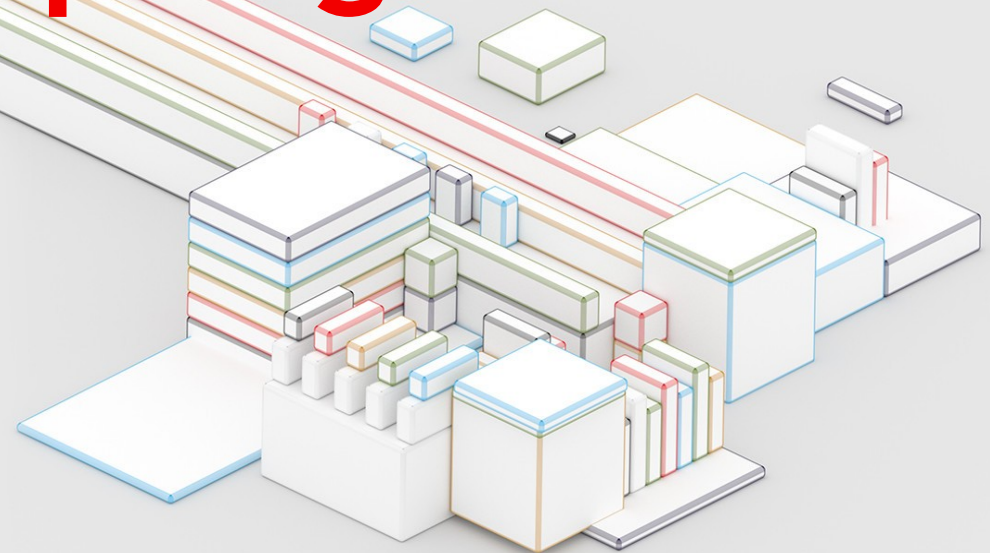
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

    @RequestMapping("/")
    public String helloWorld()
    {
        return "Hello World !";
    }

}
```

# Configurer Spring Boot



# La classe d'exécution principale

---



L'annotation **@SpringBootApplication** est une simple encapsulation de trois annotations :

- **@Configuration**  
permet de demander au conteneur d'utiliser cette classe pour instancier des beans
- **@EnableAutoConfiguration**  
permet au démarrage de spring, de générer automatiquement les configurations nécessaires en fonction des dépendances situées dans le classpath
- **@ComponentScan**  
indique qu'il faut scanner les classes de ce package afin de trouver des beans de configuration



# La classe d'exécution principale



- La classe **SpringApplication** permet de démarrer une application Spring à partir d'une méthode **main()**

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

- On peut créer une instance de **SpringApplication** et la personnaliser

```
@SpringBootApplication  
public class MyApp {  
    public static void main(String[] args) {  
        SpringApplication app = new SpringApplication(MyApp.class);  
        app.setBannerMode(Banner.Mode.OFF); // désactiver la bannière  
        app.run(args);  
    }  
}
```

# Configuration externe

Spring Boot permet d'externaliser la configuration, on peut utiliser le même code dans différents environnements

- Il offre le choix d'un fichier en **.properties** ou **.yaml**
- On peut aussi utiliser les variables d'environnements de l'OS, les argument de ligne de commande ...

L'ordre de priorité est :

- +
  - argument de ligne de commande
  - variables d'environnements de l'OS
  - fichier **.properties** ou **.yaml à l'extérieur du jar**
- - fichier **.properties** ou **.yaml à l'intérieur du jar**
- Des propriétés systèmes sont déjà définies et peuvent être redéfinies

# Configuration externe



- **Fichier properties**

```
spring.datasource.url=jdbc:mysql://localhost/FormationSpring
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
server.port=9002
```

- **Fichier yaml**

```
spring:
  datasource:
    url:jdbc:mysql://localhost/FormationSpring
    username:root
    password:
    driver-class-name:com.mysql.jdbc.Driver
server:
  port:9002
```

Les fichiers **.properties** sont prioritaires sur les fichiers **yaml**

# Configuration externe



- Par défaut SpringApplication convertie les arguments en ligne de commande qui commencent par -- en propriété
- Idem avec -D, pour les options de la JVM
- Pour désactiver les propriétés en ligne de commande: **SpringApplication.setAddCommandLineProperties(false)**

```
java -jar demo.jar --server.port=8081  
java -jar -Dserver.port=8081 myproject.jar
```

- **spring.config.name**: permet de remplacer le nom **application** du fichier de propriété

```
java -jar myproject.jar --spring.config.name=myproject
```

# Configuration externe



- Au démarrage de l'application va automatiquement trouver et charger **application.properties** et **application.yml** à partir de ces emplacements :

- +
  - racine du classpath
  - package /config du classpath
  - le répertoire courant
  - le sous-répertoire /config du répertoire courant
- - les répertoires enfants immédiat du sous répertoire /config

- **spring.config.location** permet de remplacer ces emplacements (liste de répertoire \ ou de fichier séparé par ,)

```
$ java -jar myproject.jar --spring.config.location=\
    optional:classpath:/default.properties,\
    optional:classpath:/custom-config/
```

# Configuration externe



- **spring.config.additional-location** ne remplace pas mais ajoute des emplacements
- Avec le préfixe **optional**, aucune exception ne sera lancée si l'emplacement n'existe pas
- **spring.config.import** permet d'importer d'autres données de configuration à partir d'autres emplacements

Les valeurs importées seront prioritaires sur celle du fichier qui a déclenché l'importation

```
spring.config.import=file:/etc/config/myconfig.properties
```

# Configuration externe



- **Propriétés Placeholders**




On peut faire référence aux valeurs précédemment définies avec : `${name}`

On peut aussi spécifier une valeur par défaut avec :  
`${name:default}`

```
app.name=MyApp  
app.description=${app.name} is a Spring Boot application  
written by ${username:Unknown}
```

↳ `app.description = MyApp is a Spring Boot application written by Unknown`

# Multi-profils

- Quand on déploie une application, il est nécessaire d'avoir différents environnements :
  -  application.properties
  -  application-DEV.properties
  -  application-PROD.properties
- Par exemple :
  - Les bases de données sont sur différents serveurs
  - Le système de fichiers change
  - ...
- Spring Boot permet la gestion des environnements en ajoutant un "-PROFIL" derrière le nom du fichier
- Le choix du profil se fait au démarrage de l'application  
`java -jar -Dspring.profiles.active=DEV demo.jar`
- Avec `@Component` et `@Configuration`, on peut définir le profil utilisé avec `@Profile(non_profil)`



# Propriétés par défaut

- Il est possible d'injecter des variables définies dans le fichier de propriétés

`@Value("${key}")`

```
@Component
public class MyBean {
    @Value("${converter}")
    private String path;
    // ...
}
```

```
...
spring.jpa.hibernate.connection.autocommit=false
server.port=9002
converter=C:/Program Files/ImageMagick-7.0.4-Q16/magick.exe
spring.http.multipart.max-file-size=10MB
...
```

- **path** prendra la valeur :  
C:/Program Files/ImageMagick-7.0.4-Q16/magick.exe

# Bannière

- On peut changer la bannière afficher au démarrage en ajoutant à la racine des ressources
  - un fichier texte **banner.txt**
  - ou un fichier image **banner.gif, banner.jpg, banner.png** (converties en ASCII art)
- On peut changer le chemin du fichier
  - texte avec **spring.banner.location**
  - image avec **spring.banner.image.location**
- La méthode **setBanner** de **SpringApplication** permet de générer une bannière en implémentant **printBanner()** de l'interface **org.springframework.boot.Banner**
- Générateur d'ASCII art :  
<http://patorjk.com/software/taag/#p=display&f=Standard&t=Dawan>

Uniquement  
spring boot 2.x

# Bannière



- Dans le fichier **banner.txt**, on peut utiliser les variables :

<code>\${application.version}</code>	numéro de version de l'application dans MANIFEST.MF du jar
<code>\${application.formatted-version}</code>	
<code>\${application.title}</code>	titre de l'application dans MANIFEST.MF du jar
<code>\${spring-boot.version}</code>	version de spring boot
<code>\${spring-boot.formatted-version}</code>	
<code>\${AnsiBackground.NAME}</code>	ANSI escape code → NAME est le nom de l'ANSI escape code (ex : RED)
<code>\${AnsiColor.NAME}</code>	
<code>\${AnsiStyle.NAME}</code>	
- **spring.main.banner-mode** permet de modifier le mode d'affiche de la bannière
  - **console** → avec System.out
  - **log** → l'envoi vers le logger
  - **off** → pour ne pas l'afficher

# Spring Boot Runners



- **ApplicationRunner** et **CommandLineRunner** sont deux interfaces qui contiennent une méthode **run(...)** qui est exécuté une seule fois par **SpringApplication.run(...)** après l'initialisation du context

```
interface ApplicationRunner {  
    void run(ApplicationArguments args); }  
  
interface CommandLineRunner {  
    void run(String[] args); }
```

- Si on a plusieurs implémentations de ces interfaces, l'annotation **@Order** permet d'indiquer l'ordre d'exécution
- Utilisation :
  - créer un application console non-web avec spring boot
  - préparer les données initiales de l'application

...

# Spring Boot Runners

- On peut implémenter ces interfaces :
  - comme un bean avec **@Bean** ou **@Component**
  - dans la classe annoté avec **@SpringBootApplication**

```
@Component
public class ApplicationRunner implements CommandLineRunner {
    @Override
    public void run(String[] args) {
        System.out.println(" ApplicationRunner");
    }
}

@SpringBootApplication
public class Application implements CommandLineRunner {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(SpringBootWebApplication.class, args);
    }
    @Override
    public void run(String...args) throws Exception {
        System.out.println("Application");
    }
}
```

# Developer Tools



- Developer tools → outils pour faciliter le développements
- Dépendance pour ajouter le support des devtools

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-devtools</artifactId>  
  <optional>true</optional>  
</dependency>
```

- Si l'application est lancée avec **java -jar** , Elle est considérée comme application en production et les devtools sont désactivé (risque de sécurité)

## Fonctionnalités ajoutées :

- Valeurs par défaut des propriétés  
↳ désactivation automatique des caches (thymeleaf... )  
...
- Redémarrage automatique
- Live Reload → déclencher une actualisation du navigateur lorsqu'une ressource est modifiée  
(nécessite l'installation d'une extension sur le navigateur)
- Paramètres globaux qui ne sont associés à aucune application → fichier : spring-boot-devtools.properties
- Débogage à distance via HTTP (Remote Debug Tunnel)

# Valeurs par défaut des propriétés



- Le cache des moteurs de template sont désactivé  
`spring.freemarker.cache, spring.thymeleaf.cache,  
spring.template.provider.cache, spring.web.resources.chain.cache  
spring.groovy.template.cache, spring.mustache.cache, ← false  
spring.web.resources.cache.period ← 0`
- La console de H2 est activé  
`spring.h2.console.enabled ← true`
- Les données de session sont conservées entre les redémarrages  
`server.servlet.session.persistent ← true`
- Les messages, stacktraces, erreurs sont incluses aux erreurs  
`server.error.include-message, server.error.include-stacktrace,  
server.error.include-binding-errors ← always`
- Pour ne pas appliquer les valeurs par défaut des propriétés  
`spring.devtools.add-properties ← false`



# Redémarrage automatique



- Une application qui utilise **spring-boot-devtools** redémarre automatiquement quand un fichier est modifié dans le **classpath**
- Le redémarrage avec spring boot fonctionne à l'aide de deux chargeurs de classe :
  - Les classes qui ne change pas sont chargés dans un chargeur de classe de base
  - Les classes que vous développez activement sont chargées dans un chargeur de classes de redémarrage
- Les redémarrages des applications sont généralement beaucoup plus rapides que les «démarrages à froid»

# Redémarrage automatique



- **Déclencher un redémarrage**

Le déclenchement du redémarrage dépend de l'EDI:

- **Eclipse** → l'enregistrement
  - **IntelliJ IDEA** → la construction du projet  
(Build + → + Build Project)
  - **build plugin de Maven** → mvn compile
- On peut activer/désactiver le redémarrage avec la propriété **spring.devtools.restart.enabled**

# Réglages globaux



- On peut configurer des réglages globaux pour devtools en ajoutant un de ces fichiers dans le dossier `$HOME/.config/spring-boot`
  - `spring-boot-devtools.properties`
  - `spring-boot-devtools.yaml`
  - `spring-boot-devtools.yml`
- Il sont prioritaires aux configuration externe
- Toutes les propriétés placées dans ces fichiers s'appliquent à toutes les applications de la machine qui utilisent devtools
- Les profils ne fonctionnent pas avec le réglages globaux

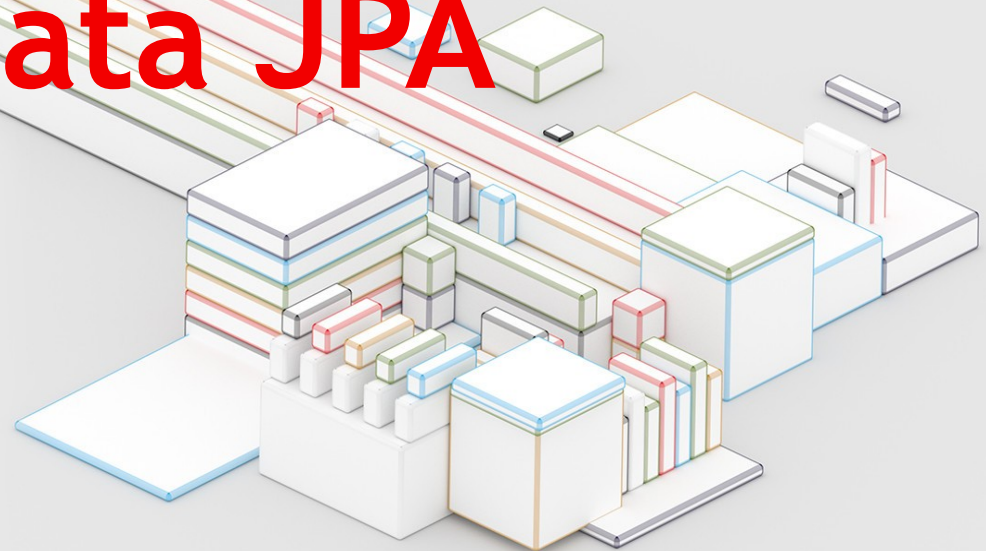
# Mise à niveau de Spring Boot



- Pour passer à une nouvelle version de spring boot et mettre à jour toutes les dépendances, il suffit de modifier la version de **spring-boot-starter-parent**
- La dépendance **spring-boot-properties-migrator** permet lorsqu'elle est ajoutée dans le pom.xml, d'afficher un diagnostic sur les propriétés qui ont été ajoutées ou renommées
- Elle va aussi migrer temporairement les propriétés au moment de l'exécution
- Une fois la mise à jour effectuée, la retirer du pom.xml

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-properties-migrator</artifactId>  
  <scope>runtime</scope>  
</dependency>
```

# Mapping d'entités avec Spring Data JPA



# Mapping relationnel-objet

---



Concept permettant de connecter un modèle objet à un modèle relationnel

Couche qui va interagir entre l'application et la base de données

- **Pourquoi utiliser ce concept?**
  - Pas besoin de connaître l'ensemble des tables et des champs de la base de données
  - Faire abstraction de toute la partie SQL d'une application

# Mapping relationnel-objet

Développeur

Objet

Personne

id  
nom  
prenom

ORM

Système

SGBD

personnes

Id	Nom	Prenom
...	...	...

# Mapping relationnel-objet

---



- **Avantages :**
  - Gain de temps au niveau du développement d'une application
  - Abstraction de toute la partie SQL
  - La portabilité de l'application d'un point de vue SGBD
- **Inconvénients :**
  - L'optimisation des frameworks/outils proposés
  - La difficulté à maîtriser les frameworks/outils



# JPA

- Une API (Java Persistence API)
- Des implémentations



- Permet de définir le mapping entre des objets Java et des tables en base de données
- Remplace les appels à la base de données via JDBC

# Configurer une connexion



- **Dépendances Maven**

- Dépendance pour initialiser JPA

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

- Dépendance pour charger le driver de base de données

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

# Configurer une connexion

Pour configurer une connexion il faut définir les propriété suivante :

Propriétés	Description
<code>spring.datasource.url</code>	URL JDBC de la base de données
<code>spring.datasource.username</code>	Nom d'utilisateur de connexion de la base de données
<code>spring.datasource.password</code>	Mot de passe de connexion de la base de données
<code>spring.datasource.driver-class-name</code>	Nom complet du pilote JDBC
<code>spring.jpa.properties.hibernate.dialect</code>	permet à Hibernate de générer du SQL optimisé pour une base de données particulière ( <a href="#">liste</a> )
<code>spring.jpa.show-sql</code>	Afficher les requête SQL (par défaut : false)
<code>spring.jpa.hibernate.ddl-auto</code>	Mode DDL: create, update, create-drop (par défaut pour une base de donnée intégrée), none (par défaut)

# Configurer une connexion

- Exemple de fichier **application.properties**

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/formation
spring.datasource.username=root
spring.datasource.password=mot de passe
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect
                                =org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
```

- Ces valeurs sont en fait les valeurs de configuration d'hibernate (persistence.xml)

# Entité



- Avec JPA, Une entité est une classe dont les instances peuvent être persisté en base de données
- Une entité est une classe Java standard qui doit :
  - être identifiée comme une entité avec l'annotation **@Entity**
  - avoir un attribut qui joue le rôle d'identifiant annoté avec **@Id** (représente la clé primaire de la table)
  - avoir un **constructeur sans argument**
  - Implémenter l'interface **Serializable**
  - ne doit pas être **final**
  - aucune de ses méthodes ne peut être **final**

# Entité

```
@Entity
public class Personne implements Serializable {
    @Id
    private long id;
    private String firstName;
    private String name;
    public Personne() {
        super();
    }
    // Getters / Setters
}
```

- **Nom de l'entité**

Par défaut, le nom de l'entité est le nom de la classe

L'attribut **name** de l'annotation **@Entity** permet de donner un autre nom à l'entité

# Entité

```
@Entity(name="dept")  
public class Departement { //... }
```

- **Nom de la table**

Par défaut, le nom de table associé est le nom de la classe

On peut le changer avec l'annotation

**@Table**(name = "nom\_table")

```
@Entity  
@Table(name="personnes")  
public class Personne {  
  
    // ...  
  
}
```

# Attributs persistants

- Par défaut, **tous les attributs** d'une entité sont persistants
- Les attribut qui ne sont pas persister sont ceux :
  - qui ont pour annotation **@Transient**
  - dont la variable de l'attribut est **transient**
  - qui sont **final** et/ou **static**

```
@Entity  
public class NoPersist{  
    @Transient  
    int attr1;  
    transient int attr2;  
    static int attr3;  
    final int attr4=0;  
}
```



# Énumération

- Une énumération est stockée par défaut sous forme numérique (0,1,... n)
- L'annotation `@Enumerated` permet de définir comment l'énumération sera stockée
  - `EnumType.ORDINAL` stockée sous forme numérique
  - `EnumType.STRING` le nom de l'énumération est stocké

```
public enum Civilite{  
    MADemoiselle, MADame, MONSIEUR  
}  
  
@Entity  
public class Personne{  
    @Enumerated(EnumType.STRING)  
    private Civilite civ; // Stocke MADemoiselle,  
                           MADame ou MONSIEUR  
    // ...                dans la base de données
```

- L'annotation **@Lob** indique que l'attribut de l'entité est un type de données de longueur variable pour stocker des objets volumineux (Large Object)
- Le type de données peut être un :
  - **CLOB** (Character Large Object) pour stocker du texte
  - **BLOB** (Binary Large Object) pour stocker des données binaires (images, audio ...)
- Un BLOB sera stocké dans un tableau d'octet

```
@Entity
public class User {
    @Id
    private long id;
    @Lob
    private byte[] photo;
```

# Propriétés de la colonne



- Par défaut, une colonne de la table aura le nom de l'attribut correspondant
- L'annotation **@Column** permet pour définir plus précisément la colonne, avec les attributs suivant :
  - **name:** permet de définir le nom de la colonne
  - **unique:** permet de définir si le champs doit être unique (par défaut à **false**)
  - **nullable:** permet de définir si le champ peut être null (par défaut à **true**)
  - **length:** pour les chaînes de caractères, permet de définir la longueur (par défaut 255)
  - **precision:** permet de définir la précision pour un nombre décimal (par défaut 0)

# Propriétés de la colonne



- **scale**: permet de définir l'échelle d'un nombre décimal (par défaut 0)
- **insertable**: permet de définir, si la colonne est prise en compte pour une requête insert (par défaut à **true**)
- **updatable**: permet de définir si la colonne est prise en compte pour une requête update (par défaut à **true**)
- **columnDefinition**: permet de donner en SQL, le code de création d'une colonne (DDL)

En général à éviter

Permet de donner un valeur par défaut à une colonne :

`@Column(columnDefinition=" default '10' ")`


```
@Column(name="family_name", length=50)
private String nom ;
@Column(name="first_name", length=50)
private String prenom ;
```

# Classe Intégrable

- Une classe intégrable va stocker ses données dans la table de l'entité mère ce qui va créer des colonnes supplémentaires
- La classe intégrable est annotée avec **@Embeddable**
- L'attribut de l'objet dans la classe mère doit utiliser l'annotation **@Embedded**

```
@Embeddable
public class PersonneDetail{
    private LocalDate birthday;
}
```

```
@Entity
public class Personne{
    @Id
    private long id;
    private String prenom;
    private String nom;
    @Embedded
    private PersonneDetail detail;
    //...
}
```



# Utilisation multiple d'une classe intégrable

---



- Une classe entité peut référencer plusieurs instances d'une même classe intégrable
- Les noms des colonnes dans la table de l'entité ne peuvent être les mêmes pour chacune des utilisations
- Un champ annoté par `@Embedded` peut être complété par une annotation `@AttributeOverride`, ou plusieurs insérées dans une annotation `@AttributeOverrides`
- Elles permettent d'indiquer le nom d'une ou de plusieurs colonnes dans la table de l'entité

# Utilisation multiple d'une classe intégrable

```
@Embeddable  
public class Adresse {  
    private String rue;  
    private String ville;  
    private int codePostal;  
    ...  
}
```

```
@Entity  
public class Employe {  
    @Embedded  
    private Adresse adresse;  
  
    @Embedded  
    @AttributeOverrides({  
        @AttributeOverride(  
            name="ville",  
            column=  
                @column(name="ville_travail")),  
        @AttributeOverride( ... )  
    })  
    // Adresse du travail  
    private Adresse adresseTravail;  
    // ...  
}
```

# Clé primaire



Une entité doit avoir un attribut qui correspond à la clé primaire dans la table associée

- **Clé primaire simple**

Une entité a un attribut unique qui sert de clé primaire  
L'attribut clé primaire est désigné par l'annotation **@Id**

- **Clé primaire composée**

Une clé primaire peut être composée de plusieurs colonnes

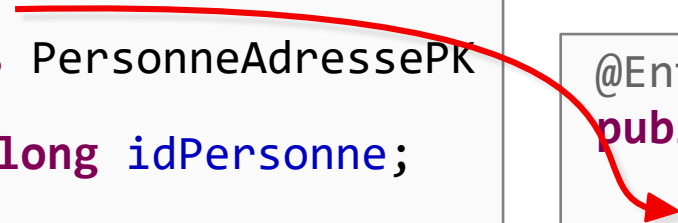
Pour mapper une clé primaire composée, on crée une classe intégrable **@Embeddable** qui ne contient que les champs de la clé primaire et on l'utilise dans l'entité principale avec l'annotation **@EmbeddedId**



# Clé primaire

```
@Embeddable  
public class PersonneAdressePK  
{  
    private long idPersonne;  
  
    private long idAdresse;  
  
    public PersonneAdressePK()  
    {  
  
    }  
    // ...  
}
```

```
@Entity  
public class PersonneAdresse {  
    @EmbeddedId  
    protected PersonneAdressePK pAddrPK;  
    //...  
}
```



# Génération automatique de clé primaire



- L'annotation **@GeneratedValue** indique que la clé primaire est générée automatiquement lors de l'insertion en Bdd
- Elle doit être utilisée en complément de l'annotation **@id**
- Elle a 2 attributs :
  - **generator** : contient le nom du générateur à utiliser
  - **strategy** : permet de spécifier le mode de génération de la clé primaire

## **GenerationType.AUTO** (par défaut)

La génération est gérée par l'implémentation de l'ORM Hibernate crée une séquence unique via la table **hibernate\_sequence**

# Génération automatique de clé primaire



## GenerationType.IDENTITY

La génération se fait à partir d'une propriété entity propre au système de gestion de bdd

## GenerationType.TABLE

La génération s'effectue en utilisant une table pour assurer l'unicité. Hibernate crée une table **hibernate\_sequence** qui stocke les noms et les valeurs des séquences à utiliser avec l'annotation **@TableGenerator**

```
@Entity
@TableGenerator(name="personneGen")
public class Personne{
    @Id
    @GeneratedValue(strategy = GenerationType.TABLE
                    ,generator = "personneGen")
    private Integer id;
}
```

# Génération automatique de clé primaire



## GenerationType.SEQUENCE

La génération se fait par une séquence définie par le système de gestion de bdd. à utiliser avec l'annotation **@SequenceGenerator**

```
@Entity
@SequenceGenerator(name="seqGen")
public class Personne{
    @Id
    @GeneratedValue(strategy =
        GenerationType.SEQUENCE,generator = "seqGen")
    private Integer id;
}
```

# Héritage



Il existe trois façons d'organiser l'héritage :

- **SINGLE\_TABLE**  
@Inheritance  
@DiscriminatorColumn  
@DiscriminatorValue
- **TABLE\_PER\_CLASS**  
@Inheritance
- **JOINED**  
@Inheritance

La différence entre elles se situe au niveau de l'optimisation du stockage et des performances

# Héritage : SINGLE\_TABLE

- Tout est dans la même table
- Une colonne, appelée "**Discriminator**" définit le type de la classe enregistrée
- De nombreuses colonnes inutilisées

```
@Entity
@Inheritance (strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="compte_discriminator",
discriminatorType=DiscriminatorType.STRING,length=15)
public abstract class Compte implements Serializable{...}

@Entity
@DiscriminatorValue("COMPTE_EPARGNE")
public class CompteEpargne extends Compte
implements Serializable {...}
```

# Héritage : TABLE\_PER\_CLASS



- Chaque Entity Bean fils a sa propre table
- Lourd à gérer pour le polymorphisme

```
@Entity
@Inheritance (strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Compte implements Serializable{
    //...
}
```

**La clé @Id ne peut pas être @GeneratedValue(Identity)**

```
@Entity
public class CompteEpargne extends Compte
implements Serializable {
    //...
}
```

# Héritage : JOINED

- Chaque Entity Bean a sa propre table
- Beaucoup de jointures

```
@Entity
@Inheritance (strategy=InheritanceType.JOINED)
public abstract class Compte implements Serializable {
    //...
}

@Entity
public class CompteEpargne extends Compte
implements Serializable {
    //...
}
```



# Héritage : récapitulatif

Stratégie	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
<b>Avantages</b>	Aucune jointure, donc très performant	Performant en insertion	Intégration des données proche du modèle objet
<b>Inconvénients</b>	Organisation des données non optimale	Polymorphisme lourd à gérer	Utilisation intensive des jointures, donc baisse des performances

# Classe mère persistante

- Une entité peut aussi avoir une classe mère dont l'état est persistant, sans que cette classe mère ne soit une entité
- La classe mère a pour annotation **@MappedSuperclass**
- Aucune table ne correspondra à cette classe mère dans la base de données. L'état de la classe mère sera rendu persistant dans les tables associées à ses classes entités filles

```
@MappedSuperclass
public abstract class Base {
    @Id @GeneratedValue
    private Long Id;
    @ManyToOne
    private User user;
    ...
}
```

# Gestion de la concurrence



- La gestion de la concurrence est essentielle dans le cas de longues transactions
- Hibernate possède plusieurs modèles de concurrence :
  - **None** : la transaction concurrentielle est déléguée au SGBD → Elle peut échouer
  - **Optimistic (Versioned)** : si on détecte un changement dans l'entité, nous ne pouvons pas la mettre à jour  
@Version(Numeric, Timestamp, DB Timestamp)  
↳ On utilise une colonne explicite Version (meilleure stratégie)
  - **Pessimistic** : utilisation des LockMode spécifiques à chaque SGBD

# Gestion de la concurrence

## Versioned



- L'élément **@Version** indique que la table contient des enregistrements versionnés
- La propriété est incrémentée automatiquement par Hibernate
- Automatiquement, la requête générée inclura un test sur ce champ :

```
UPDATE Player SET version = @p0, PlayerName = @p1  
WHERE PlayerId = @p2  
AND version = @p3;
```

# Mapping des collections simples



- **@ElementCollection** sur une collection **simple** permet de générer une table **NomClasse\_nomVar**
- **@CollectionTable** permet de personnaliser de la table

```
@ElementCollection(targetClass = String.class)
@CollectionTable(name = "prod_comments" , joinColumns =
@JoinColumn(name = "prod_id"))
private List<String> comments;
```

- **@MapKeyColumn** permet de personnaliser, la colonne de la clé dans la table  
pour la colonne valeur, on utilise **@Column**

```
@ElementCollection
@CollectionTable(name="EMP_PHONE")
@MapKeyColumn(name="PHONE_TYPE")
@Column(name="PHONE_NUM")
private Map<String, String> phoneNumbers;
```

# Relations entre Entity Beans



- **One-To-One** → 1,1  
@OneToOne  
@PrimaryKeyJoinColumn  
@JoinColumn
- **Many-To-One** → 1,n  
@ManyToOne  
@JoinColumn
- **One-To-Many**  
@OneToMany  
(pas de @JoinColumn)
- **Many-To-Many** → n,m  
@ManyToMany  
@JoinTable

# Stratégies de chargement des relations



- **Chargement tardif : LAZY**

Les entités en relation ne sont chargées qu'au moment de l'accès

↳ par défaut pour **@OneToMany** et **@ManyToMany**

- **avantages**

- temps de chargement initial beaucoup plus court
- moins de consommation de mémoire

- **désavantages**

- peut avoir un impact sur les performances lors de moments indésirables
- risque d'exception **LazyInitializationException**

```
@OneToMany(mappedBy="customer", fetch=FetchType.EAGER)
```

# Stratégies de chargement des relations



- **Chargement immédiat : EAGER**

Les entités en relation sont chargées dès le load de l'objet

↳ par défaut pour `@OneToOne` et `@ManyToOne`

- **avantages**

- aucun impact sur les performances lié à l'initialisation retardée

- **désavantages**

- temps de chargement initial long
    - charge trop de données inutiles, peut avoir un impact sur les performances

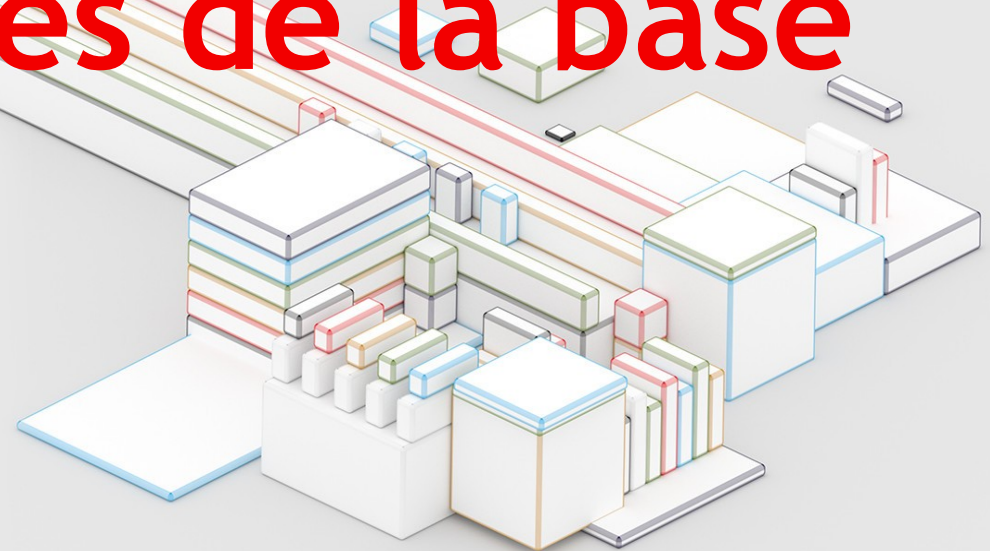


# Traitement en cascade



- Les annotations **@OneToOne**, **@OneToMany**, **@ManyToOne** et **@ManyToMany** possèdent l'attribut **cascade**
- Une opération appliquée à une entité est propagée aux relations de celle-ci :  
par exemple, lorsqu'un utilisateur est supprimé, son compte l'est également
- 4 Types : **PERSIST** , **MERGE** , **REMOVE** , **REFRESH**
- **CascadeType.ALL** : cumule les 4

# Implémenter des requêtes sur les données de la base



# Repository



- Pour implémenter des requêtes sur les données de la base, on va créer une classes d'accès aux données, annoté avec **@Repository**
- Le framework Spring Data JPA permet d'offrir une surcouche de JPA avec un ensemble de DAO déjà prêts
- L'interface central de spring Data est **Repository<T, ID>**
  - **T** correspond à la classe de l'entité
  - **ID** correspond à la classe de l'Id de l'entité
- C'est un interface marqueur qui sert à définir les données avec lesquels on va travailler

# CrudRepository, JpaRepository



- L'interface **CrudRepository** fournit à la classe entité des fonctionnalités **CRUD** (**C**reate, **R**ead, **U**psert et **D**eleter) hérite de l'interface Repository)

```
public interface CrudRepository<T, ID> extends Repository<T, ID>
{
    <S extends T> S save(S entity);    // Sauvegarde l'entité
    Optional<T> findById(ID primaryKey); // Retourne l'entité
                                         en fonction de son ID
    Iterable<T> findAll(); // Retourne toutes les entités
    long count();          // Retourne le nombre d'entités
    void delete(T entity); // Supprime l'entité
    boolean existsById(ID primaryKey); // test si l'entité qui
                                         a pour id ID existe

    // ...
}
```

- Il existe aussi des interfaces dépendant de la technologie utilisée (ex: **JpaRepository**, **MongoRepository** ...) qui hérite de **CrudRepository**

# Définir les méthodes de la requête



- Dans l'interface repository, on peut créer les requêtes de deux façons :
  - En dérivant la requête depuis le nom de la méthode directement
  - En utilisant une méthode annotée avec **@Query** qui contient la requête écrite manuellement (en JPQL ou en SQL natif)

```
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {
    List<Person> findByEmailAddressAndLastname(EmailAddress
                                                emailAddress, String lastname);
}

@Repository
public interface QuizRepository extends JpaRepository<Quiz, Long> {
    @Query(value = "FROM Quiz qz WHERE qz.id=:quizId")
    Quiz findQuizById(@Param("quizId") long quizId);
}
```

# Création de requêtes avec le nom des méthodes



L'analyse des noms de méthodes est divisée en

- **Sujet**(find...By, exists...By, count...By, delete...By)  
Il peut contenir d'autre expression, tout le texte entre find et by est considéré comme descriptif  
Sauf pour les mots clés qui limite le résultat (Distinct, Top/First)  
Le premier By agit comme un délimiteur pour indiquer le début du prédicat
- **Prédicat**, de façon général :
  - On peut définir des conditions sur les propriétés d'entité et les concaténer avec **AND** et **OR** et aussi les opérateurs comme **Between**, **LessThan**, **GreaterThan** et **Like**

# Création de requêtes avec le nom des méthodes



**LIKE** permet d'effectuer une recherche sur un modèle particulier

% et \_ sont 2 jokers :

- % représente : 0, 1 ou plusieurs caractères
- \_ représente : un caractère

– On peut utiliser **IgnoreCase** pour

- des propriétés individuelles  
(ex: `findByLastnameIgnoreCase`)
- toutes les propriétés d'un type  
(ex: `findByLastnameAndFirstnameAllIgnoreCase`)

– On peut ordonner avec **OrderBy** et l'ordre de trie avec **Asc** ou **Desc**

# Création de requêtes avec le nom des méthodes



```
@Repository
interface PersonRepository extends Repository<Person, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress,
                                              String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
                                                         String firstname);
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname,
                                                         String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);
    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname,
                                                         String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```



# Paging and Sorting



- **Pageable** → permet d'ajouter dynamiquement une pagination à la requête
- **Page** → contient le nombre d'élément et le nombre de page disponible
- **Slice** → un slice ne sait seulement si un autre Slice à la suite

```
Page<User> findByLastname(String lastname, Pageable pageable);  
Slice<User> findByLastname(String lastname, Pageable pageable);
```

- Les options de tri sont également gérées via l'instance **Pageable**, Si on a besoin que d'un tri, on ajoute un paramètre **Sort** à la méthode

```
List<User> findByLastname(String lastname, Sort sort);
```

# Paging and Sorting



- **Sort** et **Pageable** doivent être différent de null si l'on ne veut pas
  - de trie: **Sort.unsorted()**
  - de pagination: **Pageable.unpaged()**
- à la place de **Sort** et **Slice**, on peut retourner aussi une liste

```
List<User> findByLastname(String lastname, Pageable pageable);
```

# Limiter les résultats de la requête



- On peut limiter les résultat d'une requête en utilisant **first** ou **top**(identique)
- On peut aussi ajouter un valeur numérique pour spécifier le nombre maximum de résultat retourné  
sinon par défaut le nombre d'élément retourné est par défaut égal à 1

```
User findFirstByOrderByLastnameAsc();  
User findTopByOrderByAgeDesc();  
Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);  
Slice<User> findTop3ByLastname(String lastname, Pageable pageable);  
List<User> findFirst10ByLastname(String lastname, Sort sort);  
List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

# Procédure stockée

- Une procédure stockée est une série d'instructions SQL désignée par un nom qui est enregistré dans la base de donnée

**Avantage** → Réduire le trafic réseau entre l'application et la SGBD

```
CREATE PROCEDURE doubler( IN val_in INT, OUT val_out INT)
BEGIN
    SET val_out = val_in *2 ;
END
```

- Pour utiliser une procédure stockée, on utilise l'annotation **@Procedure**

```
@Procedure("doubler")    // appel explicite
Integer explicitlyNamedDoubler(Integer arg);
@Procedure                // appel implicite
Integer doubler(@Param("arg") Integer arg);
```

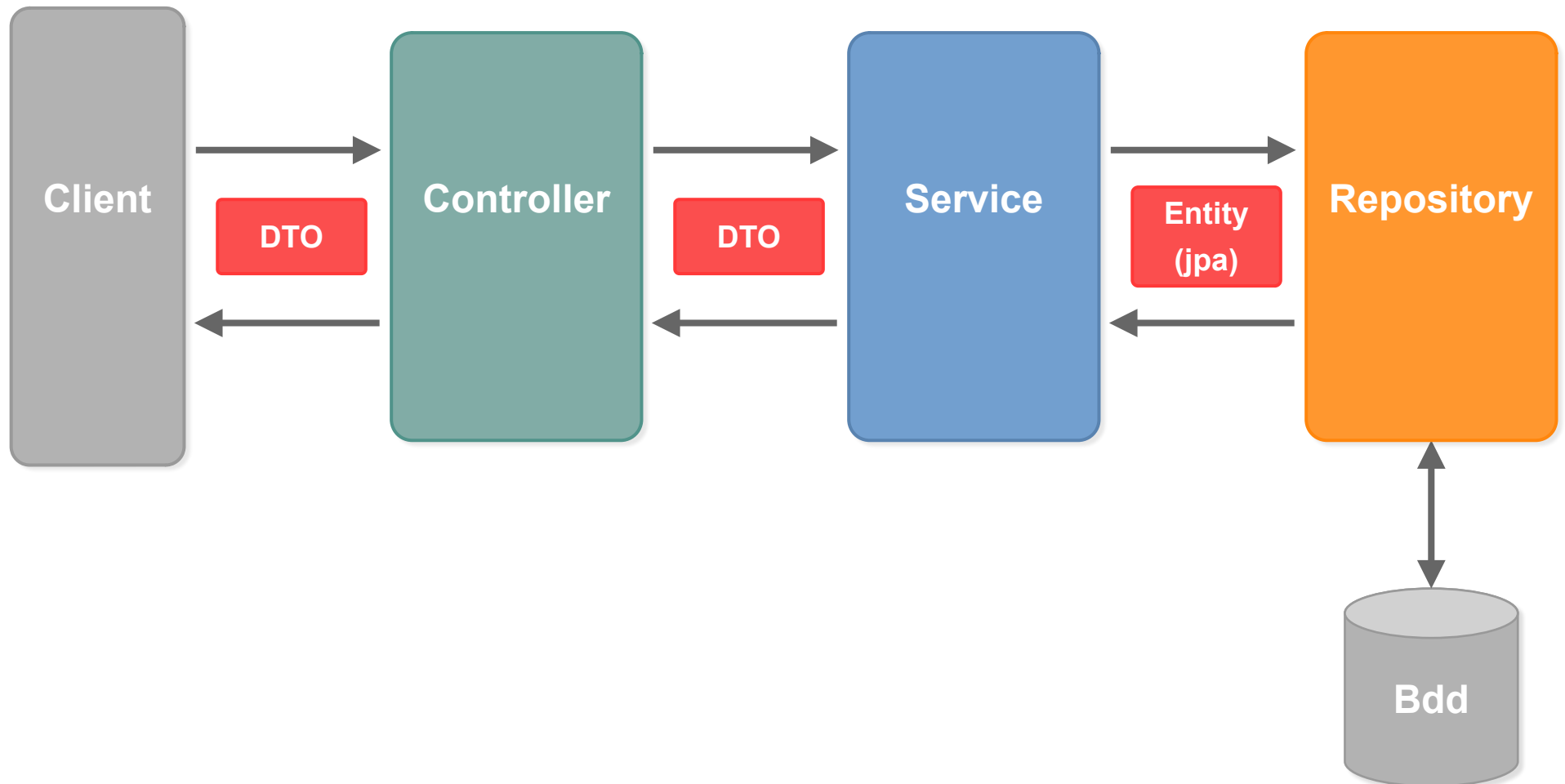
# DTO (Data Transfer Objet)



- Un DTO permet de créer des objets indépendants qui vont servir à transférer des données entre la couche de services et la couche des contrôleurs
- Le contrôleur ne doit manipuler que des DTO
- Le service retourne ou prend en paramètre des DTO et sera en charge de la transformation pour appeler les repositories
- La conversion **entity/dto** peut être réalisée manuellement ou en utilisant une librairie externe comme :

**ModelMapper** ou **MapStruct**

# DTO (Data Transfer Objet)



# Avantages de l'utilisation des DTO



- Transmettre des données avec plusieurs attributs en une seule fois du client au serveur, afin d'éviter plusieurs appels à un serveur distant
- Dissocier les modèles de persistance de l'API
- Ne pas exposer l'ensemble des attributs des entités et éviter l'utiliser des annotations comme `@JsonIgnore` ou `@XmlTransient` pour ne pas les sérialiser
- Éviter d'utiliser des annotations non liées à la persistance dans les entités
- Avoir différents DTO pour chaque version de l'API
- Avoir un contrôle sur les attributs que l'on reçoit lors de la création ou de la mise à jour d'une ressource

# ModelMapper

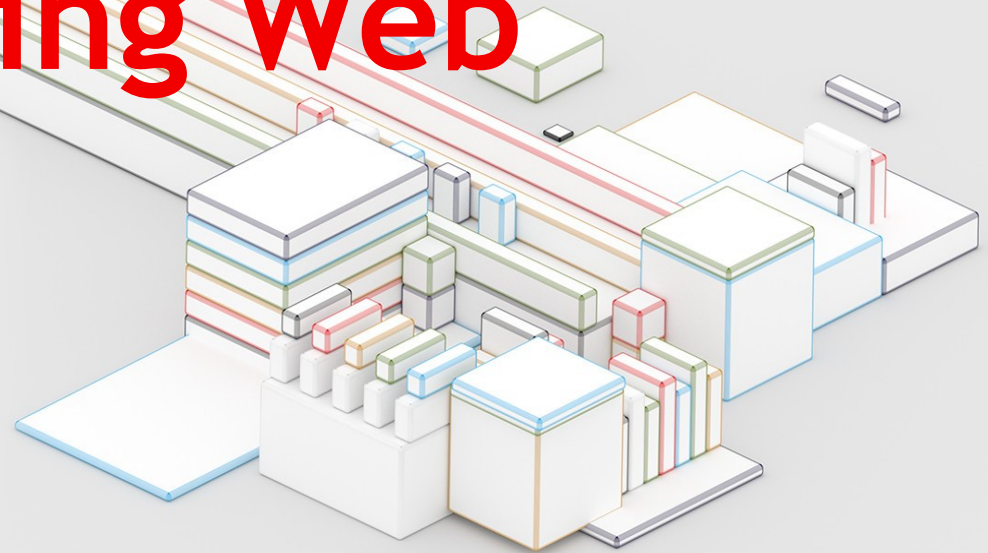


- Dépendance

```
<dependency>  
  <groupId>org.modelmapper</groupId>  
  <artifactId>modelmapper</artifactId>  
  <version>3.1.0</version>  
</dependency>
```



# Micro-services avec Spring Web



# WebService

---



- Un service Web peut-être vu comme une brique fonctionnelle, accessible depuis n'importe où sur le réseau
- On peut simplifier en disant que c'est une fonction à distance
- Cette fonction respecte des spécifications :
  - Spécification d'entrée : le format des données et des paramètres en entrée
  - Spécification en sortie : le format des données en sortie

Il existe deux grandes familles de WebServices

- **SOAP** (Simple Object Access Protocol)
  - On échange du XML
  - Repose sur un protocole SOAP dépendant de HTTP POST
  - Possède un WSDL comme spécification et contrat de service
- **REST**
  - On Échange du JSON, du XML, du Text, des données binaires ...
  - Repose sur un protocole HTTP POST, GET, PUT, DELETE et OPTIONS

# Architecture REST



- **REST** (Representational State Transfer)  
Un style d'architecture logicielle pour les systèmes distribués, tels que le www. Il est axé sur les ressources
- Repose sur l'architecture originelle du Web
- Utilise :
  - uniquement HTTP et un ensemble restreint d'actions :  
↳ **GET, POST, PUT, DELETE**
  - une URI (Uniform Resource Identifier) comme moyen d'interroger le service
  - des types MIME pour indiquer la nature des informations retournées par le service
- REST permet aux ressources d'avoir des représentations différentes : texte , XML , JSON ...

# Architecture REST



Selon son créateur, **Roy Fielding**, une architecture REST doit respecter :

- **Client-serveur** : on sépare le producteur du consommateur pour garantir un faible couplage
- **Sans état** : une requête doit contenir l'ensemble des informations nécessaires au serveur pour traiter la demande
- **Mise en cache** : la réponse serveur peut contenir des informations comme la date de création ou de validité de la réponse
- **Système par couches** : on accède à des ressources individuelles, une fonctionnalité complexe entraîne un ensemble d'appel au serveur

# Explication des Services REST (URL)



L'appel à un WebService se fait par une URL se composant :

1. de l'hôte

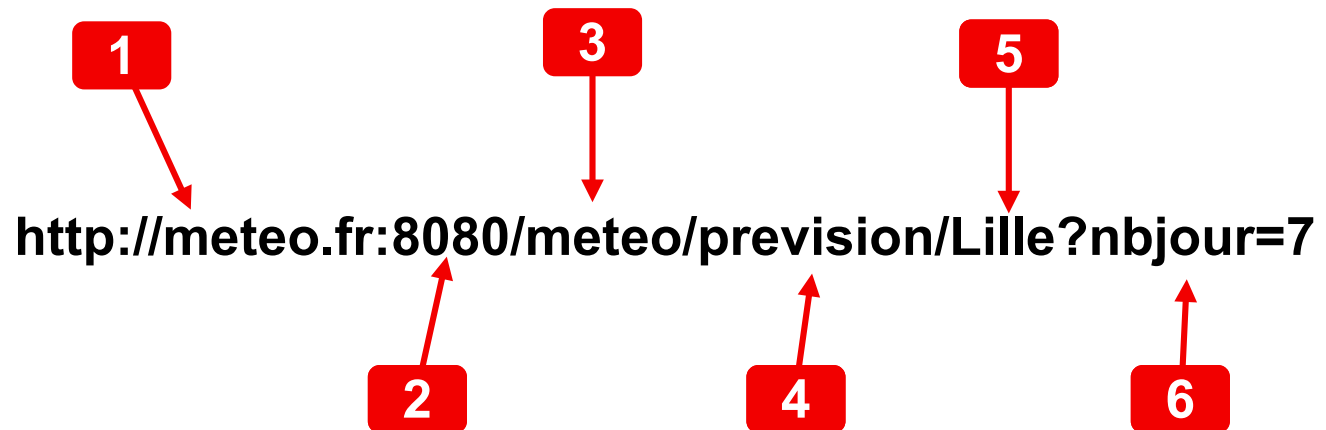
2. du port

3. d'un controller

4. d'un service

5. de paramètres de chemins

6. de paramètres de requêtes



On peut ajouter :

7. les entêtes

8. le corps du message

# Protocole HTTP simplifié



- Toutes les requêtes en **POST**, **PUT** et **DELETE** possèdent :
  - un hôte
  - un port
  - des paramètres de requêtes
  - des paramètres de chemin
  - des paramètres d'entêtes
  - un corps
- Les requêtes en **GET** ne possède pas de corps
- En général :
  - **GET** → sert à la consultation
  - **POST** → sert à la création
  - **PUT** → sert à la modification
  - **DELETE** → sert à la suppression

# Les codes de retours

- Les codes de retours d'un service indiquent l'état de la réponse d'un Webservice

Les codes commençant par :

- **1** : indiquent une information  
100 → Continue, 101 → Switching Protocols ...
- **2** : indiquent un succès  
200 → Ok, 201 → Created, 202 → Accepted ...
- **3** : indiquent une redirection  
302 → Found, 308 → Permanent Redirect ...
- **4** : indiquent une erreur  
403 → Forbidden, 404 → Not Found ...
- **5** : indiquent une erreur serveur  
500 → Internal Server Error



# Les codes de retours



- En général un service ne devrait pas retourner des codes commençant par **5**
- Un service en **GET** devrait retourner :
  - ↳ un code **OK (200)** ou **ACCEPTED (202)**
- Un service en **POST** devrait retourner :
  - ↳ un code **CREATED (201)** ou **OK (200)**
- Un service en **PUT** devrait retourner :
  - ↳ un code **ACCEPTED (202)** ou **OK (200)**
- Un service en **DELETE** devrait retourner :
  - ↳ un code **ACCEPTED (202)** ou **OK (200)**

# Les principaux clients REST

Des applications permettent de tester les web services :

- Pour tester les Web Services REST :



**PostMan** standalone ou plugin Chrome



**RESt** plugin Chrome ou plugin Firefox



**ARC** (Advanced REST Client) plugin Chrome



**SOAP UI** (limité)



**CURL** en ligne de commande

- Pour tester les Web Services SOAP :



**SOAP UI**

# Contrôleur REST



- Annoter le contrôleur avec **@RestController**
  - Évite de spécifier **@ResponseBody** sur chacune des méthodes du contrôleur
  - Format de la réponse : données au format JSON, XML possible
- Dépendance : jackson-databind

Conversion automatique des données du bean au format JSON

# Contrôleur REST



- **Lecture**
  - @GetMapping
- **Création**
  - @PostMapping
  - Données dans @RequestBody
- **Mise à jour**
  - @PutMapping
  - Données dans @RequestBody
  - Id en PathVariable
- **Suppression**
  - @DeleteMapping
  - Id en PathVariable

# Exemple de contrôleur REST

```
@RestController
@RequestMapping("trainings") //@Path
public class WSTrainingController {
    @Autowired
    private TrainingDao trainingDao;
    @GetMapping(value = "/json", produces = "application/json")
    public List<Training> listAll() {
        return trainingDao.findAllBasicInfosTrainings();
    }
    @GetMapping(value = "/xml", produces = "application/xml")
    public List<Training> listAllXml() {
        return trainingDao.findAllBasicInfosTrainings();
    }
    @GetMapping(value =("/{id}", produces = "application/json")
    public Training find(@PathVariable int id) {
        return trainingDao.findBasicInfosTrainingById(id);
    }
    @PostMapping(value = "/insert", consumes = "application/json")
    public void find(@RequestBody Training training) {
        trainingDao.persist(training);
    }
}
```

# RequestMapping



Peut être appliqué sur une méthode ou un contrôleur

- URL :
  - `@RequestMapping("/users")`
  - `@RequestMapping("/users", "/clients")`
- URI templates :
  - `@RequestMapping("/users/{userId}")`
  - `@RequestMapping("/users/{userId:[0-9]++}")`
- Méthodes HTTP :
  - `@RequestMapping(method={RequestMethod.GET})`
  - `@RequestMapping(method={RequestMethod.GET, ...})`

# RequestMapping



- Paramètres :
  - `@RequestMapping(params="id=8")`
  - `@RequestMapping(params={"id=8", "name=DOE"})`
  - `@RequestMapping(params="id")`
- Entêtes :
  - `@RequestMapping(headers="host=127.0.0.1")`
- Consommation/production :
  - `@RequestMapping(consumes=MediaType.APPLICATION_JSON_VALUE)`
  - `@RequestMapping(produces=MediaType.APPLICATION_JSON_VALUE)`

# PathVariable



## Identifier un élément de l'URI

- Variable nommée

```
@RequestMapping("/users/{id}")
public String handleRequest (@PathVariable("id") String userId, Model map)
{
    return "my-page";
}
```

- Variable avec nommage implicite

```
@RequestMapping("/users/{id}")
public String handleRequest (@PathVariable String id, Model map) {
    return "my-page";
}
```

- Variables multiples

```
@RequestMapping("/users/{id}/adress/{adrId}")
public String handleRequest (@PathVariable ("id") String userId,
                             @PathVariable("adrId") String adrId, Model map) {
    return "my-page";
}
```



# PathVariable

- Variables multiples dans une map

```
@RequestMapping("/users/{id}/adress/{adrId}")  
public String handleRequest (@PathVariable Map<String, String> vMap, Model m){  
    return "my-page";  
}
```

- Variables ambiguës

```
@RequestMapping("/users/{id}")  
public String handleRequest(@PathVariable("id") String userId, Model m){  
    return "my-page";  
}  
  
@RequestMapping("/users/{name}")  
public String handleRequest2 (@PathVariable("name") String userName, Model m) {  
    return "my-page";  
}
```

↳ **Solution** : introduire des expressions régulières

```
@RequestMapping("/users/{id:[0-9]+}")  
public String handleRequest(@PathVariable("id") String userId, Model m{...}  
@RequestMapping("/users/{name:[A-Za-z]+}")  
public String handleRequest2 (@PathVariable("name") String uName, Model m){ }
```

## Identifier un paramètre de l'URL

- Paramètre nommé

```
@RequestMapping
public String employeeByDept(@RequestParam("dept") String deptName, Model m){
    return "my-page";
}
```

- Paramètre avec nommage implicite

```
@RequestMapping
public String employeeByDept (@RequestParam String dept, Model m) {
    return "my-page";
}
```

- Paramètres multiples

```
@RequestMapping
public String EmployeeByDept (@RequestParam("dept") String deptName,
                             @RequestParam("state") String stateCode, Model m) {
    return "my-page";
}
```

# RequestParam



- Paramètres multiples dans une map

```
@RequestMapping()  
public String handleRequest (@RequestParam Map<String, String> paramsMap,  
                             Model m) {  
    return "my-page";  
}
```

- Paramètres ambigus : params dans le RequestMapping

```
@RequestMapping(params = "dept")  
public String handleEmployeeRequestByDept(@RequestParam("dept")  
                                           String deptName, Model map) {  
    return "my-page";  
}  
  
@RequestMapping(params = "state")  
public String handleEmployeesRequestByArea (@RequestParam("state")  
                                             String state, Model map) {  
    return "my-page";  
}
```

# RequestParam



- Paramètre requis, valeur par défaut

```
@RequestMapping("/users")  
public String handleRequest(@RequestParam(value = "project", defaultValue="kara")  
                             String projectName, Model model){  
    return "my-page" ;  
}
```

- Conversion implicite

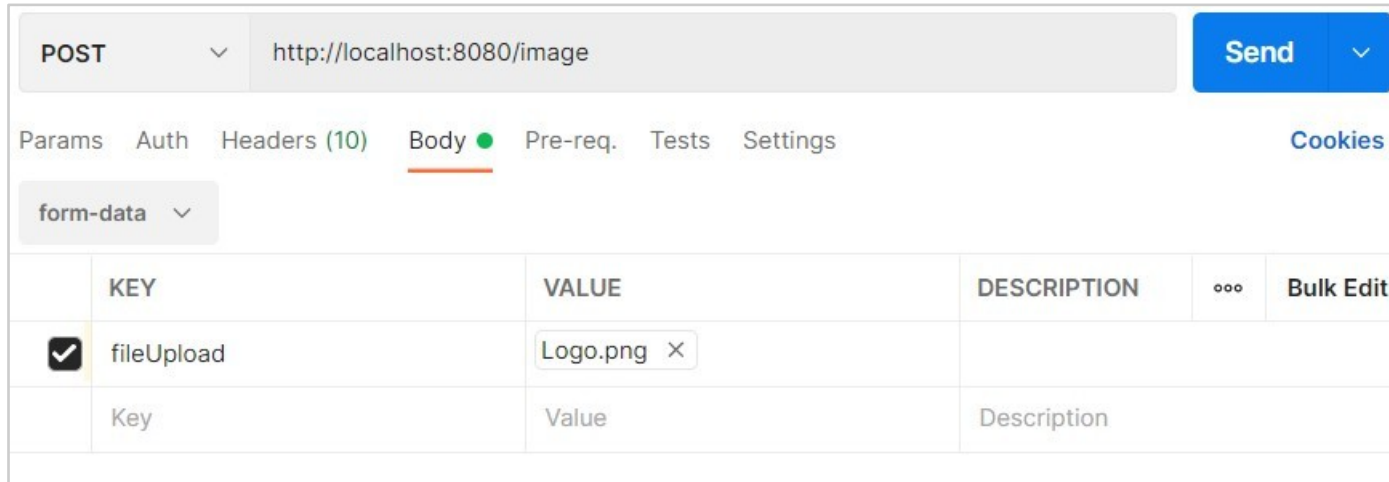
```
@RequestMapping()  
public String handleRequest (@RequestParam("nbitems") int nbItems, Model model) {  
    return "my-page";  
}
```

- Conversion de format de dates

```
@RequestMapping()  
public String handleRequest (@PathVariable("id") String employeeId,  
                             @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)  
                             @RequestParam("startDate") LocalDate startDate,  
                             Model model) {  
    return "my-page";  
}
```

# Envoi de fichier

```
@RequestMapping(value="/image",  
                method=RequestMethod.POST,  
                consumes="multipart/form-data")  
String uploadFile(@RequestParam(value = "fileUpload",  
                                required = true) MultipartFile file)  
  
throws IOException {  
    return "uploaded"+file.getBytes().length;  
}
```



POST http://localhost:8080/image Send

Params Auth Headers (10) Body Pre-req. Tests Settings Cookies

form-data

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	fileUpload	Logo.png			
	Key	Value	Description		

# Gestion des exceptions



Annotation d'une méthode **@ExceptionHandler** dans le contrôleur :

- Cette méthode sera invoquée lors du lancement d'une exception

```
@ExceptionHandler  
public String handleError() {...}
```

- Possibilité de spécifier l'exception à gérer

```
@ExceptionHandler({SQLException.class,DataAccessException.class})  
public String databaseError() {...}
```

- Cascade de gestionnaires d'exception
- Généraliser la gestion des exceptions dans une classe annotée **@ControllerAdvice**

# Codes retour HTTP

---



- Annoter la méthode du contrôleur avec @ResponseStatus
  - Classe **HttpStatus**

# Documentation du service web

---



- **OpenAPI Specification (Swagger)**

C'est une spécification qui définit un format standard pour documenter un service web REST

Elle permet de générer :

- Une documentation
- Le code de la documentation → client et serveur

- **Springdoc-openapi** → librairie qui supporte :

- OpenAPI 3
- Spring-boot
- Swagger-ui

Elle génère automatiquement la documentation de l'API en JSON, YAML et en HTML



# Documentation du service web



- **Dépendances Maven**

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <version>1.6.14</version>
</dependency>
```

- La documentation du service est disponible au travers de l'URL: **/v3/api-docs** et le fichier yaml : **/v3/api-docs.yaml**
- Pour générer l'HTML :
  - à partir du fichier yaml avec: **<https://editor.swagger.io/>**
  - Au travers de l'URL : **/swagger-ui/index.html?configUrl=/v3/api-docs/swagger-config**

# Documentation du service web



- **Pour générer le squelette du code**

- <https://github.com/swagger-api/swagger-codegen>

- L'outil en ligne commande

```
https://repo1.maven.org/maven2/io/swagger/codegen/v3/swagger-codegen-cli/3.0.29/swagger-codegen-cli-3.0.29.jar
```

```
java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar generate \
```

```
    -i URL_du_service \
```

```
    -l Langage \
```

```
    -o dossier_cible
```

```
java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar generate \
```

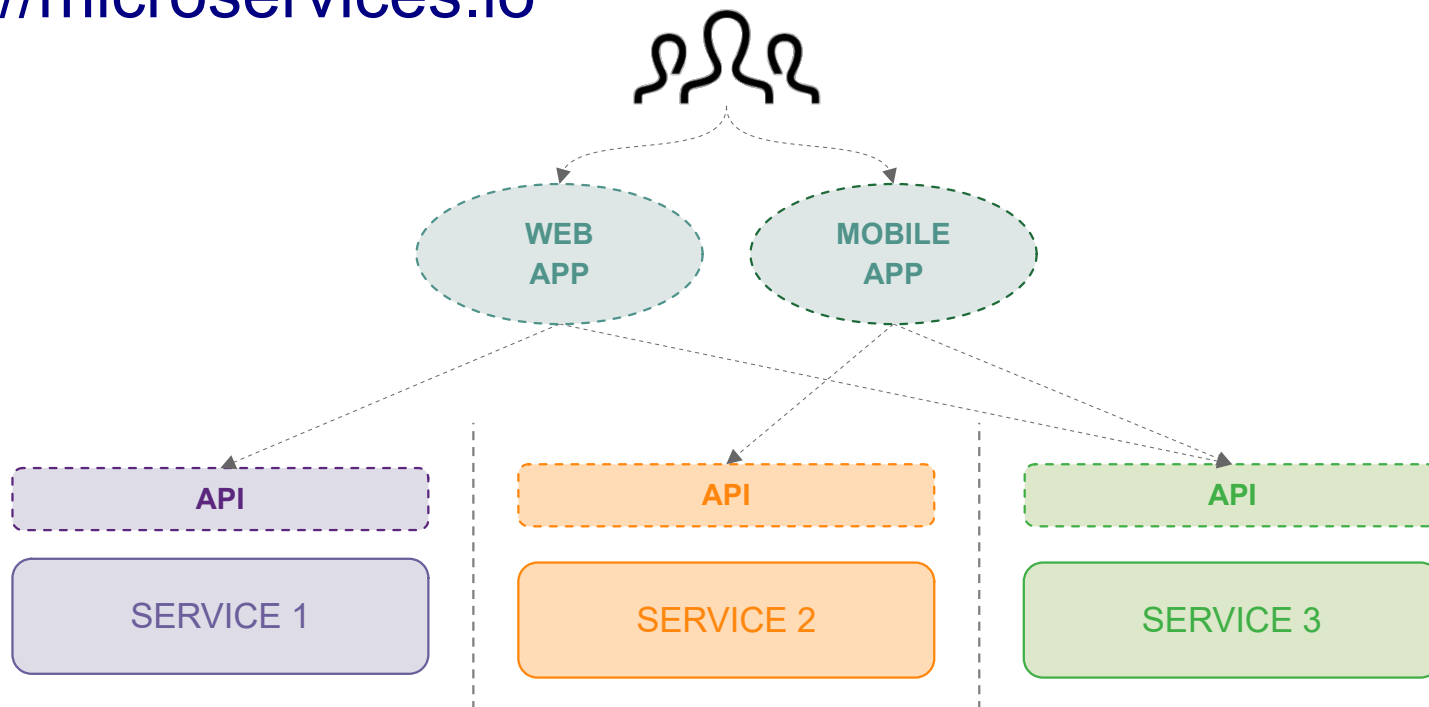
```
    -i https://petstore.swagger.io/v2/swagger.json \
```

```
    -l java \
```

```
    -o samples/client/petstore/java
```

# Micro-services

- Découper une application monolithique en plusieurs applications indépendantes
- Un micro-service répond à un besoin métier particulier, par exemple : s'occuper de toute la gestion de la facturation
- <http://microservices.io>



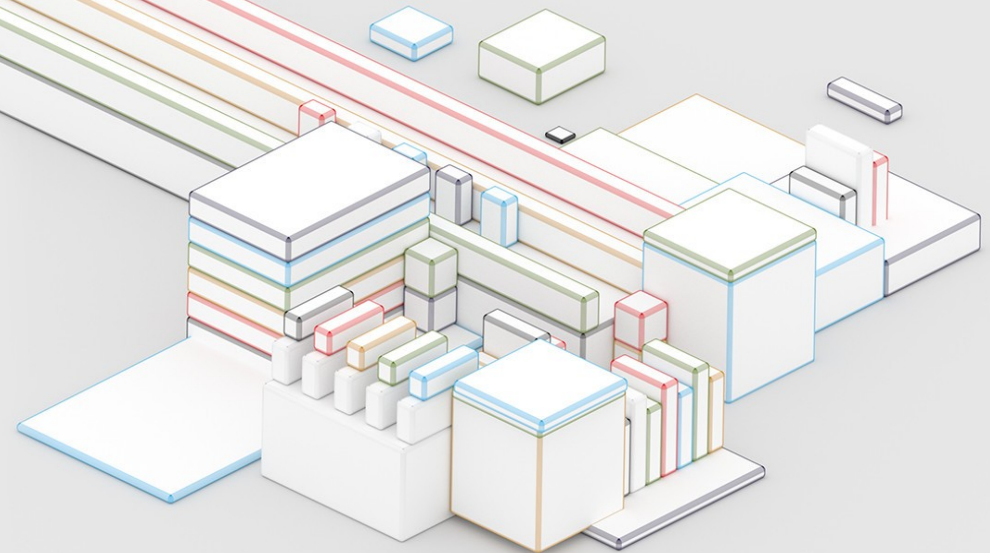
- **Avantages**

- Un besoin métier correspond à un projet : le code à l'intérieur de chaque projet est donc concentré à répondre à un seul besoin métier
- Possibilité de déployer fréquemment un service sans impacter les autres
- Possibilité de déployer des serveurs additionnels pour un seul service uniquement (scalabilité)
- Possibilité d'utiliser une technologie différente pour chaque service

- **Inconvénients :**

- Nécessite de passer plus de temps sur les tâches d'industrialisation et de déploiement car elles varient d'un service à l'autre
- Il faut pouvoir gérer les erreurs de communication entre les services avec la mise en place de systèmes comme un circuit breaker, du fallback, du retry ou autre
- Il faut tenir compte des soucis de dégradations de performances réseau : mettre en place un système d'élection, de synchronisation de données ...

# Sécuriser un service web



# Authentication et Protection des Services



- Les dépendances de Spring Boot sécurité :

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

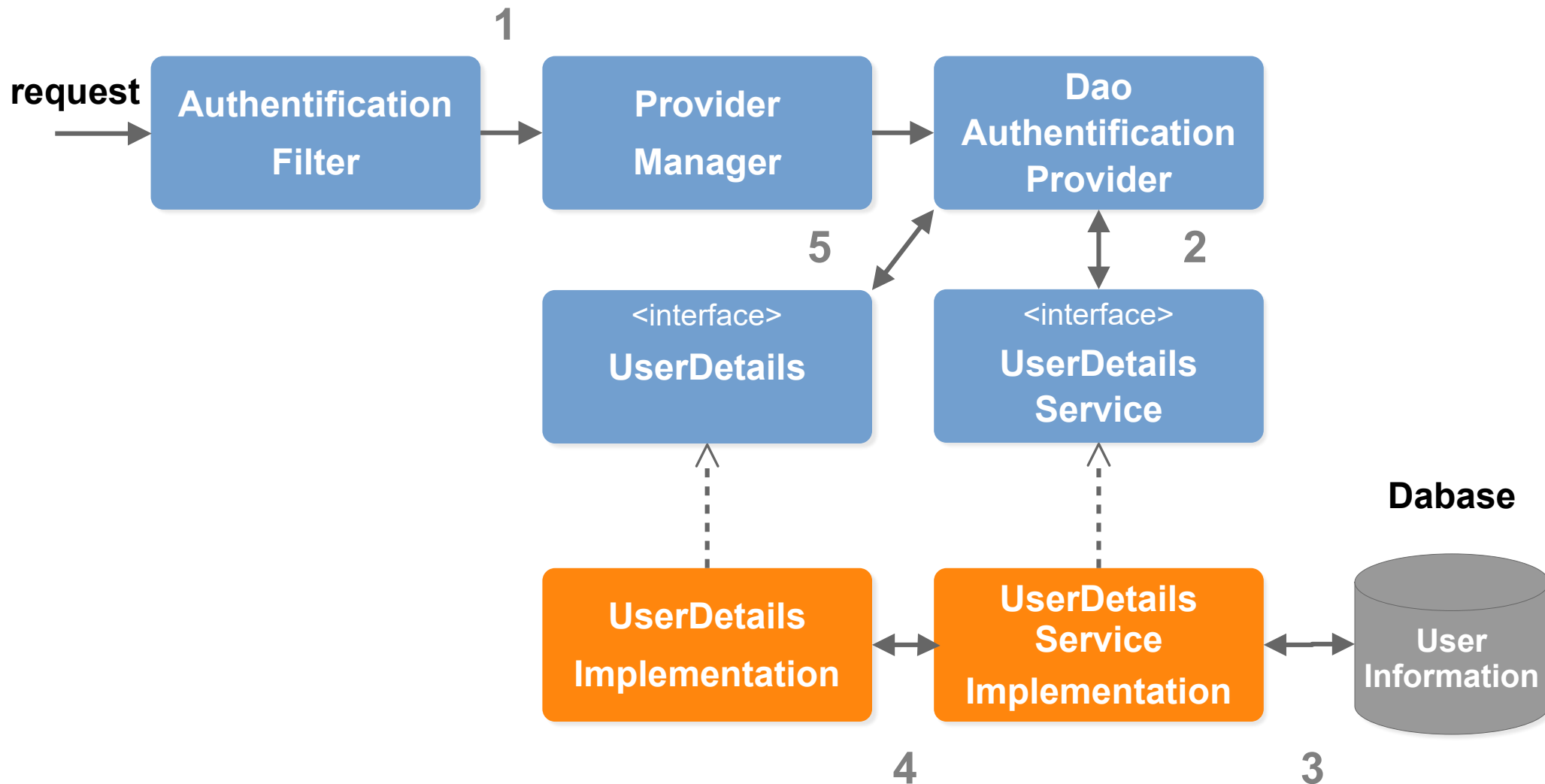
- Au démarrage, un code est donné les services sont alors automatiquement protégés
  - login : user
  - mot de passe : <voir la console>

```
Using default security password: 1da2694c-b3cd-4581-94c5-0b31cdef92ca
```

- Il est possible de spécifier un login et un mot de passe dans le fichier de propriétés

```
spring.security.user.name=training  
spring.security.user.password=training
```

# Authentication et Protection des Services



# Les utilisateurs



- L'interface **UserDetails**

C'est une interface Spring qui va permettre au framework d'automatiser quelques tâches

Elle définit un Utilisateur

```
public interface UserDetails extends Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
    String getPassword();  
    String getUsername();  
    boolean isAccountNonExpired();  
    boolean isAccountNonLocked();  
    boolean isCredentialsNonExpired();  
    boolean isEnabled();  
}
```



# Implémentation de l'interface

```
@Entity
@Table(name = "USER")
public class User implements Serializable , UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer userId;
    private String username;
    private String password;
    public boolean isAccountNonExpired() {return true;}
    public boolean isAccountNonLocked() { return true;}
    public boolean isCredentialsNonExpired() {return true;}
    public boolean isEnabled() {return true;}
    public Collection<? extends GrantedAuthority>
        getAuthorities() {
            return null;
        }
    // Setters et Getters
}
```

# Création du Repository associés aux utilisateurs



- On va créer un Repository pour stocker les utilisateurs

```
public interface UserRepository extends  
    JpaRepository<User, Integer> {  
    @Query(" select u from User u where u.username = ?1")  
    Optional<User> findUserWithName(String username);  
}
```

# Création du Service UserService



```
@Service
public class UserService implements UserDetailsService {

    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException
    {
        Objects.requireNonNull(username);
        User user = userRepository.findUserWithName(username)
            .orElseThrow(
                () -> new UsernameNotFoundException("User not found"));
        return user;
    }
}
```

# Fournisseur d'Authentification



```
public class AppAuthProvider extends DaoAuthenticationProvider {
    @Autowired
    UserService userDetailsService;

    @Override
    public Authentication authenticate(Authentication authentication) throws
        AuthenticationException {
        UsernamePasswordAuthenticationToken auth =
            (UsernamePasswordAuthenticationToken) authentication;
        String name = auth.getName();
        String password = auth.getCredentials().toString();
        UserDetails user = userDetailsService.loadUserByUsername(name);
        if (user == null) {
            throw new BadCredentialsException("l/p does not match for " +
                auth.getPrincipal());
        }
        return new UsernamePasswordAuthenticationToken(user, null,
            user.getAuthorities());
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return true;
    }
}
```

# Security Config

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserService userDetailsService;

    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.userDetailsService(userDetailsService);
    }

    @Bean
    public AuthenticationProvider getProvider() {
        AppAuthProvider provider = new AppAuthProvider();
        provider.setUserDetailsService(userDetailsService);
        return provider;
    }
    //... (suite)
}
```

# Security Config

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserService userDetailsService;
    @Autowired
    AuthenticationProvider provider;

    protected void configure(AuthenticationManagerBuilder auth)
        throws Exception {
        auth.userDetailsService(userDetailsService);
    }

    @Bean
    public AuthenticationProvider getProvider() {
        AppAuthProvider provider = new AppAuthProvider();
        provider.setUserDetailsService(userDetailsService);
        return provider;
    }

    //... (suite)
}
```

# Security Config

```
protected void configure(HttpSecurity http) throws Exception {  
    http.csrf()  
        .disable()  
        .authenticationProvider(getProvider())  
        .authorizeRequests()  
        .antMatchers("/monuments").authenticated()  
        .antMatchers(HttpMethod.GET, "/monument").authenticated()  
        .antMatchers(HttpMethod.POST, "/monument").authenticated()  
        .anyRequest().permitAll()  
        .and()  
        .httpBasic();  
}
```

# Security Config

```
protected void configure(HttpSecurity http) throws Exception {  
    http.csrf()  
        .disable()  
        .authenticationProvider(provider)  
        .authorizeRequests()  
        .antMatchers("/monuments").authenticated()  
        .antMatchers(HttpMethod.GET, "/monument").authenticated()  
        .antMatchers(HttpMethod.POST, "/monument").authenticated()  
        .anyRequest().permitAll()  
        .and()  
        .httpBasic();  
}
```



# Les Authority

- Un utilisateur est associé à plusieurs Authorities

```
@Entity
@Table(name = "USER")
public class User implements Serializable, UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer userId;
    private String username;
    private String password;

    @ManyToMany(fetch = FetchType.EAGER)
    private List<Authority> authority = new
        ArrayList<Authority>();

    //...
    @Override
    public Collection<? extends GrantedAuthority>
        getAuthorities() {
        return authority;
    }
}
```

# Une Authority

- Une Authority va consister en un ensemble d'action possible (SUPER / CREATE / READ / DELETE)

```
@Entity
@Table
public class Authority implements GrantedAuthority {
    @Id
    private String authority;

    public Authority() {
    }

    @Override
    public String getAuthority() {
        return authority;
    }

    public void setAuthority(String authority) {
        this.authority = authority;
    }
}
```

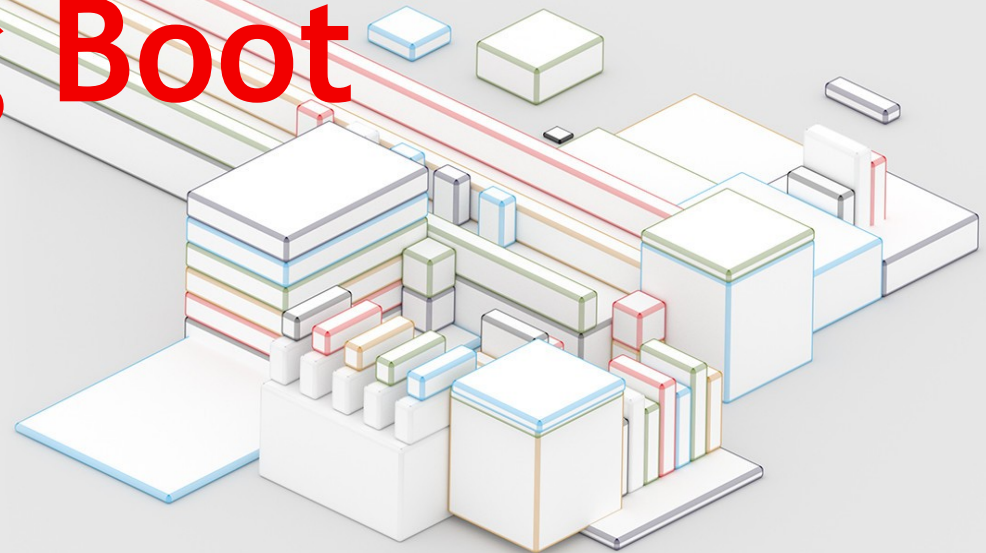
# Mapping des services avec les Autorities



- On peut mapper les Services avec les Authorities

```
protected void configure(HttpSecurity http) throws Exception {  
    http.csrf().disable()  
    .authenticationProvider(getProvider())  
    .authorizeRequests()  
    .antMatchers(HttpMethod.POST, "/image").authenticated()  
    .antMatchers("/monuments").hasAnyAuthority("DELETE", "SUPER")  
  
    .antMatchers(HttpMethod.GET,  
                  "/monument").hasAnyAuthority("READ", "SUPER")  
    .antMatchers(HttpMethod.POST,  
                  "/monument").hasAnyAuthority("WRITE", "SUPER")  
    .antMatchers(HttpMethod.DELETE,  
                  "/monument").hasAnyAuthority("DELETE", "SUPER")  
    .anyRequest().permitAll()  
    .and()  
    .httpBasic();  
}
```

# Tester une application Spring Boot



# Les Test sous Spring Boot



- La capacité de Spring Boot de se lancer avec son conteneur applicatif permet de faire une conception de test vraiment simple pour les développeurs
- Annotations pour l'initialisation du test:
  - **@RunWith** Indique qui va exécuter les tests unitaires
  - **@ContextConfiguration** Indique la classe à charger pour les tests d'intégrations
  - **@SpringBootTest** Indique la stratégie de choix du port
- Les Tests peuvent fonctionner avec leurs propres fichiers de propriétés

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

# Configuration de l'application



```
@RunWith(SpringRunner.class)
@ContextConfiguration(classes=Main.class)

// test avec le fichier de configuration des test
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
public class MonumentControllerTest {

// La classe de Test doit terminer par le suffixe Test

}
```

- |                |  |
|----------------|--|
| <b>@Test</b>   | Indique la méthode représentant le test                      |
| <b>@Before</b> | Indique la méthode exécutée avant chaque Test                |
| <b>@After</b>  | Indique la méthode exécutée après chaque Test                |
| <b>@Setup</b>  | Indique la méthode exécutée lors de l'initialisation du Test |

# Initialisation des Objets



- On peut créer les objets permettant d'initialiser la base de données et de requêter les services

```
@Autowired
MonumentRepository monumentRepository;
private RestTemplate restTemplate = new RestTemplate();

// Avant chaque Test on vide la base
@Before
public void emptyDatabase(){
    monumentRepository.deleteAll();
}
```

# Test d'un service

```
@Test
public void testServiceGet(){
    // Préparation du Test
    Monument m = new Monument();
    m.setNom("Tour Eiffel");
    m.setPosition(Arrays.asList(new Point(10.5,10.0)));
    m.setDescription("...");
    m.setId(1);
    m = monumentRepository.save(m);

    // Execution
    ResponseEntity<Monument> response = restTemplate.getForEntity(
        "http://localhost:9090/monument/"+m.getId(),Monument.class);
    Monument monument = response.getBody();

    // Vérification que l'on obtient les bons résultats
    assertEquals(HttpStatus.OK, response.getStatusCode());
    assertEquals("Tour Eiffel", monument.getNom());
}
```



# Mockage avec Mockito



- On veut créer un service générant un nombre aléatoire

```
@Service
public class RandomService {
    public int getNumber(int min, int max) {
        return (int)(Math.random()*(max-min))+min;
    }
}
```

- Le mockage consiste à remplacer l'implémentation faite du RandomService par une implémentation de Mockito
- Pour différents paramètres des méthodes nous spécifions un type de retour

# Test du Service Random

```
@ActiveProfiles("test")
@RunWith(SpringRunner.class)
@ContextConfiguration(classes = Main.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
public class SampleTest {
    @Test
    public void test() {
        RandomService random = Mockito.mock(RandomService.class);
        Mockito.when(random.getNumber(1, 50)).thenReturn(15);
        Mockito.when(random.getNumber(5, 15)).thenReturn(12);
        AssertEquals(12, random.getNumber(5, 15));
        AssertEquals(15, random.getNumber(1, 50));
    }
}
```

# Injection de Mock

- On crée un Bean avec Mockito
- Ce bean est injectable directement dans les tests

```
@Profile("test")
@Configuration
public class MockRandomService {
    @Bean
    @Primary
    public RandomService randomService() {
        return Mockito.mock(RandomService.class);
    }
}

@Autowired
RandomService randomService;
@Test
public void test2() {
    Mockito.when(randomService.getNumber(1,50)).thenReturn(15);
    Mockito.when(randomService.getNumber(5,15)).thenReturn(12);
    assertTrue(randomService.getClass().getName().contains("Mockito"));
    assertEquals(12,randomService.getNumber(5,15));
    assertEquals(15,randomService.getNumber(1,50));
}
```

# Injection de Service

- Un service utilisant le Service va aussi bénéficier du service Mocké pour les tests

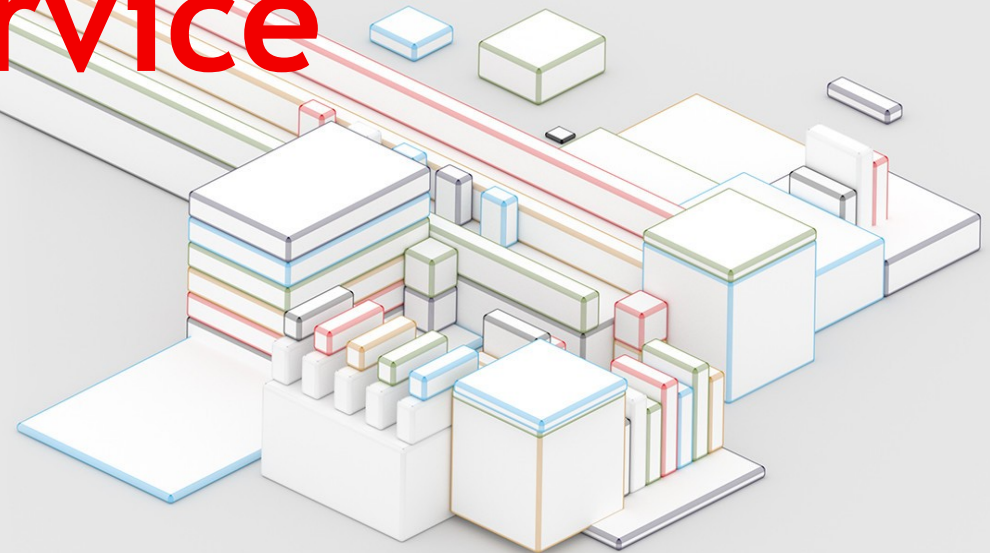
```
@Service
public class TirageService {
    @Autowired
    RandomService randomService;

    public int next() {
        return randomService.getNumber(1, 50);
    }
}

@Autowired
RandomService randomService;
@Autowired
TirageService TirageService;

@Test
public void test3() {
    Mockito.when(randomService.getNumber(1, 50)).thenReturn(15);
    Mockito.when(randomService.getNumber(5, 15)).thenReturn(12);
    assertEquals(15, TirageService.next());
}
```

# Mettre en production un service



- **@EnableScheduling** permet d'activer la prise en charge de la planification des tâches
- **@Scheduled** est une annotation utilisée pour configurer une planification sur un méthode sans paramètre et sans type de retour
- Attribut de l'annotation @Scheduled
  - fixedDelay** → la méthode est exécutée avec une période fixe en milliseconde entre la fin de la dernière invocation et le début de la prochaine  
`@Scheduled(fixedDelay=1000)`
  - **fixedRate** → la méthode est exécutée avec une période fixe en milliseconde entre les invocations

# Planification



- **initialDelay** → Le nombre de millisecondes à retarder avant la première exécution de `fixedRate()` ou `fixedDelay()`
- **cron** → comprend 6 champs : seconde, minute, heure, le jour du mois, le mois et le jour de la semaine

`@Scheduled(cron = "0 15 10 15 * ?")`

↳ exécuté chaque mois le 15 à 10h 15

\* quelques, \*/X tous les X, ? aucune valeur spécifique

- **zone** → Un fuseau horaire pour lequel l'expression cron Par défaut le fuseau horaire local

`zone = "Europe/Paris"`

# Planification



- On peut utiliser des valeurs spécifiées dans le fichier de propriété

```
@Scheduled(fixedDelayString = "${fixedDelayMs}")
```

```
@Scheduled(fixedRateString = "${fixedRateMs}")
```

```
@Scheduled(cron = "${cronExp}")
```



# Gestion du cache

- Un cache permet d'économiser les temps de traitement ou d'accès en base en stockant la réponse et en redonnant la réponse précédemment donnée
- On utilisera l'implémentation de EhCache
- 1. Ajouter le module de gestion du cache (starter) et inclure l'implémentation du cache

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-cache</artifactId>  
</dependency>  
<dependency>  
  <groupId>net.sf.ehcache</groupId>  
  <artifactId>ehcache</artifactId>  
</dependency>
```

# Gestion du cache

## 2. faire une référence au fichier configurant le cache

### **application.properties**

```
cache.jcache.config=classpath:ehcache.xml
```

### **ehcache.xml (ressources)**

```
<?xml_version="1.0" encoding="UTF-8"?>
<ehcache>
  <defaultCache eternal="true"
    maxElementsInMemory="100"
    overflowToDisk="false" />
  <cache name="test" maxElementsInMemory="1000"
    timeToIdleSeconds="10" timeToLiveSeconds="10"
    overflowToDisk="false"
    memoryStoreEvictionPolicy="LRU" />
</ehcache>
```

## 3. Vérifier que le cache fonctionne en injectant le cache manager

```
@Autowired
private CacheManager cacheManager;

@RequestMapping(value="/monuments",method=RequestMethod.GET)
@ResponseBody
Page<Monument> all(Pageable p) {
    cacheManager.getCache("test");
    return monumentRepository.findAll(p);
}
```

# Gestion du cache



## 4. Tester la gestion du cache sur une méthode

```
@Repository
public interface MonumentRepository extends
CrudRepository<Monument, Integer> {
    @Query("FROM Monument m WHERE m.description like %:desc%")
    List<Monument> findByDescription(String desc);

    @Cacheable("test")
    @Query("SELECT m FROM Monument m")
    Page<Monument> findAll(Pageable p);
}
```

# Gestion des logs



- Au démarrage de l'application on peut déterminer le niveau de Log :

```
java -jar -Dserver.port=9090 demo.jar -error
```

**logging.file** permet de spécifier l'adresse d'un fichier de Log

**logging.path** permet de spécifier l'adresse d'un répertoire de Log

**logging.level.<logger-name>=<level>**

permet de spécifier un niveau de Log pour un logger particulier

```
logging.level.org.springframework.web=ERROR
```

```
logging.level.org.hibernate=ERROR
```

```
logging.level.app.Main=ERROR
```

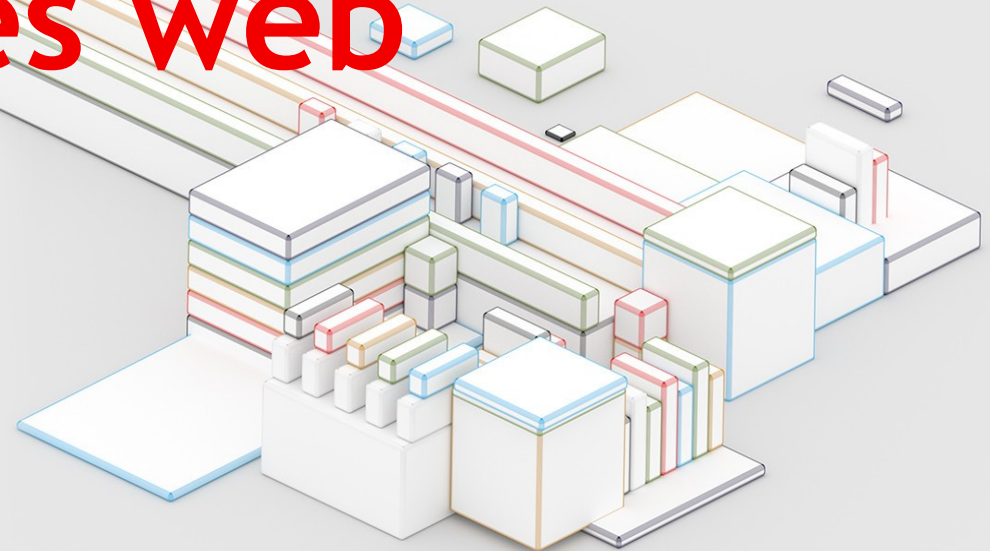
# Spring Boot Actuator

- Ce sont des services déjà conçu permettant la récupération d'information

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

- Par défaut, seul /actuator/health et /actuator/info sont activés
- Il est possible de tous les activer avec la propriété :  
management.endpoints.web.exposure.include=\*
- Il est possible d'inclure des services spécifiques (ou d'inclure tout sauf un ensemble de services)
- Exemple, voir la configuration des loggers :  
<http://localhost:8080/actuator/loggers>

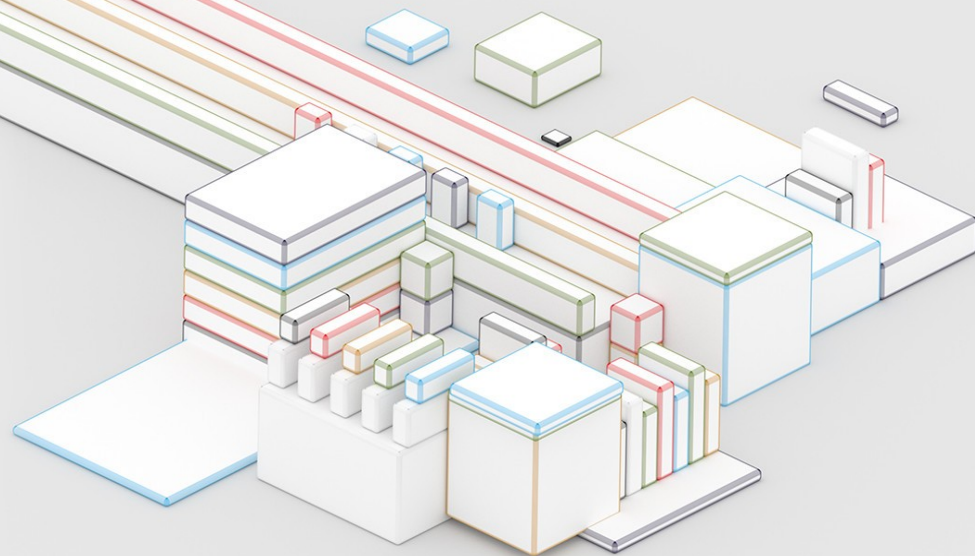
# Écrire des clients de services web





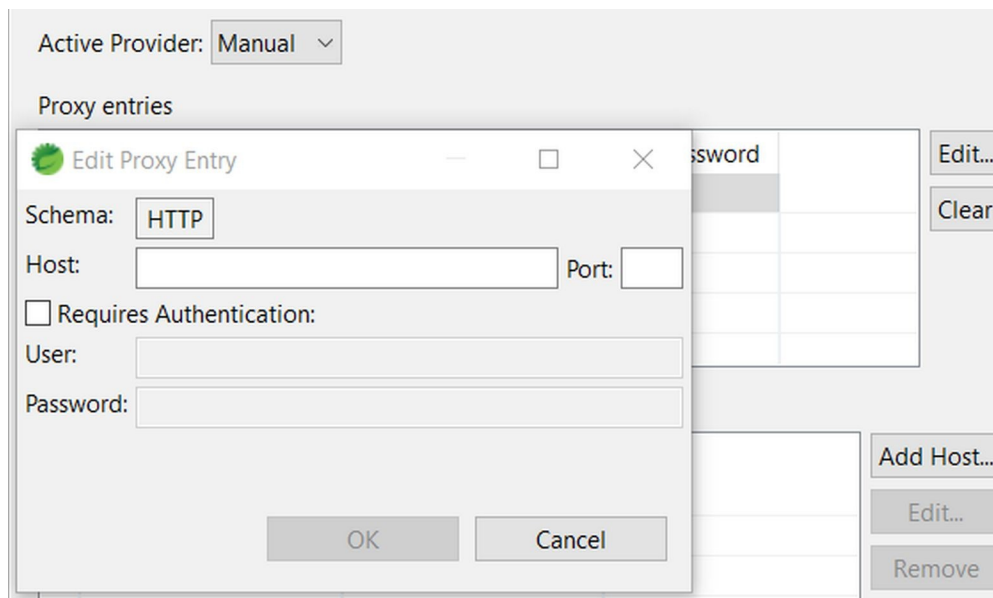


# Annexe



# Configuration Proxy Eclipse

- Dans windows → préférence
  - filtre sur **Network Connections**
    - Action provider : Manuel
    - Sélectionner : Http → Edit
    - Remplir : l'hôte du serveur proxy, le numéro de port, le nom d'utilisateur et le mot de passe



# Configuration Proxy Maven



- Dans settings.xml

```
<proxies>
  <proxy>
    <id>example-proxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.example.com</host>
    <port>8080</port>
    <username>proxyuser</username>
    <password>password</password>
    <nonProxyHosts>www.google.com|*.example.com</nonProxyHosts>
  </proxy>
</proxies>
```

- Dans eclipse :
  - windows→préférences
  - filtre sur Maven
  - User settings et sélectionner le fichier settings.xml

# Mot clef requête repository

Mot-clés	Exemples
<b>Distinct</b>	<code>findDistinctByLastnameAndFirstname</code>
<b>And</b>	<code>findByLastnameAndFirstname</code>
<b>Or</b>	<code>findByLastnameOrFirstname</code>
<b>Is, Equals</b>	<code>findByFirstnameIs, findByFirstnameEquals</code>
<b>Between</b>	<code>findByStartDateBetween</code>
<b>LessThan</b>	<code>findByAgeLessThan</code>
<b>LessThanEqual</b>	<code>findByAgeLessThanEqual</code>
<b>GreaterThan</b>	<code>findByAgeGreaterThan</code>
<b>GreaterThanEqual</b>	<code>findByAgeGreaterThanEqual</code>
<b>After</b>	<code>findByStartDateAfter</code>
<b>Before</b>	<code>findByStartDateBefore</code>
<b>IsNull, Null</b>	<code>findByAge(Is)Null</code>
<b>IsNotNull, NotNull</b>	<code>findByAge(Is)NotNull</code>
<b>Like</b>	<code>findByFirstnameLike</code>

# Mot clef requête repository



Mot-clés	Exemples
<b>NotLike</b>	<code>findByFirstnameNotLike</code>
<b>StartingWith</b>	<code>findByFirstnameStartingWith</code>
<b>EndingWith</b>	<code>findByFirstnameEndingWith</code>
<b>Containing</b>	<code>findByFirstnameContaining</code>
<b>OrderBy</b>	<code>findByAgeOrderByLastnameDesc</code>
<b>Not</b>	<code>findByLastnameNot</code>
<b>In</b>	<code>findByAgeIn(Collection ages)</code>
<b>NotIn</b>	<code>findByAgeNotIn(Collection ages)</code>
<b>True</b>	<code>findByActiveTrue()</code>
<b>False</b>	<code>findByActiveFalse()</code>
<b>IgnoreCase</b>	<code>findByFirstnameIgnoreCase()</code>

# JSON

- **JSON** signifie **J**ava**S**cript **O**bject **N**otation  
Il a été créé par Douglas Crockford entre 2002 et 2005  
Il est décrit par les normes: [RFC 8259](#) et [ECMA-404](#)  
Il a pour type MIME : **application/json**
- JSON est :
  - un format textuel
  - un format d'échange de données léger
  - auto-descriptif et facile à comprendre
  - indépendant du langage
- La syntaxe est dérivée de la syntaxe de notation d'objet JavaScript

# Syntaxe de JSON

- Les données sont dans des paires nom/valeur  
le nom est entre guillemet " et il est séparé de la valeur par :

```
"nom": "John"
```

- Les données sont séparés par des virgules ,
- Les objets sont placés entre accolade { }  
le nom des attributs sont placées entre guillemet "

```
{ "prenom": "John", "nom": "Doe", "age": 30 }
```

- Les tableaux sont placés entre crochets [ ]

```
[ "John", "Jane", "Alan" ]
```

- Les valeurs doivent être l'un des types de données suivants :
  - une **chaîne de caractère** : placé entre guillemet " → "John"

# JSON

- un **nombre** : entier ou à virgule flottante
- un **booléens** : `true` ou `false`
- un **objet** → `"employee": { "prenom": "John", "nom": "Doe" }`
- Un **tableau** → `"employees": [ "John", "Jane", "Alan" ]`
- `null`

```
{  
  "name": "test",  
  "description": "c'est un test",  
  "publik": true,  
  "visible": true,  
  "note": [ 9.5, 10.0, 15.0 ]  
}
```



# Bannière : couleur et style



- **Valeurs pour AnsiColor et AnsiBackground**

BLACK	BRIGHT_BLACK	DEFAULT
BLUE	BRIGHT_BLUE	
CYAN	BRIGHT_CYAN	
GREEN	BRIGHT_GREEN	
MAGENTA	BRIGHT_MAGENTA	
RED	BRIGHT_RED	
WHITE	BRIGHT_WHITE	
YELLOW	BRIGHT_YELLOW	

- **Valeurs pour AnsiStyle**

BOLD	FAINT
ITALIC	NORMAL
UNDERLINE	

# Bannière : couleur et style

- Valeurs numériques pour AnsiColor et AnsiBackground

000	000	001	001	002	002	003	003	004	004	005	005	006	006	007	007
008	008	009	009	010	010	011	011	012	012	013	013	014	014		015
232	232	233		234	234	235	235	236	236	237	237	238	238	239	239
240	240	241	241	242	242	243	243	244	244	245	245	246	246	247	247
248	248	249	249	250	250	251	251	252	252	253	253	254	254		255

016	016	017	017	018	018	019	019	020	020	021	021	124	124	125	125	126	126	127	127	128	128	129	129
022	022	023	023	024	024	025	025	026	026	027	027	130	130	131	131	132	132	133	133	134	134	135	135
028	028	029	029	030	030	031	031	032	032	033	033	136	136	137	137	138	138	139	139	140	140	141	141
034	034	035	035	036	036	037	037	038	038	039	039	142	142	143	143	144	144	145	145	146	146	147	147
040	040	041	041	042	042	043	043	044	044	045	045	148	148	149	149	150	150	151	151	152	152	153	153
046	046	047	047	048	048	049	049	050	050	051	051	154	154	155	155	156	156	157	157	158	158	159	159
052	052	053	053	054	054	055	055	056	056	057	057	160	160	161	161	162	162	163	163	164	164	165	165
058	058	059	059	060	060	061	061	062	062	063	063	166	166	167	167	168	168	169	169	170	170	171	171
064	064	065	065	066	066	067	067	068	068	069	069	172	172	173	173	174	174	175	175	176	176	177	177
070	070	071	071	072	072	073	073	074	074	075	075	178	178	179	179	180	180	181	181	182	182	183	183
076	076	077	077	078	078	079	079	080	080	081	081	184	184	185	185	186	186	187	187	188	188	189	189
082	082	083	083	084	084	085	085	086	086	087	087	190	190	191	191	192	192	193	193	194	194	195	195
088	088	089	089	090	090	091	091	092	092	093	093	196	196	197	197	198	198	199	199	200	200	201	201
094	094	095	095	096	096	097	097	098	098	099	099	202	202	203	203	204	204	205	205	206	206	207	207
100	100	101	101	102	102	103	103	104	104	105	105	208	208	209	209	210	210	211	211	212	212	213	213
106	106	107	107	108	108	109	109	110	110	111	111	214	214	215	215	216	216	217	217	218	218	219	219
112	112	113	113	114	114	115	115	116	116	117	117	220	220	221	221	222	222	223	223	224	224	225	225
118	118	119	119	120	120	121	121	122	122	123	123	226	226	227	227	228	228	229	229	230	230	231	231



**Plus d'informations sur <http://www.dawan.fr>**

**Contactez notre service commercial au  
09.72.37.73.73 (prix d'un appel local)**