

Architecture Microservices avec Spring Boot

Christophe Fontaine
cfontaine@dawan.fr

16/01/2023

Objectifs



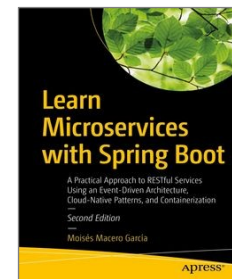
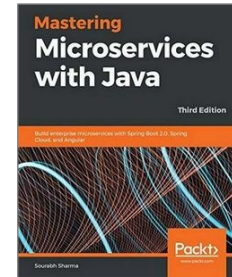
- Maîtriser l'utilisation de Spring Boot pour la construction de web services

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 - 2nd edition - Novembre 2020
- **Spring in Action**
Craig Walls
Manning - 6nd edition - Janvier 2022
- **Spring Microservices in Action**
John Carnell, Illary Huaylupo Sánchez
Manning - 2nd 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**

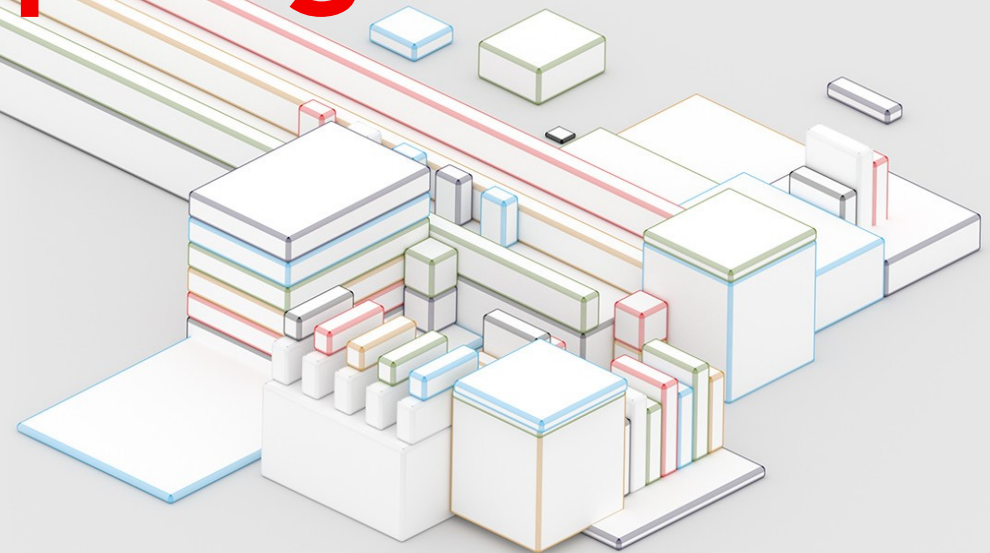
<https://books.sonatype.com/mvnref-book/reference/index.html>

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)

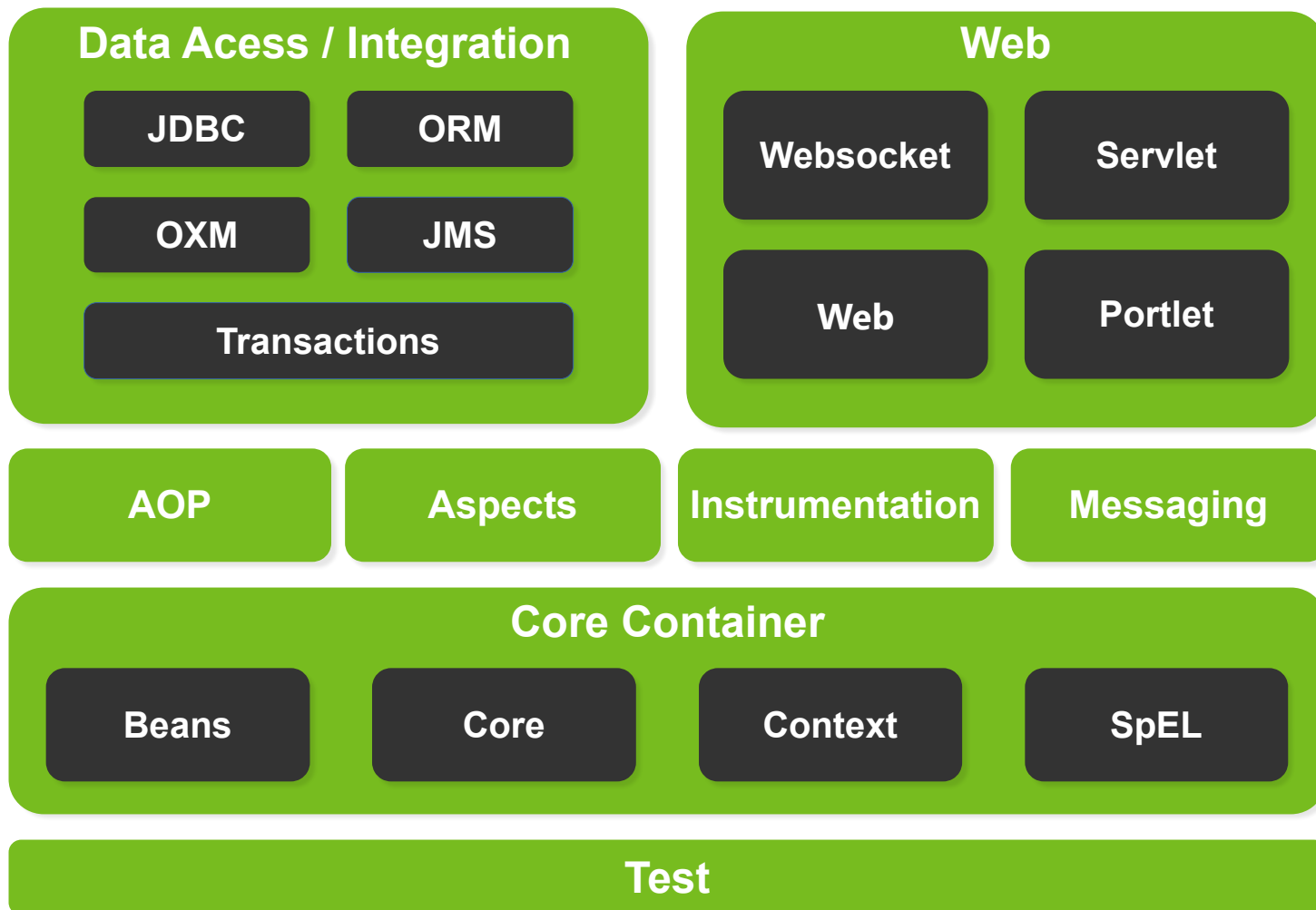
2022

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



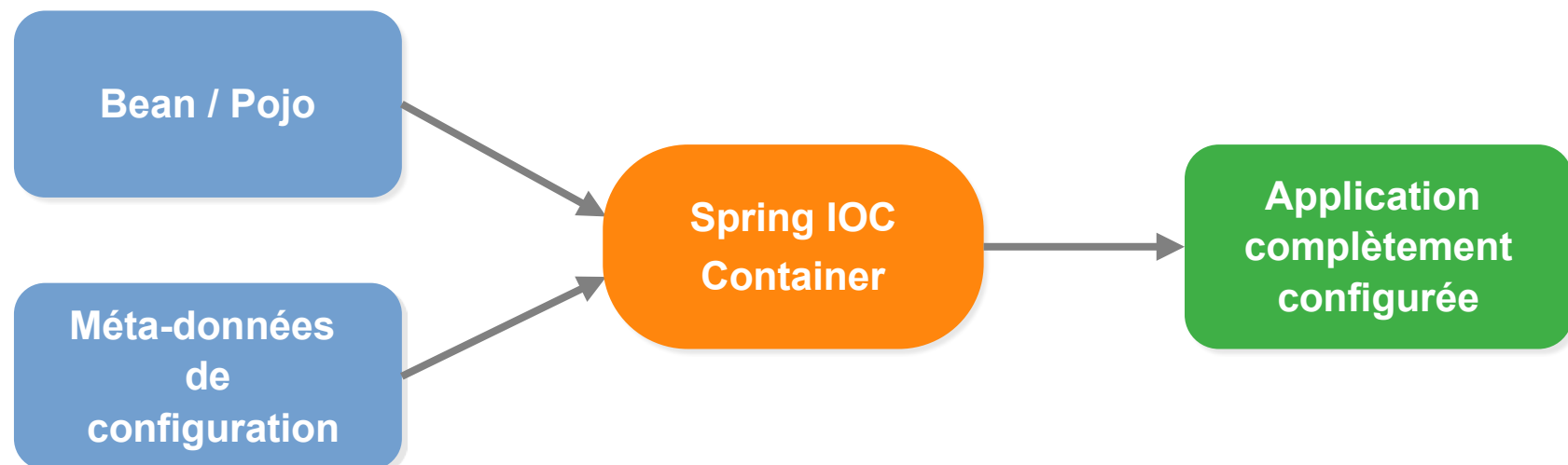
- Patron d'architecture qui fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application mais du framework
- Avec l'IoC, le framework prend en charge l'exécution principale du programme, il coordonne et contrôle l'activité de l'application
- IoC 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)

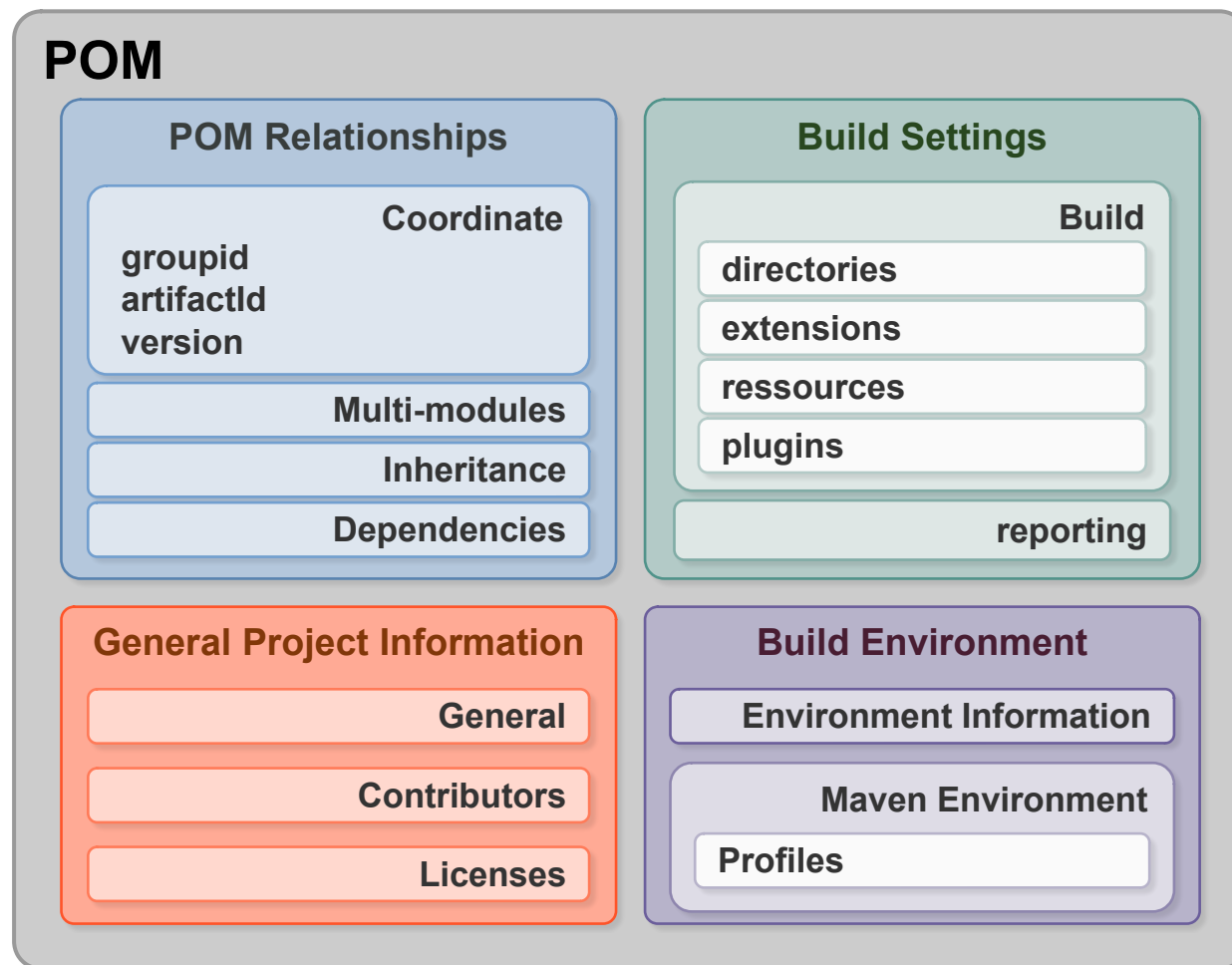


- Maven identifie de manière unique un projet avec :
 - **groupId** identifiant arbitraire du groupe de projet habituellement basé sur le package Java (sans espace, ni :)
 - **artifactId** nom arbitraire du projet (sans espaces, ni :)
 - **version** version du projet : **Major.Minor.Maintenance** si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

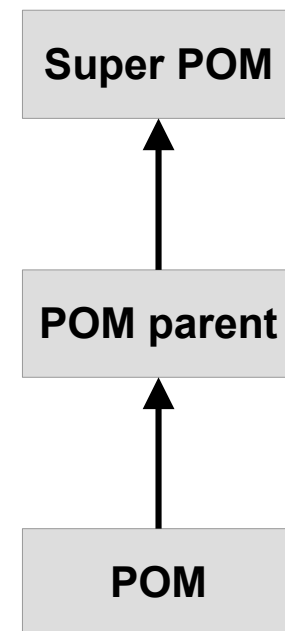
Le POM se compose de 4 catégories de description et de configuration



POM - Héritage

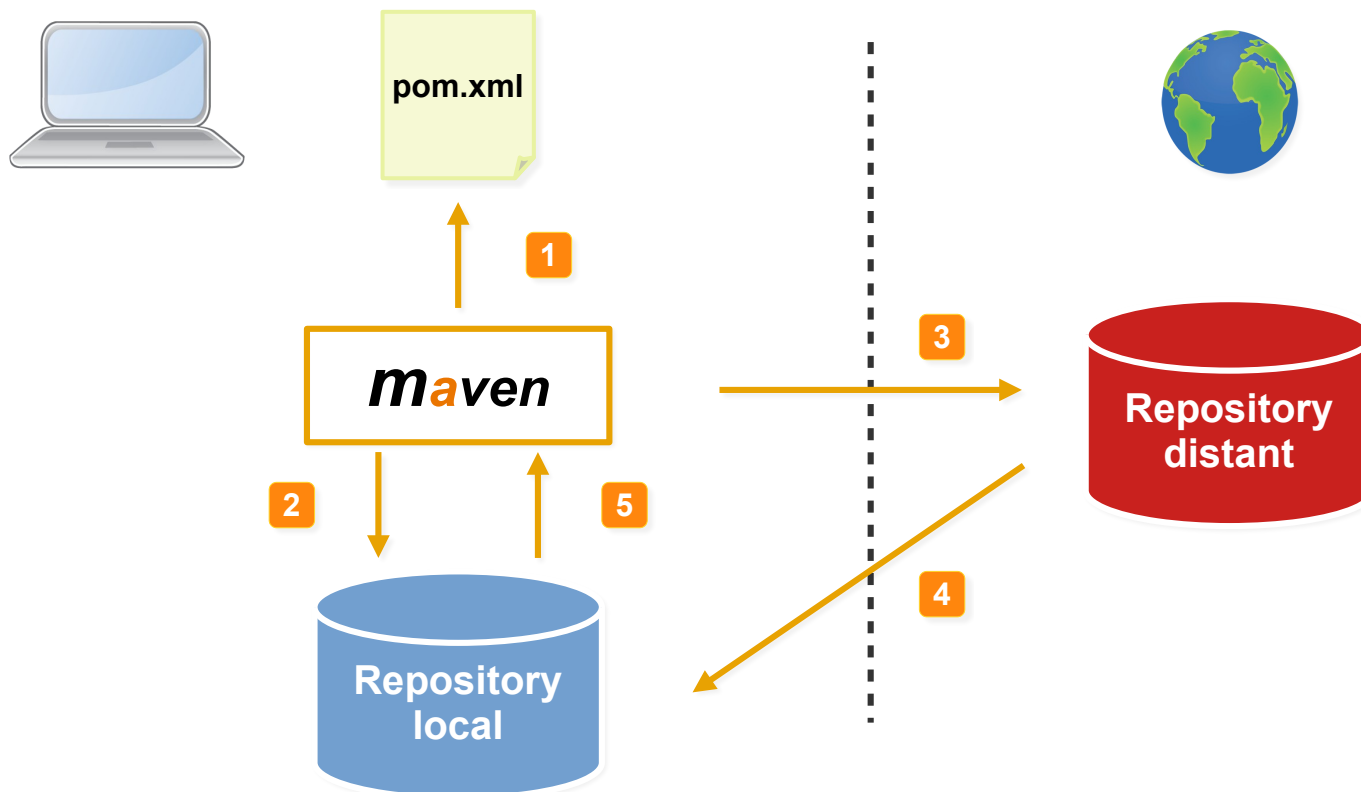
- Les fichiers POM peuvent hériter d'une configuration : groupId, version, configuration, dépendances ...

```
<?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

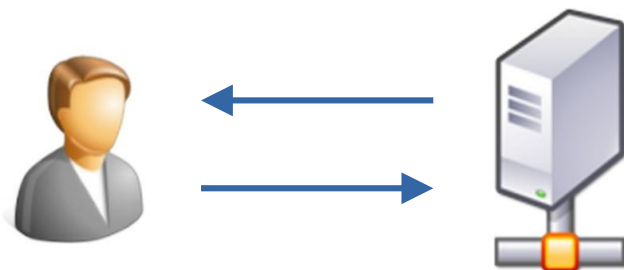
Spring Boot aide à créer des applications Spring autonomes

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

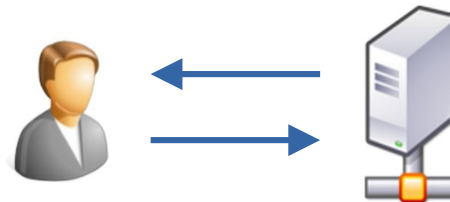
Spring Boot

- 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

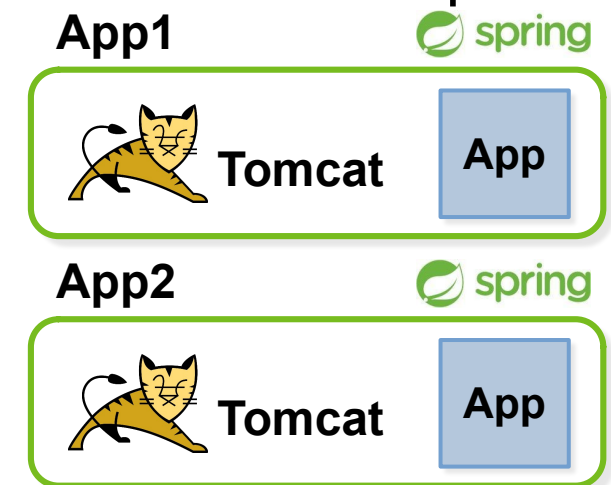


Spring Boot

- 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>
```

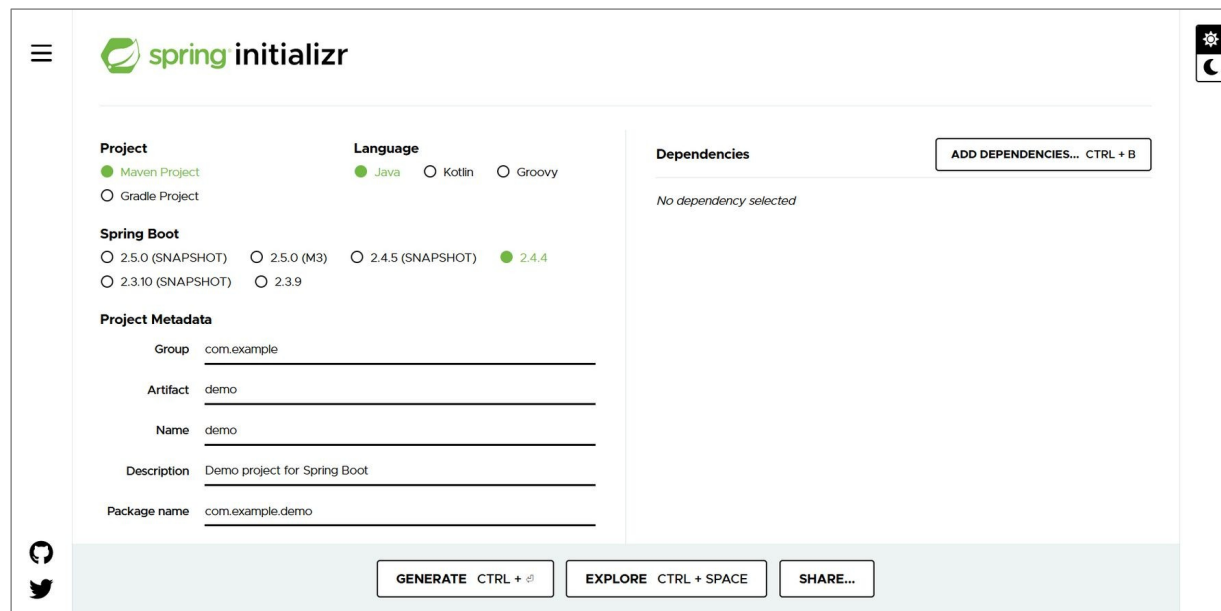
Auto-configuration



- 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 application interface. It features a sidebar with a hamburger menu and a Twitter icon. The main content area is divided into sections: 'Project' with radio buttons for 'Maven Project' (selected) and 'Gradle Project'; 'Language' with radio buttons for '.Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for versions 2.5.0 (SNAPSHOT), 2.5.0 (M3), 2.4.5 (SNAPSHOT), 2.4.4 (selected), 2.3.10 (SNAPSHOT), and 2.3.9; and 'Project Metadata' with input fields for Group (com.example), Artifact (demo), Name (demo), Description (Demo project for Spring Boot), and Package name (com.example.demo). A 'Dependencies' section on the right has a button 'ADD DEPENDENCIES... CTRL + B' and the text 'No dependency selected'. At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

- **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**

```
$ 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**

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;

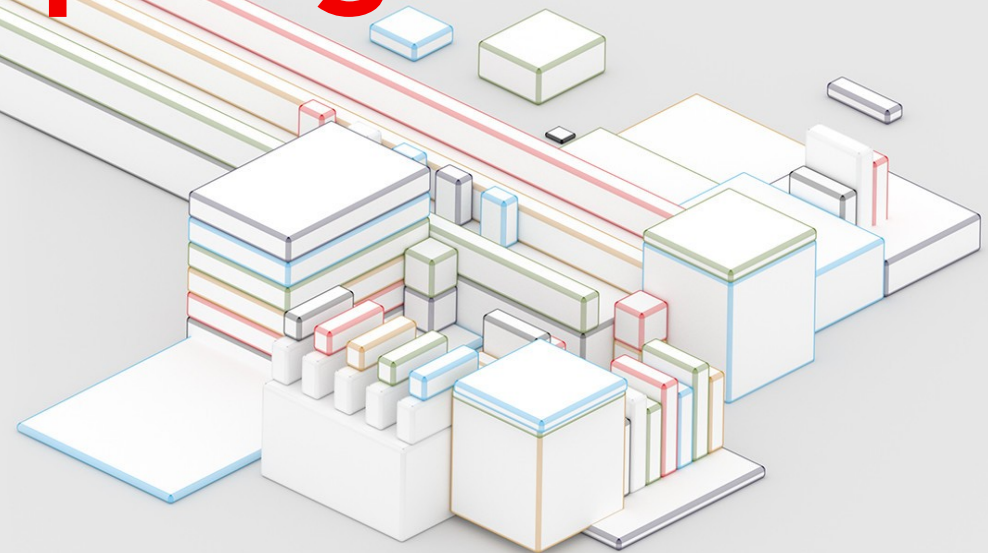
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@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.yaml** à partir de ces emplacement :
 - + |
 - 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 explicitement ces emplacements (liste de répertoire \ ou de fichier séparé par ,)
- **spring.config.additional-location** ne remplace pas mais ajoute des emplacements
- Avec le préfixe **optional**, aucun exception ne sera lancer si l'emplacement n'existe pas

Configuration externe



- **spring.config.import** permet d'importer d'autres données de configuration à partir d'autres emplacements
Les valeurs importées seront prioritaire sur celle du fichier qui a déclenché l'importation

- **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>

Bannière



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

<code>\${application.version}</code>	numéro de version de l'application dans
<code>\${application.formatted-version}</code>	MANIFEST.MF du jar
<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)
- On peut activer/désactiver les devtools avec la propriété **spring.devtools.restart.enabled**

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

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

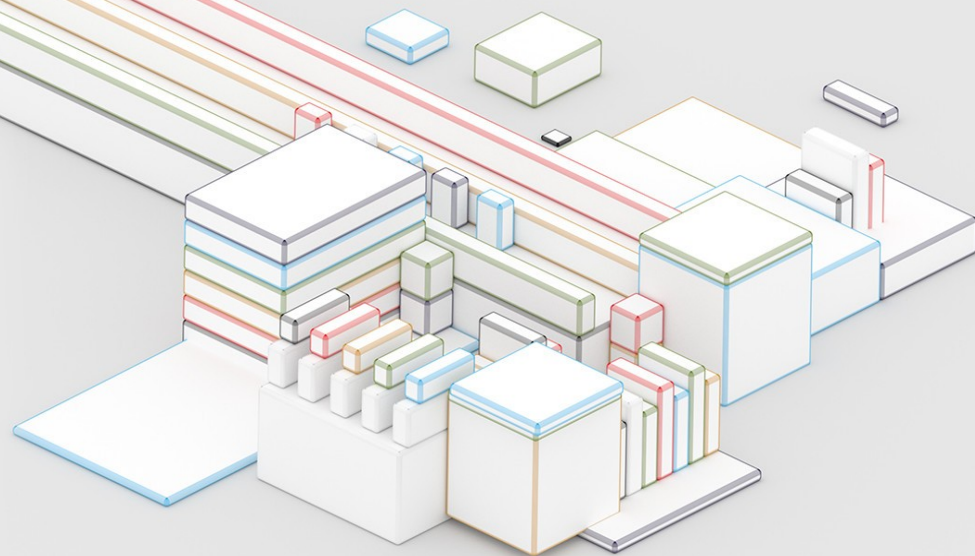
Mise à niveau de Spring Boot



- 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>
```

Annexe



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)**