

SPRING

Développer des applications
d'entreprise

PLAN

SPRING IOC

STRATÉGIE DE DÉPLOIEMENT D'UNE
APPLICATION SPRING

SPRING BOOT

SPRING IOC

SPRING AOP

SPRING JDBC

TRANSACTIONS

SPRING MVC REST SERVICES

BEAN VALIDATION

SPRING MVC EXCEPTIONS

SPRING MVC WEB

SPRING WEBFLUX

SPRING SECURITY

SPRING WEBSOCKETS

SPRING GESTION DU CACHE

SCHEDULERS AVEC SPRING

SUPERVISION AVEC SPRING ACTUATOR

Spring IOC

SPRING

- ❖ Spring est un projet opensource
- ❖ Historiquement un framework Java
- ❖ Supporté par une large communauté et la société SpringSource
- ❖ Permet la création d'applications allant des microservices aux applications d'entreprise

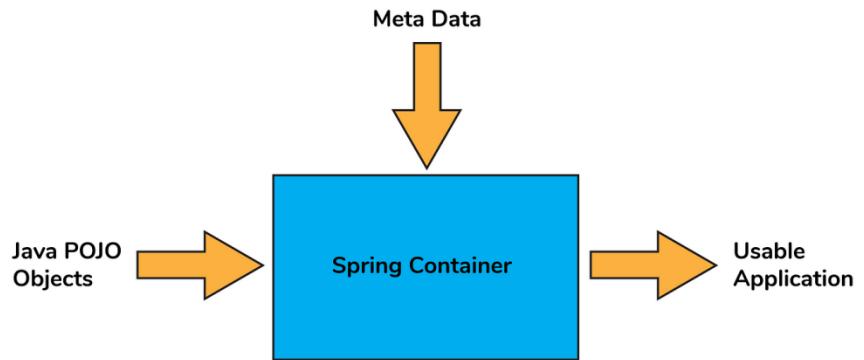


SPRING

- ❖ Spring s'occupe de la « plomberie »
- ❖ Les développeurs se concentrent sur la logique métier
- ❖ Utilisation de proxies via la programmation orientée aspect pour « injecter » du code
- ❖ Développement à base de POJO (Plain old Java object)

SPRING

- ❖ Spring s'occupe de la « plomberie » des problématiques transverses (transactions, journalisation, cache ...)
- ❖ Les développeurs se concentrent sur la logique métier
- ❖ Utilisation de proxies via la programmation orientée aspect pour « injecter » du code
- ❖ Développement à base de POJO (Plain old Java object)

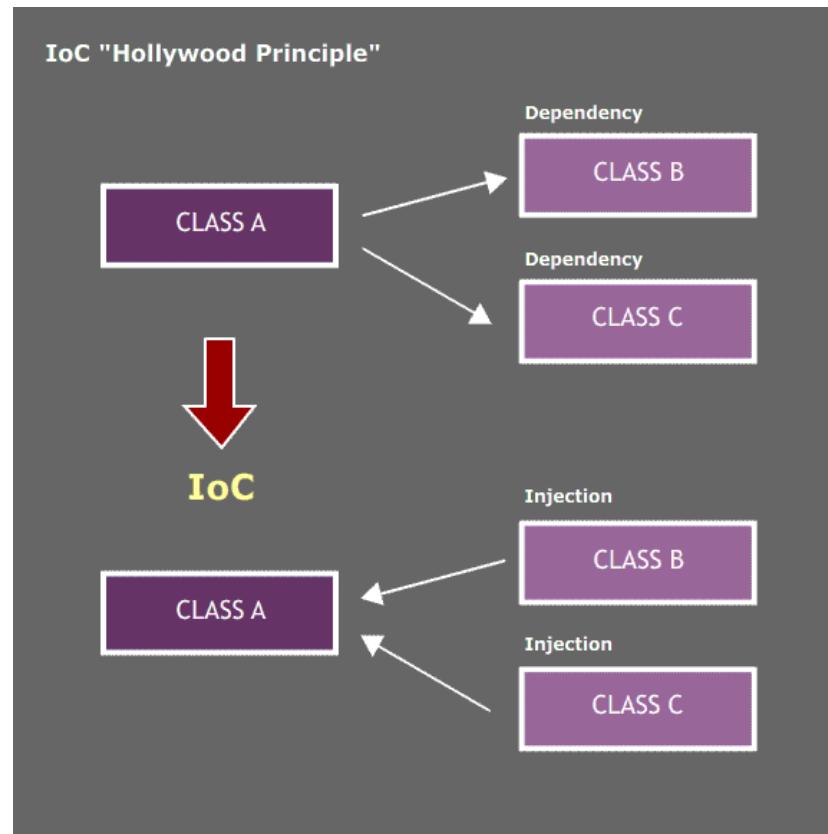


SPRING

- ❖ A l'origine un framework mettant en ouvre l'inversion de contrôle
- ❖ Inversion de contrôle (dans le cadre du génie logiciel):
principe qui postule qu'un programme est plus facilement maintenable, testable et bénéficie d'un couplage faible si la gestion de ses flux et ses dépendances est porté par un autre système
- ❖ Principe d'Hollywood : Ne nous appelez pas, c'est nous qui vous appellerons

SPRING

- ❖ Principe d'Hollywood, **l'injection de dépendances** en est une forme



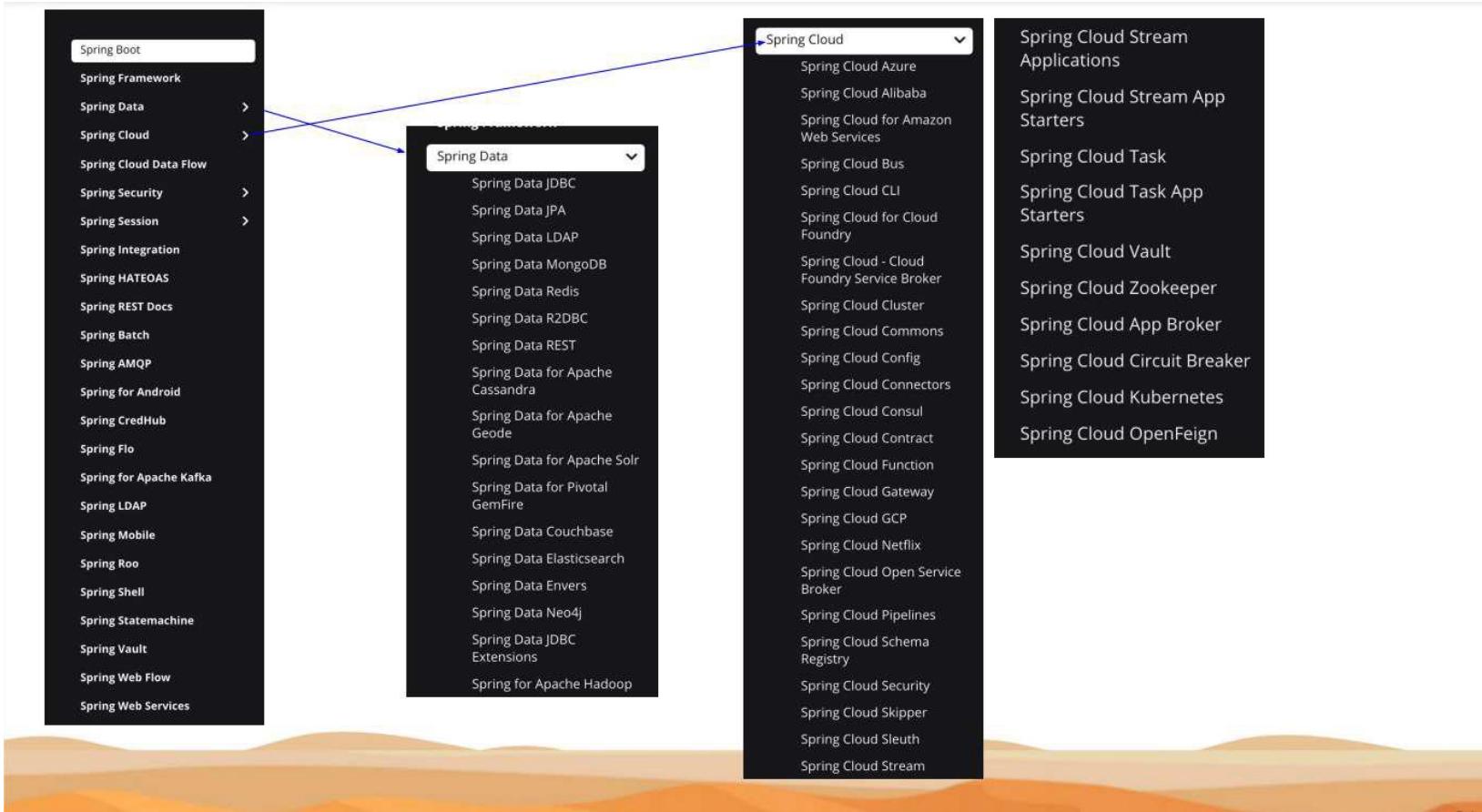
SPRING

- ❖ A l'origine Spring est un framework d'injection de dépendances
- ❖ Spring injecte et gère les dépendances dans l'application
- ❖ Il peut enrichir ces dépendances et apporter une couche pour traiter les problématiques transverses
- ❖ Utilisation de la programmation orientée aspects

- ❖ Exemples : Rendre les dépendances transactionnelles, sécuriser les accès aux méthodes ...

SPRING

Pas que de l'injection de dépendances !



21

SPRING

- ❖ Code opensource
- ❖ Licence Apache License 2.0 permet à chacun d'utiliser, étudier, améliorer, distribuer le contenu. Seule obligation : mentionner l'auteur
- ❖ Géré par VMWare maîtrise le cycle de vie et les releases



SPRING

- ❖ Code source <https://github.com/spring-projects/spring-framework>
- ❖ Site officiel <https://spring.io/>
- ❖ Liste des projets <https://spring.io/projects>
- ❖ Versions stables pour Java disponibles sur le central Maven

SPRING

- ❖ Spring framework vs sous projets
- ❖ Plusieurs modules sur le framework de base (JDBC, AOP, Beans, Context...). Tous les modules partagent la même version et font partie du même projet.
- ❖ Outre le Spring Framework et ses différents modules, il existe d'autres projets pluggé à Spring . Ces projets apportent des solutions à d'autres problématiques rencontrés par les applications d'entreprise.
- ❖ Versionning différents entre framework de base et les sous projets

SPRING

- ❖ Spring boot
- ❖ Spring Cloud
- ❖ Spring Data
- ❖ Spring Batch
- ❖ Spring Security
- ❖ Spring Integration
- ❖ Spring HATEOAS
- ❖ ...

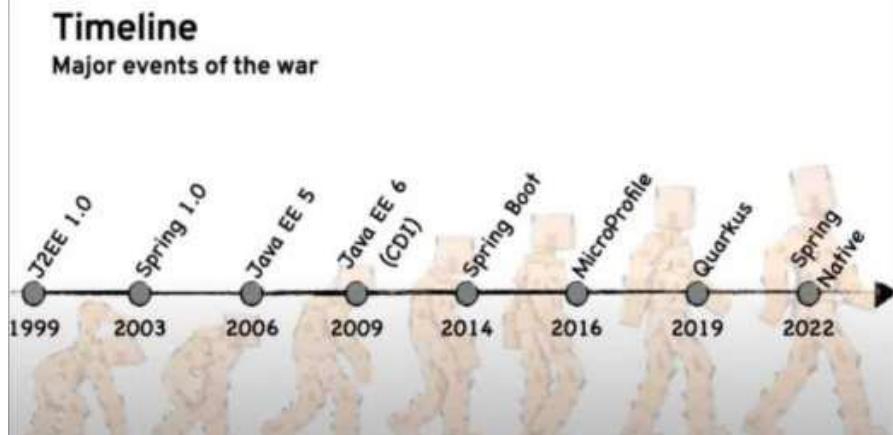
SPRING

- ❖ Pas d'obligation d'embarquer tout
- ❖ Choix des dépendances en fonction des exigences applicatives
- ❖ Moins de contraintes et plus de souplesse
- ❖ Framework léger comparé à Jakarta

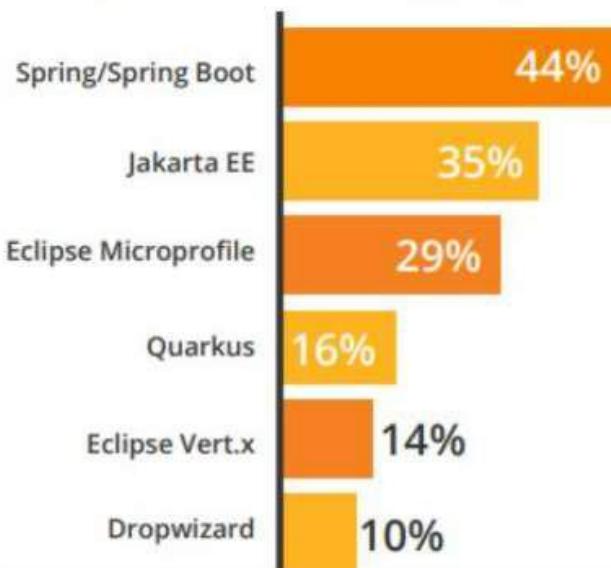
SPRING VS JAKARTA

- ❖ La spécification Jakarta est le standard Java pour des applications d'entreprise
- ❖ Implémentation de Jakarta via les serveurs d'applications : Jboss, Weblogic, Websphere ...
- ❖ Spring n'est pas un serveur d'application : Utilisation possible de Spring pour remplacer tout ou en partie des serveurs d'applications
- ❖ Spring se déploie la plupart du temps dans des serveurs légers comme Tomcat ou Jetty

SPRING VS JAKARTA



Which Java frameworks are you using for cloud native applications?



SPRING VS JAKARTA

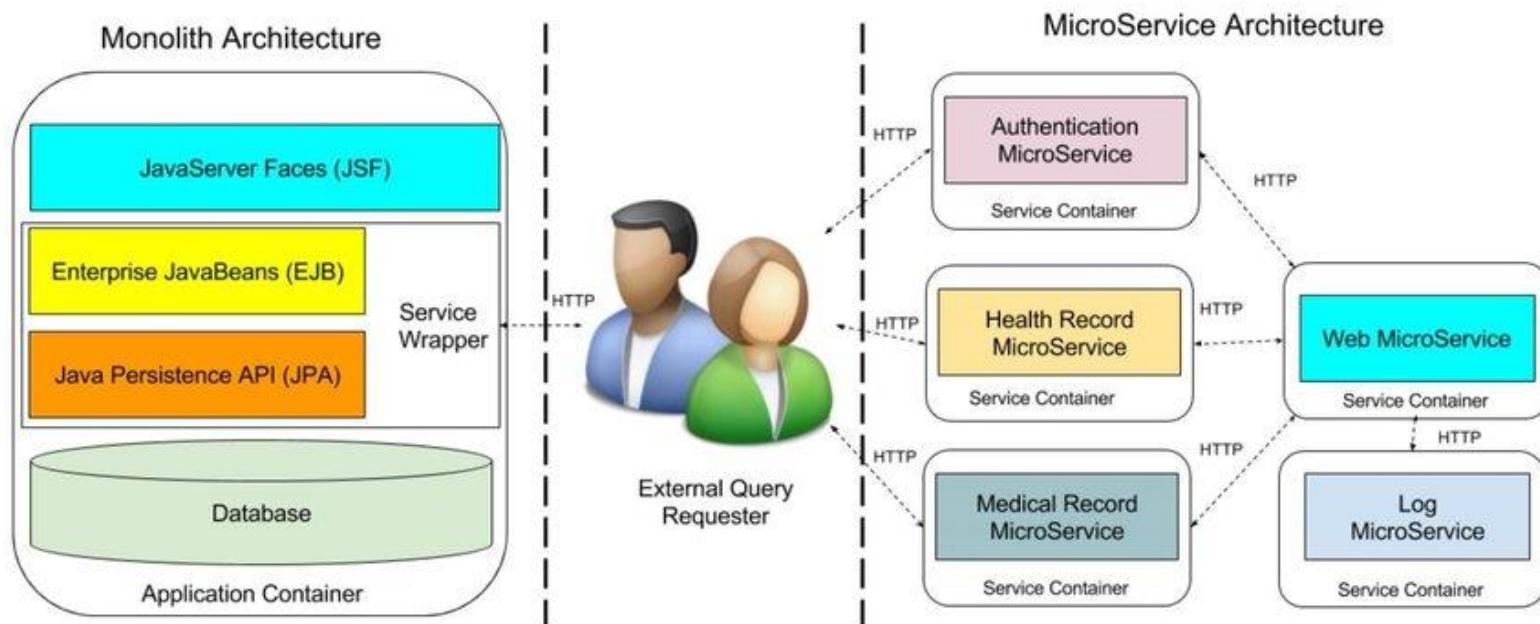
- ❖ Spring à l'origine est une alternative au modèle d'architecture de J2EE réputé lourd et complexe
- ❖ J2EE a eu plusieurs succès comme les servlets, JSP, JMS, JTA
....
- ❖ La composante métier incarnée par les EJBs s'avéra être un échec : composants distribués trop lourd et pas forcément nécessaires pour la plupart des applications
- ❖ Emergence de plusieurs solutions opensource basé sur les POJOs pour combler les lacunes de J2EE : Spring, Hibernate entre autres

JAKARTA

- ❖ Les serveurs déploient et gèrent les composants logiciels
- ❖ Les serveurs d'application ont la responsabilité de fournir les services prenant en charge la sécurité, la gestion des transactions, la gestion des accès aux services externes (comme les bases de données)
- ❖ Les composants (Servlet, EJB...) doivent être conformes à une spécification technique afin qu'ils puissent être pris en charge par un serveur.
- ❖ Nécessité pour le développeur de connaître les APIs souvent complexes

JAKARTA

- ❖ Serveur monolithique exigeant en ressources et peu adapté à des redémarrages fréquents
- ❖ Architecture peu conformes aux applications actuelles, notamment cloud



SPRING VS JAKARTA

Avec Spring

- ❖ Bâtir des applications qui embarquent elles-mêmes les services dont elles ont besoin (conteneur léger)
- ❖ Une application utilisant Spring devient elle-même un mini conteneur pour accueillir les services normalement offerts par un serveur Jakarta EE.
- ❖ Approche modulaire, dimensionnement des applications sans dépendre d'un serveur monolithique : très adapté aux infrastructures cloud et à la mise en place de micro-services.

SPRING VS JAKARTA

Framework Spring

Pas de contraintes, ni de normes de développement

Approche non intrusive :

- ❖ Utilisation de l'inversion de contrôle
- ❖ Utilisation de la programmation orienté aspects
- ❖ Intégration de solutions existantes (notamment standards Jakarta EE)

Stratégie de déploiement d'une application Spring

QUESTION

ANSWER

STRATÉGIE DE DÉPLOIEMENT

Déploiement standalone
(springboot)

Déploiement dans un serveur
d'applications (tomcat, widlfly...)

Image docker

Cluster Kubernetes

Installation comme service
init.dService (System V)

Installation comme service systemd

Services cloud(Azure, AWS, OCI ...)

```
[Unit]
Description=myapp
After=syslog.target

[Service]
User=myapp
ExecStart=/var/myapp/myapp.jar
SuccessExitStatus=143

[Install]
WantedBy=multi-user.target
```

Spring boot

SPRING BOOT

- ❖ Sous projet de Spring
- ❖ Fournit un cadre pour la création rapide d'applications de type micro services et d'applications web
- ❖ Mise en œuvre de RAD (Rapid Application Development) pour le framework Spring
- ❖ Convention over configuration



SPRING BOOT

Pourquoi Spring boot ?

Mise en œuvre d'une application Spring peut être complexe :
Quel archetype Maven utiliser ?

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-archetype	1.0
org.apache.maven.archetypes	maven-archetype-j2ee-simple	1.0
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1

Quelles sont les dépendances requises ?(SpringJdbc, Springcore...)
Comment gérer la configuration ?

```
<!-- Step 1: Configure Spring MVC Dispatcher Servlet -->
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</s
  <!-- Step 2: Configure HelloMessageService bean -->
  <bean id="HelloMessageService" class="com.example.HelloMessageService">
    <!-- Step 3: Configure HelloController bean -->
    <!-- Step 4: Configure HelloController bean -->
  </bean>
</servlet>
```

SPRING BOOT

- ❖Création d'application standalone (pas de besoin de déployer un WAR)
- ❖Rendre le plus minime possible la configuration manuelle et se base l'auto configuration via les annotations
- ❖Résolution des conflits entre dépendances
- ❖Serveur embarqué pour démarrer et tester rapidement
- ❖Augmente la productivité et réduit le temps de développement

SPRING BOOT

- ❖ Démarrage d'un projet

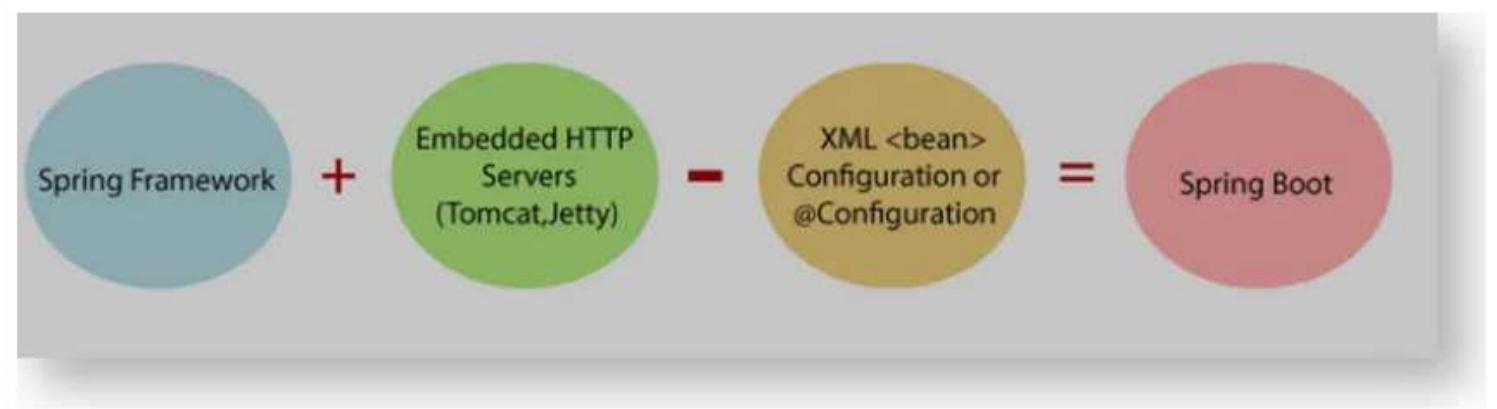
<https://start.spring.io/>

The screenshot shows the Spring Initializr web application interface. At the top, there are two tabs: 'Maven Project' (selected) and 'Gradle Project'. Below these are 'Language' options: Java (selected), Kotlin, and Groovy. Under 'Spring Boot', the version '2.3.2 (SNAPSHOT)' is selected. The 'Dependencies' section is currently empty, indicated by the text 'No dependency selected' and a button labeled 'ADD DEPENDENCIES... CTRL + B'. The 'Project Metadata' section contains fields for Group ('com.example'), Artifact ('demo'), Name ('demo'), Description ('Demo project for Spring Boot'), Package name ('com.example.demo'), and Packaging ('Jar' selected). At the bottom, there are links for Java 14, Java 11, and Java 8.

SPRING BOOT

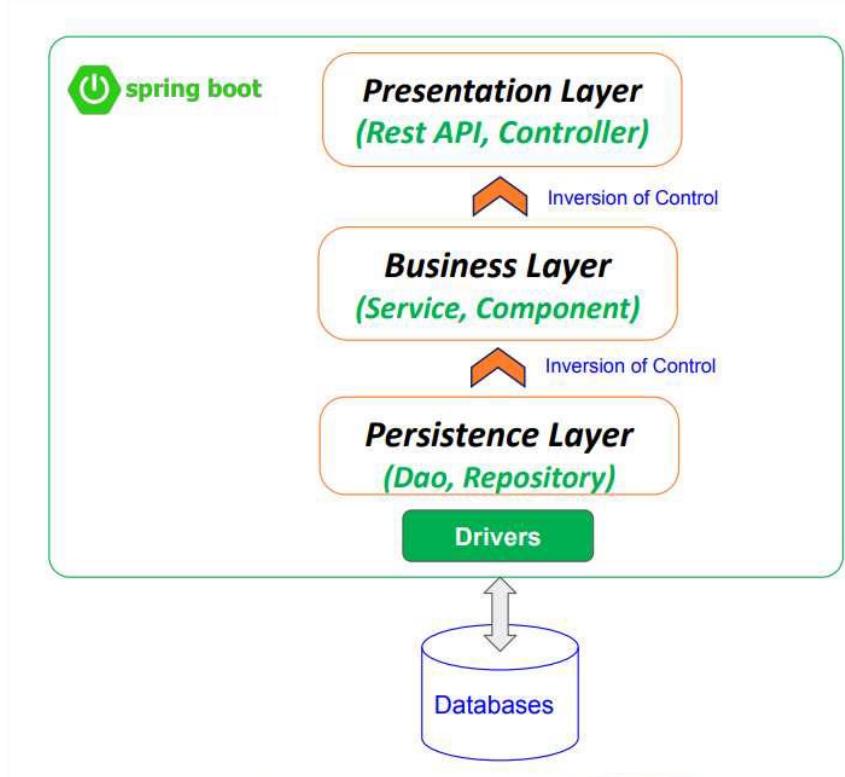
Remarques :

- ❖ Springboot ne remplace aucun des modules de Spring (MVC, Data ...) : il utilise ces modules en background
- ❖ Un code avec Springboot ne s'exécute pas plus rapidement qu'un code avec les modules «classiques» de Spring
- ❖ Pas de d'IDE spécifique pour Springboot



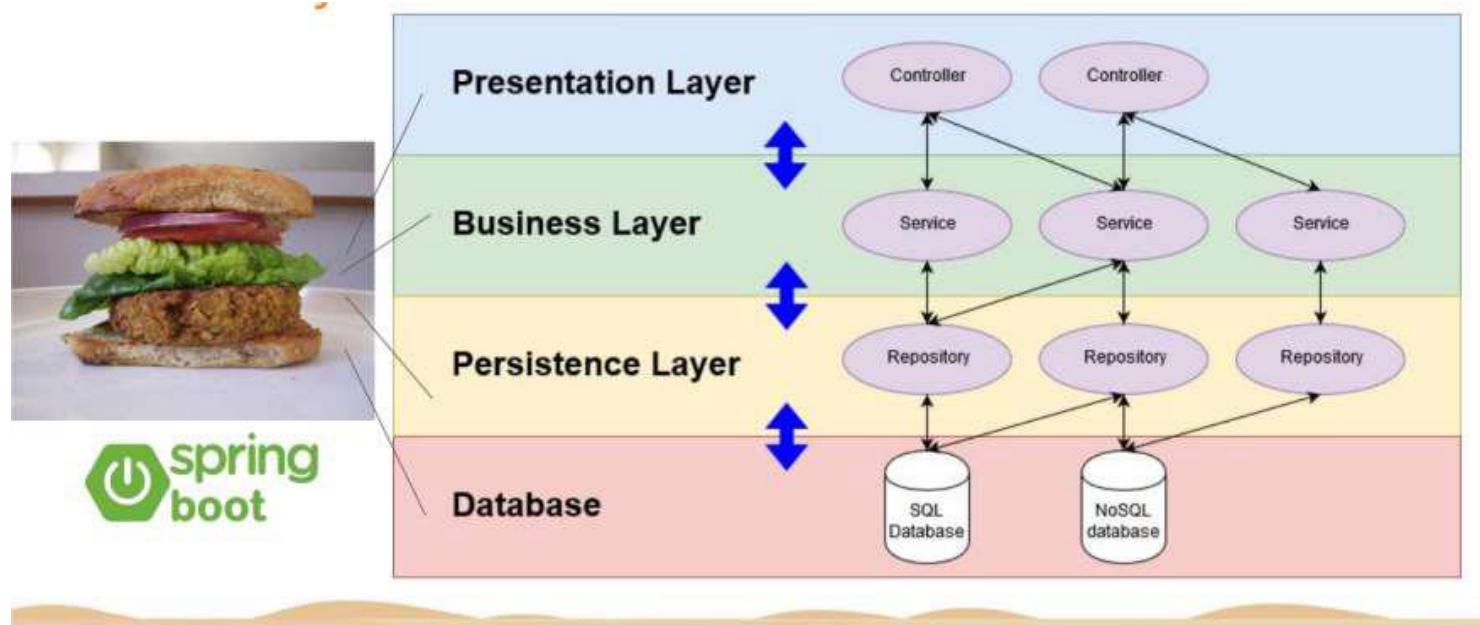
SPRING BOOT

Architecture



SPRING BOOT

Vocabulaire



SPRING BOOT

Run : simple méthode main

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

@SpringBootApplication

@Configuration

@EnableAutoConfiguration

@ComponentScan

SRING BOOT

VS

SPRING

Spring	Spring boot
Principalement utilisé pour créer des applications d'entreprise	Principalement utilisé pour créer des micro services de type Rest
a pour objectif de proposer un cadre plus simple et moins lourd que Jakarta EE pour créer des applications d'entreprise	Permet de réduire le temps nécessaire pour développer des applications Spring
L'inversion de contrôle est la base de Spring	L'auto configuration est la base de Spring boot
Permet un couplage faible des composants d'une application	Permet de créer une application standalone avec peu de configuration
Le développeur fait beaucoup de configuration	Convention over configuration
Besoin de configurer un serveur pour tester	Serveur déjà intégré et prêt à fonctionner
Pas de support pour les bases en mémoire	Support pour des bases en mémoire comme H2

SPRING BOOT

Remarques :

- ❖ Springboot ne remplace aucun des modules de Spring (MVC, Data ...) : il utilise ces modules en background
- ❖ Un code avec Springboot ne s'exécute pas plus rapidement qu'un code avec les modules «classiques» de Spring
- ❖ Pas de d'IDE spécifique pour Springboot

Spring IOC

INJECTION DE DÉPENDANCES

Une forme d'inversion de contrôle

Différent du design pattern Factory :

Le pattern Factory permet de créer des objets

L'jection de dépendances permet de lier des objets

INJECTION DE DÉPENDANCES

Une classe a une dépendance et délègue l'injection de cette dépendance : la classe doit préciser un moyen d'injecter cette dépendance

```
9  
0  public class InvoiceService {  
1  
2      private final UserService userService;|
```



userService doit être injecté

INJECTION PAR CONSTRUCTEUR

La classe précise dans le constructeur la/les dépendance(s)

```
public InvoiceService(UserService userService) {  
    this.userService = userService;  
}
```

Spring utilise le constructeur
pour injecter la dépendance

INJECTION PAR SETTER

La classe spécifie un setter et le Spring utilise ce setter pour faire l'injection

```
public void setUserService(UserService userService) {  
    this.userService = userService;  
}
```



Spring utilise le setter pour injecter la dépendance

INJECTION PAR CHAMP

Injection de la dépendance directement dans le champ en utilisant l'api de reflection Java

```
public class InvoiceService {  
    private UserService userService;
```

Spring injecte directement dans
le champ la dépendance

CHOIX DE LA MÉTHODE D'INJECTION

Le choix du type d'injection est toujours un débat quant à la complexité, la performance et d'autres paramètres complexes

Être cohérent sur le choix

Injection par constructeur :
recommandée pour les dépendances obligatoires

Injection par setter
recommandée pour les dépendances optionnelles

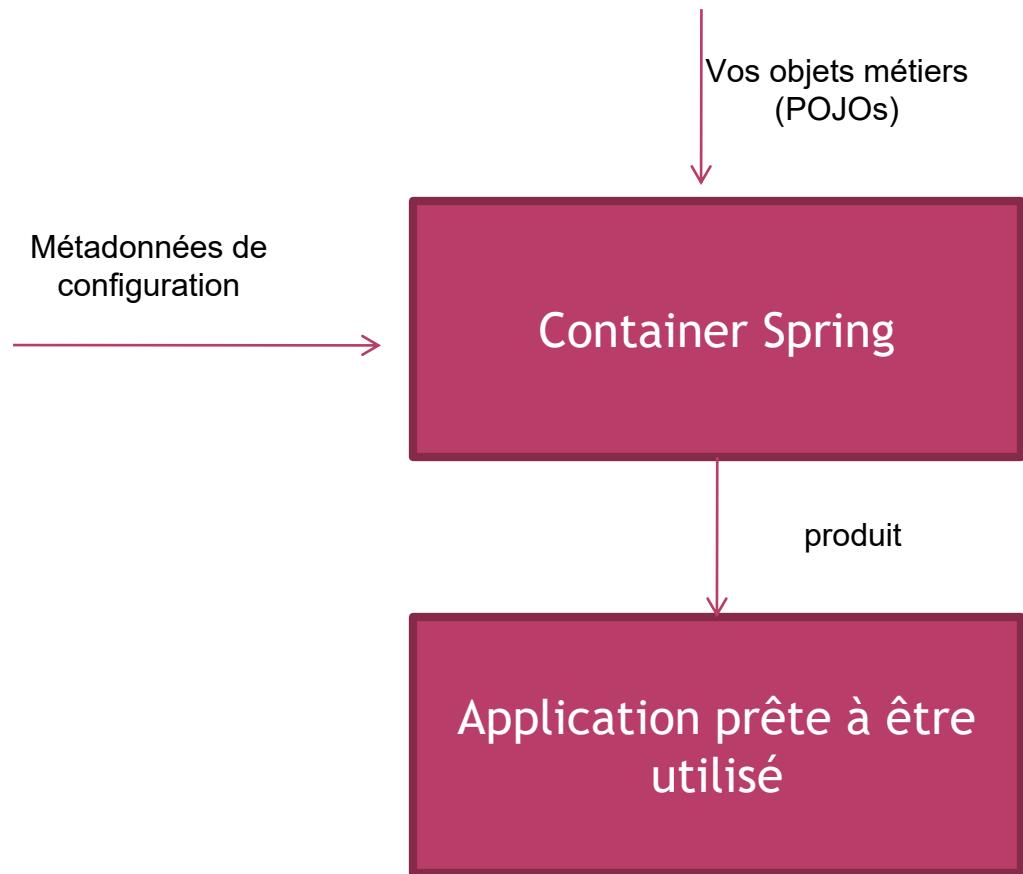
Injection par champ
plus concise mais perturbant les tests

SPRING IOC

Basé sur les POJOs



SPRING IOC



SPRING IOC

Termes

- ❖ **ApplicationContext**

- Represents the Spring IOC container

- ❖ **Bean**

- An object created and managed by Spring

- ❖ **BeanDefinition**

- describes a bean instance

SPRING IOC

La configuration peut être fournie de plus manières

- ❖ Via XML
- ❖ Via une classe de configuration Java
- ❖ Via des annotations

Les objets sont de simples objets java (POJO) implémentant des interfaces métiers

- ❖ Architecture plus souple
- ❖ Meilleure intégration des tests

SPRING IOC

Cycle de vie



SPRING IOC

Création d'une application context

Une ApplicationContext peut être lié à un type de contexte:

Standalone

Web

Contexte de tests unitaire

....

Le fichier de configuration peut être spécifié par des préfixes indicatifs de la source

classpath

Système de fichiers

url

...

SPRING IOC

Exemple : application standalone avec ClassPathApplicationContext

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(
        new String[] {"services.xml", "repositories.xml"}));

// retrieve configured instance
CurrencyService currencyService
    = context.getBean("currency", CurrencyService.class);

AccountService accountService
    = context.getBean(AccountService.class);
```

SPRING IOC

Exemple : contexte pour une application Web

```
<servlet>
  <servlet-name>
    context
  </servlet-name>
  <servlet-class>
    org.springframework.web.context.ContextLoaderServlet
  </servlet-class>
  <load-on-startup>1
  </load-on-startup>
</servlet>
```

SPRING IOC

Configuration XML : utilisation de namespaces

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <bean id="" class=""></bean>

    <bean id="" class="" />
</beans>
```

SPRING IOC

Exemple configuration XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="book" class="com.formation.spring">
        <property name="titre" value="Spring XML Configuration"/>
        <property name="auteur" value="Hugues Test"/>
        <property name="publications" value="Ma Publication"/>
    </bean>
</beans>
```

SPRING IOC

Par constructeur :

```
<bean id="mercedes" class="com.formation.domain.voiture">
<constructor-arg index="0" ref="diesel" />
<constructor-arg index="1" ref="pirelli" />
</bean>
```

```
<bean id="diesel" class="com.formation.domain.Moteur">
<constructor-arg index="0" value="v1" />
<constructor-arg index="1" value="2" />
</bean>
```

```
<bean id="pirelli"
class="com.formation.domain.Roue">
<constructor-arg value="100b" />
</bean>
```

SPRING IOC

Par setter

```
<bean id = "mercedes" class = "com.formation.domain.voiture">
    <property name = "moteur" ref = "diesel"/>
</bean>

<bean id = "diesel" class = "com.formation.domain.Moteur">
    <property name = "version" value = "v12"/>
    <property name = "nbreCoeurs" value = "2"/>
</bean>
```

SPRING IOC

Configuration par annotations

Annoter les classes avec
@Component,
@Autowired pour les
dépendances (à placer sur
les setters, champs ou
constructeurs selon le
type d'injection choisi)

```
@Component
public class ClientDAO
{
    @Override
    public String toString() {
        return "Ici un client DAO";
    }
}
```

```
package com.formation.client.services;

import org.springframework.beans.factory.annotation.Autowired;

@Component
public class ClientService
{
    @Autowired
    ClientDAO clientDAO;

    @Override
    public String toString() {
        return "ClientService a une dépendance [clientDAO=" + clientDAO + "]";
    }
}
```

SPRING IOC

Configuration par annotations

Activer le scan des packages contenant ces classes

```
http://www.springframework.org/schema/beans/spring-beans.xsd (xsi:schemaLocation) | http://w
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframewo
http://www.springframework.org/schema/context http://www.springframework.org/schema/context
<context:component-scan
    base-package="com.formation.client" />
</beans>
```

SPRING IOC

Recherche des dépendances à injecter

Par nom : Spring cherche un objet qui a le même nom que l'id spécifié dans le fichier de configuration xml

Par type : pas de nommage explicite du bean à injecter; Spring recherche alors un bean du même type

SPRING IOC

- ❖ Configuration avec Java

Définition d'une classe Java qui jouera le rôle de factory de beans

Classe annotée avec `@Configuration` et optionnellement `@ComponentScan` pour les packages des beans Spring

- ❖ Méthode préconisée pour les beans métiers

SPRING IOC

❖ Configuration avec Java

```
package com.formation.configuration;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan({ "com.formation"})
public class AppConfigration {

}
```

SPRING IOC

- ❖ Configuration avec Java : classe
AnnotationConfigApplicationContext à utiliser comme contexte

```
package com.formation.client;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.formation.client.services.ClientService;
import com.formation.configuration.AppConfiguration;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new AnnotationConfigApplicationContext(AppConfiguration.class);

        ClientService cust = (ClientService)context.getBean("clientService");
        System.out.println(cust);

    }
}
```

SPRING IOC

Résumé types de configuration

- ❖ Privilégier la configuration Java pour tous les beans métiers
- ❖ Pour tous les beans qui viennent de librairies externes, utiliser la configuration XML (pas le choix, les classes de ces librairies ne pouvant être annotées ...)
- ❖ Tjrs regarder la documentation pour définir correctement un bean d'une librairie

SPRING IOC

- ❖ Nommage des beans
- ❖ Les noms des beans doivent être uniques dans le context Spring
- ❖ Par convention, nom de la classe en camelCase
- ❖ Configuration XML : spécifier le nom sur l'attribut id
- ❖ Avec @Component (par défaut nom de la classe en camelCase) :

```
package configuration.client.services;

import org.springframework.beans.factory.annotation.Autowired;

@Component(value = "mon_nom_de_beans")
public class ClientService {
    @Autowired
    ClientDAO clientDAO;

    @Override
    public String toString() {
        return "ClientService a une dépendance [clientDAO=" + clientDAO + "]";
    }
}
```

SPRING IOC

Portée des beans

Se définit à la configuration des beans

- ❖ **Singleton** (par défaut) : un seul bean est créé dans tout le contexte; Spring l'injecte pour tous les objets qui ont une dépendance du même type
- ❖ **Prototype** : un bean est créé chaque fois qu'un autre bean a une dépendance du même type
- ❖ **Session** : un bean par session
- ❖ **Request** : un bean par requête

...

SPRING IOC

Portée des beans

XML :

```
<bean id="customerService" class="com.formation.customer.services.CustomerService" scope="singleton">
    <property name="customerDAO" ref="customerDAO" />
</bean>

<bean id="customerDAO" class="com.formation.customer.dao.CustomerDAO" scope="prototype"/>
```

Annotation :

```
package com.formation.client.dao;

import org.springframework.context.annotation.Sc
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class ClientDAO
{
    @Override
    public String toString() {
        return "Ici un client DAO";
    }
}
```

SPRING IOC

Cycle de vie des beans

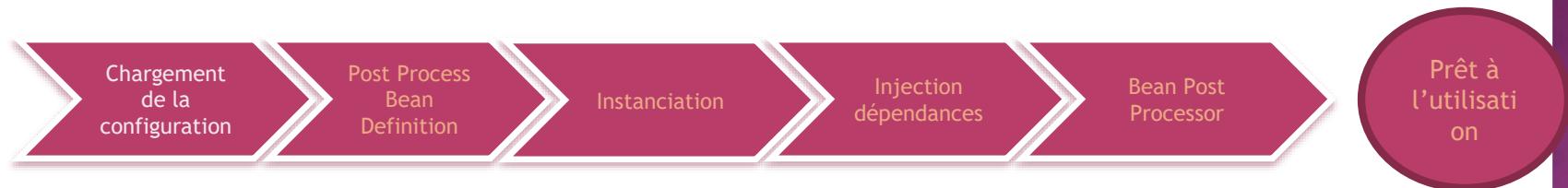


Spring gère le cycle de vie des beans depuis leurs créations jusqu'à leurs destructions

Il utilise l'AOP pour enrichir ces beans

SPRING IOC

Cycle de vie des beans

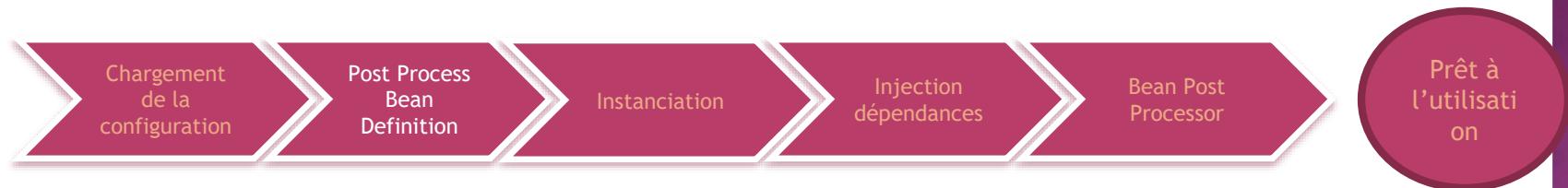


Etape 1 : Chargement la configuration

Peu importe le type de configuration choisie, Spring toutes les métadonnées de configuration des beans en mémoire

SPRING IOC

Cycle de vie des beans



Etape 2 : Exécution des BeanFactoryPostProcessor

Dans cette étape, Spring permet de modifier la configuration des beans, la plupart du temps en injectant de nouvelles propriétés; Cela est fait avec l'interface BeanFactoryPostProcessor

SPRING IOC

BeanFactoryPostProcessor permet d'ajouter ou d'overrider des propriétés avant l'instanciation des beans

```
@FunctionalInterface
public interface BeanFactoryPostProcessor {

    /**
     * Modify the application context's internal bean factory after its standard
     * initialization. All bean definitions will have been loaded, but no beans
     * will have been instantiated yet. This allows for overriding or adding
     * properties even to eager-initializing beans.
     *
     * @param beanFactory the bean factory used by the application context
     * @throws org.springframework.beans.BeansException in case of errors
     */
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;

}
```

SPRING IOC

Exemple de BeanFactoryPostProcessor :

PropertyPlaceholderConfigurer

Permet de remplacer des variables définis dans la configuration des beans par des valeurs définis dans des fichiers

```
<bean  
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
        <property name="location">  
            <value>monfichier.properties</value>  
        </property>  
    </bean>  
  
<bean id="customerService"  
    class="com.formation.customer.services.CustomerService">  
        <property name="customerDAO" ref="customerDAO" />  
        <property name="devise" value="${devise.pays}" />  
    </bean>
```

SPRING IOC

Exemple de BeanFactoryPostProcessor :

PropertyPlaceholderConfigurer avec configuration Java

```
package com.formation.configuration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration
@ComponentScan({ "com.formation" })
@PropertySource("classpath:monfichier.properties")
public class AppConfiguration {

    @Bean
    public static PropertySourcesPlaceholderConfigurer placeHolderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```

SPRING IOC

Exemple de BeanFactoryPostProcessor :

PropertyPlaceholderConfigurer avec configuration Java

```
@Component(value = "mon_nom_de_beans")
public class ClientService
{
    @Autowired
    ClientDAO clientDAO;

    @Value("${devise.pays}")
    private String devise; ←

    @Override
    public String toString() {
        return "CustomerService : devise ("+getDevise()+" [customerDAO=" + clientDAO + "]"
    }

    public String getDevise() {
        return devise;
    }
}
```

SPRING IOC

Cycle de vie des beans



Etape 3 : Instanciation des beans

Spring crée les beans et des proxies de ces beans afin de réaliser les traitements transverses définis par le développeur (rendre transactionnelles les beans, sécuriser les méthodes) lors des appels sur ces beans

SPRING IOC

Cycle de vie des beans

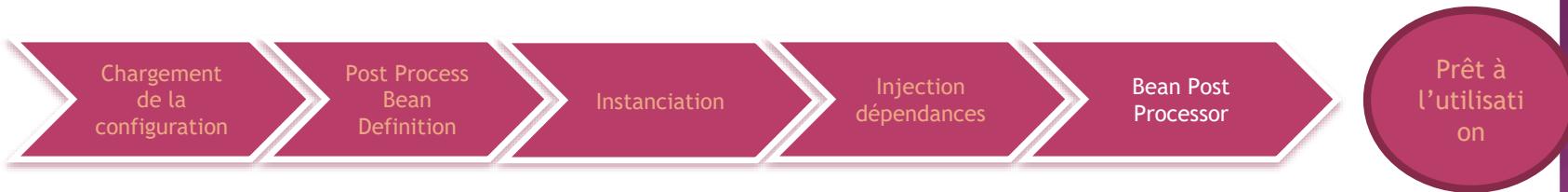


Etape 4 : Injection des dépendances

Pour l'injection par constructeur, cette étape est couplée à l'étape d'instanciation

SPRING IOC

Cycle de vie des beans



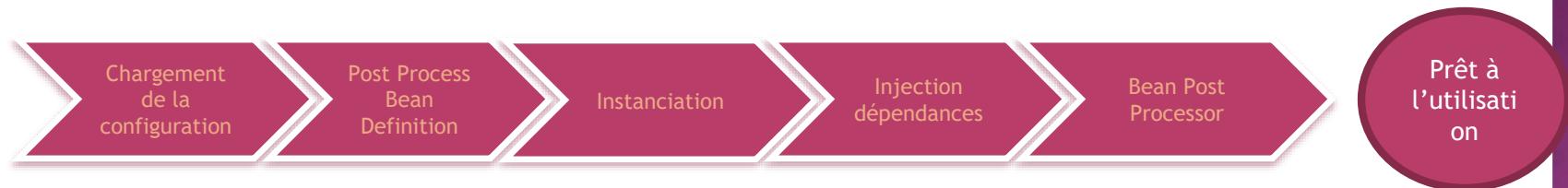
Etape 5 : Execution des BeanPostProcessor ; exemple avec PostConstruct (annotation de Jakarta)

```
@PostConstruct  
public void maMethodePostConstruct() {  
    System.out.println("Je m'execute après  
l'instanciation et l'initialisation ! ");  
}
```

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

SPRING IOC

Cycle de vie des beans

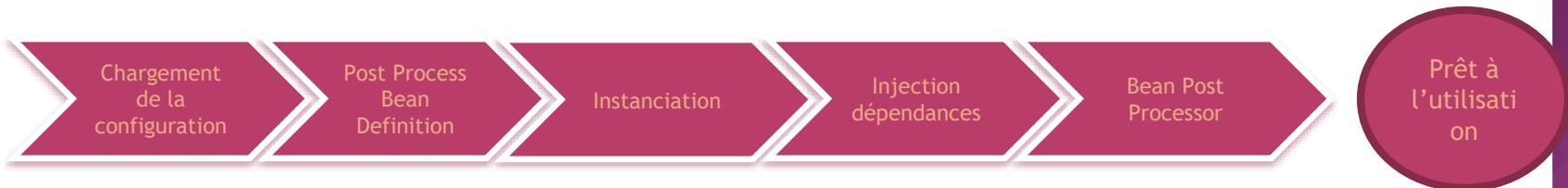


Etape 6 : L'application est prête à l'emploi

On peut alors demander des beans aux contexte Spring

SPRING IOC

Cycle de vie des beans



Spring permet d'exécuter de la logique avant la destruction des beans avec l'annotation `@PreDestroy` sur une méthode

```
@PreDestroy
public void maMethodePreDestroy() {
    System.out.println("Je m'execute après
l'instanciation et l'initialisation !");
}
```

SPRING IOC

Lazy loading

Spring permet de n'instancier des beans qu'à la demande

Utilisation de l'annotation `@Lazy` sur une classe ou de l'attribut
« `lazy-init=true` » sur une configuration XML

Le cycle de vie classique n'est plus utilisé alors

SPRING IOC

Démarrage de l'application

Instanciation d'une classe implémentant ApplicationContext

Plusieurs implémentations existent en fonction du type du fichier de configuration, de la nature de l'application ...

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext(new String[] {"applicationContext.xml"});
```

Configuration
xml

```
ApplicationContext context =  
    new AnnotationConfigApplicationContext(AppConfiguration.class);
```

Configuration
Java

SPRING IOC

Démarrage de l'application dans un contexte web

Utilisation d'un listener fourni par Spring dans le fichier web.xml

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/spring/applicationContext.xml</param-value>
</context-param>
```

SPRING IOC

Injection de liste

```
<bean name="France" class="com.formation.customer.services.Pays">
    <property name="villes">
        <list>
            <value>Paris</value>
            <value>Lille</value>
            <value>Marseille</value>
            <value>Angers</value>
        </list>
    </property>
</bean>
```

```
package com.formation.customer.services;
import java.util.List;

public class Pays {
    private List<String> villes;

    public List<String> getVilles() {
        return villes;
    }

    public void setVilles(List<String> villes) {
        this.villes = villes;
    }
}
```

SPRING IOC

Injection de Set

```
<bean name="France" class="com.formation.customer.services.Pays">
    <property name="villes">
        <set>
            <value>Paris</value>
            <value>Lille</value>
            <value>Marseille</value>
            <value>Angers</value>
        </set>
    </property>
</bean>
```

```
package com.formation.customer.services;
import java.util.Set;

public class Pays {
    private Set<String> villes;

    public Set<String> getVilles() {
        return villes;
    }

    public void setVilles(Set<String> villes) {
        this.villes = villes;
    }
}
```

SPRING IOC

Injection de Map

```
<bean name="France" class="com.formation.customer.services.Pays">
    <property name="habitants">
        <map>
            <entry key="reims" value="20000" />
            <entry key="lens" value="30000" />
        </map>
    </property>
</bean>
```

```
Map<String, Integer> habitants;
```

```
public Set<String> getVilles() {
    return villes;
}
```

```
public void setVilles(Set<String> villes) {
    this.villes = villes;
}
```

```
public Map<String, Integer> getHabitants() {
    return habitants;
}
```

```
public void setHabitants(Map<String, Integer> habitants) {
    this.habitants = habitants;
}
```

SPRING IOC

Inner bean

```
<bean id="client" class="com.formation.domain.Client">
    <property name="personne">
        <bean class="com.formation.domain.Personne">
            <property name="nom" value="Alain" />
            <property name="adresse" value="Paris 1ier" />
            <property name="age" value="28" />
        </bean>
    </property>
</bean>
```

Conseillé si un bean est utilisé uniquement par une propriété donnée

Connu uniquement que par le bean parent : pas besoin d'id

SPRING IOC

Découpage en plusieurs fichiers de configuration

Il est conseillé de découper et de regrouper les beans relatifs à un domaine sur des fichiers différents

un fichier de configuration pour les beans gérant la sécurité

un fichier de configuration pour les beans gérant la transaction

....

Et les importent sur le fichier de configuration principales

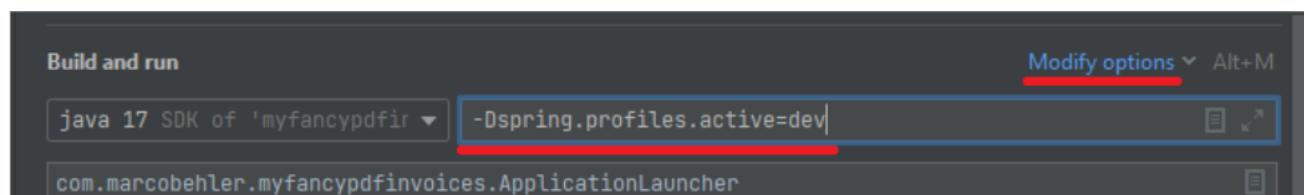
```
<import resource="beans-security.xml" />  
<import resource="beans-transactions.xml" />  
...
```

SPRING IOC

Profiles Spring

Permet de spécifier au démarrage un profil et d'adapter la configuration en fonction du profil (typiquement dev, prod ...)

On choisit le profil au démarrage :

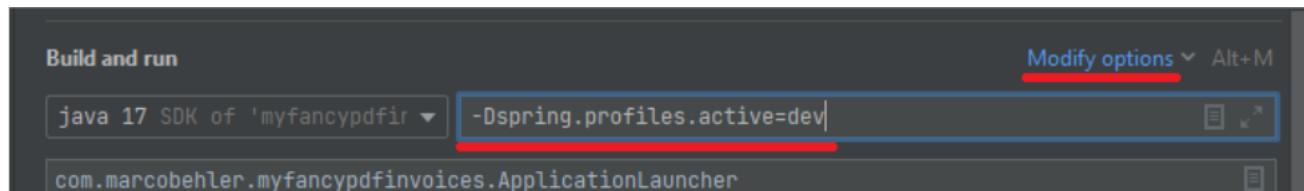


SPRING IOC

Profiles Spring

Permet de spécifier au démarrage un profil et d'adapter la configuration en fonction du profil (typiquement dev, prod ...)

On choisit le profil au démarrage :



Un bean annoté avec `@Profile("dev")` ne sera chargé qu'avec ce profil

SPRING IOC

Profiles Spring

Exemple pratique d'utilisation : charger un fichier de configuration avec l'aide du nom du profile spécifié au démarrage :

```
@PropertySource(value = "classpath:/application-  
${spring.profiles.active}.properties" , ignoreResourceNotFound =  
true)
```

Spring AOP

SPRING AOP

Programmation orienté aspects

Paradigme de programmation permettant aux développeurs de modulariser les aspects transverses des applications comme la journalisation, la sécurité ...

Permet de ne pas mélanger le code métier avec du code technique : l'ajout de nouvelles fonctionnalités se fait sans modification du code; On déclare simplement quel code doit être modifié

AspectJ est le standard Java pour faire de la programmation orienté aspects

SPRING AOP

Programmation orienté aspects

Les aspects transverses d'une application sont typiquement :

La journalisation

Les transactions

La sécurité

...

SPRING AOP

Programmation orienté aspects

Exemple

Préoccupations transverse



Méthode debiterCompte

```
// Démarrer transaction  
// Loguer paramètres  
// Vérification privilèges  
du user
```

// le code métier du débit
bancaire se positionne ici

Méthode crediterCompte

```
// Démarrer transaction  
// Loguer paramètres  
// Vérification privilèges  
du user
```

// le code métier du crédit
bancaire se positionne ici

SPRING AOP

Programmation orienté aspects

Le mélange des aspects transverses avec le code métier rend les classes

Duplication de code

Difficulté de maintenance

Difficulté de compréhension

Difficulté d'utilisation

...

SPRING AOP

Programmation orienté aspects

Faire de l'AOP permet :

- d'être focalisé pour les aspects sur une seule place

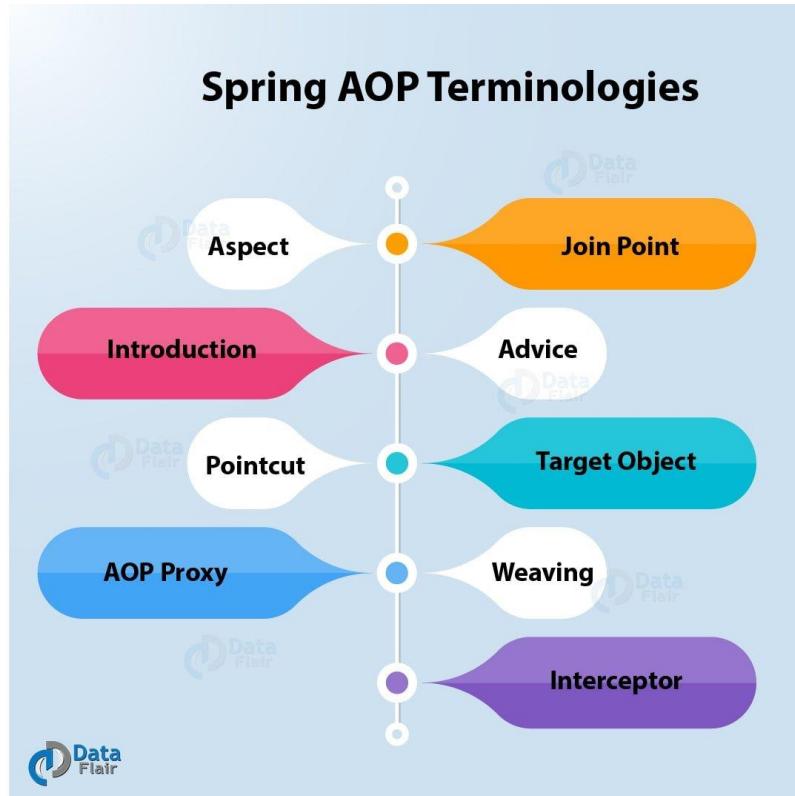
- Une facilité d'ajouter et retirer des aspects

- Une meilleure compréhension du code

...

SPRING AOP

Terminologie



SPRING AOP

Terminologie

Aspect : préoccupation transverse qu'on veut implémenter sur l'application comme la journalisation, la sécurité ...

Advice : Implémentation concrète d'un aspect. L'aspect est un concept et l'advice en est l'implémentation

JoinPoint : Point d'exécution d'un programme où un aspect doit s'appliquer : Avant ou après execution de la méthode, avant le lancement d'une exception ... Spring AOP ne supporte les aspects que sur les méthodes

Pointcut : permet de préciser pour quels joinpoints un aspect doit s'exécuter. Un aspect est associé à une expression pointcut et s'appliquera à tous les joinpoints qui satisfont cette expression

SPRING AOP

Terminologie

Target : l'objet à laquelle s'applique un advice

Proxy : Objet créé par le framework après avoir appliqué l'advice sur l'objet cible

Weaving : processus d'application d'un advice sur la target afin de produire le proxy. Le weaving peut être fait à la compilation, au chargement de classes ou au runtime. Spring fait le weaving au runtime

SPRING AOP

Aspects Spring

Spring permet d'utiliser cinq types d'advices :

Before : l'advice est exécuté avant la méthode réelle

After : l'advice est exécuté après la méthode réelle, peu importe le résultat

After-Returning : l'advice est exécuté après la méthode réelle si le résultat est sans exception

After-Throwing: l'advice est exécuté après la méthode réelle si lève une exception

Around : l'advice englobe la méthode réelle et permet d'ajouter du code avant et après la logique contenue dedans

SPRING AOP

Configuration des aspects

Ajouter des dépendances Maven de AspectJ

```
<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjtools</artifactId>
<version>1.9.20</version>
</dependency>

<dependency>
<groupId>org.aspectj</groupId>
<artifactId>aspectjrt</artifactId>
<version>1.9.20</version>
<scope>runtime</scope>
</dependency>
```

SPRING AOP

Configuration des aspects

Ajout des namespaces AOP de Spring sur le fichier de configuration

```
1.<?xml version="1.0" encoding="UTF-8"?>
2.<beans xmlns="http://www.springframework.org/schema/beans"
3.  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.  xmlns:aop="http://www.springframework.org/schema/aop"
5.  xsi:schemaLocation="http://www.springframework.org/schema/beans
6.  http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
7.  http://www.springframework.org/schema/aop
8.  http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
```

SPRING AOP

Configuration des aspects

Activation de Spring AOP sur le fichier de configuration Spring

```
<aop:aspectj-autoproxy />
```

SPRING AOP

Configuration des aspects

Création des aspects

```
package com.formation.aspects;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CustomerAspect {

    @Before("execution(public String toString())")
    public void getNameAdvice(){
        System.out.println("Execution de la méthode sur toString ");
    }

}
```

SPRING AOP

Configuration des aspects

Déclaration comme beans des aspects dans le fichier Spring

```
<bean name="customerAspect" class="com.formation.aspects.CustomerAspect" />
```

SPRING AOP

Configuration des aspects

Accès au joinpoint

```
package com.formation.aspects;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CustomerAspect {

    @Before("execution(public String toString())")
    public void getNameAdvice(JoinPoint joinPoint){
        System.out.println("Execution de la méthode sur toString ");
        System.out.println("Signature "+joinPoint.getSignature());
    }

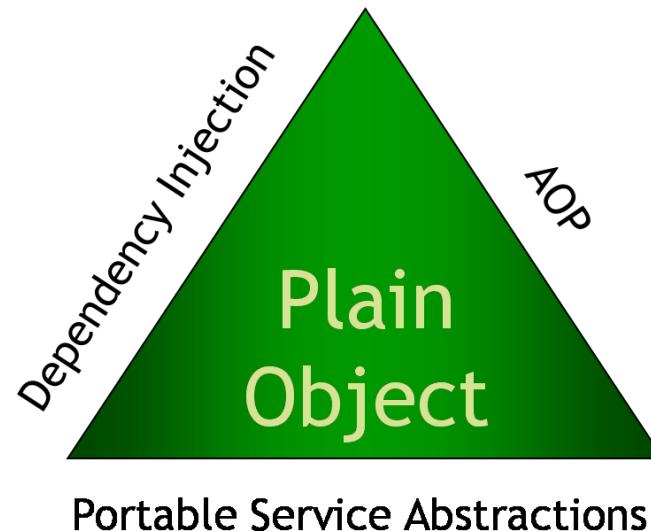
}
```

Spring JDBC

SPRING JDBC

Spring JDBC != Spring Data JDBC

Spring JDBC est une des abstractions de Spring pour se connecter aux bases de données relationnelles



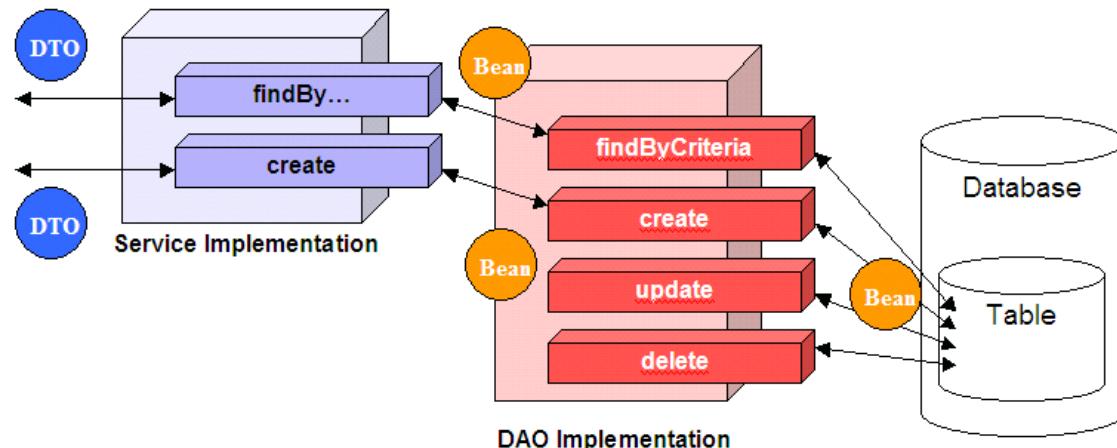
SPRING JDBC

Pattern DAO

Le modèle DAO (Data Access Object) est un pattern pour isoler les classes qui effectuent des actions en base de données

Spring implémente ce modèle avec Spring JDBC nativement et plus tard avec les projets Spring Data

Ce modèle permet de passer de l'objet au relationnel de manière assez fluide



SPRING JDBC

Exceptions JDBC

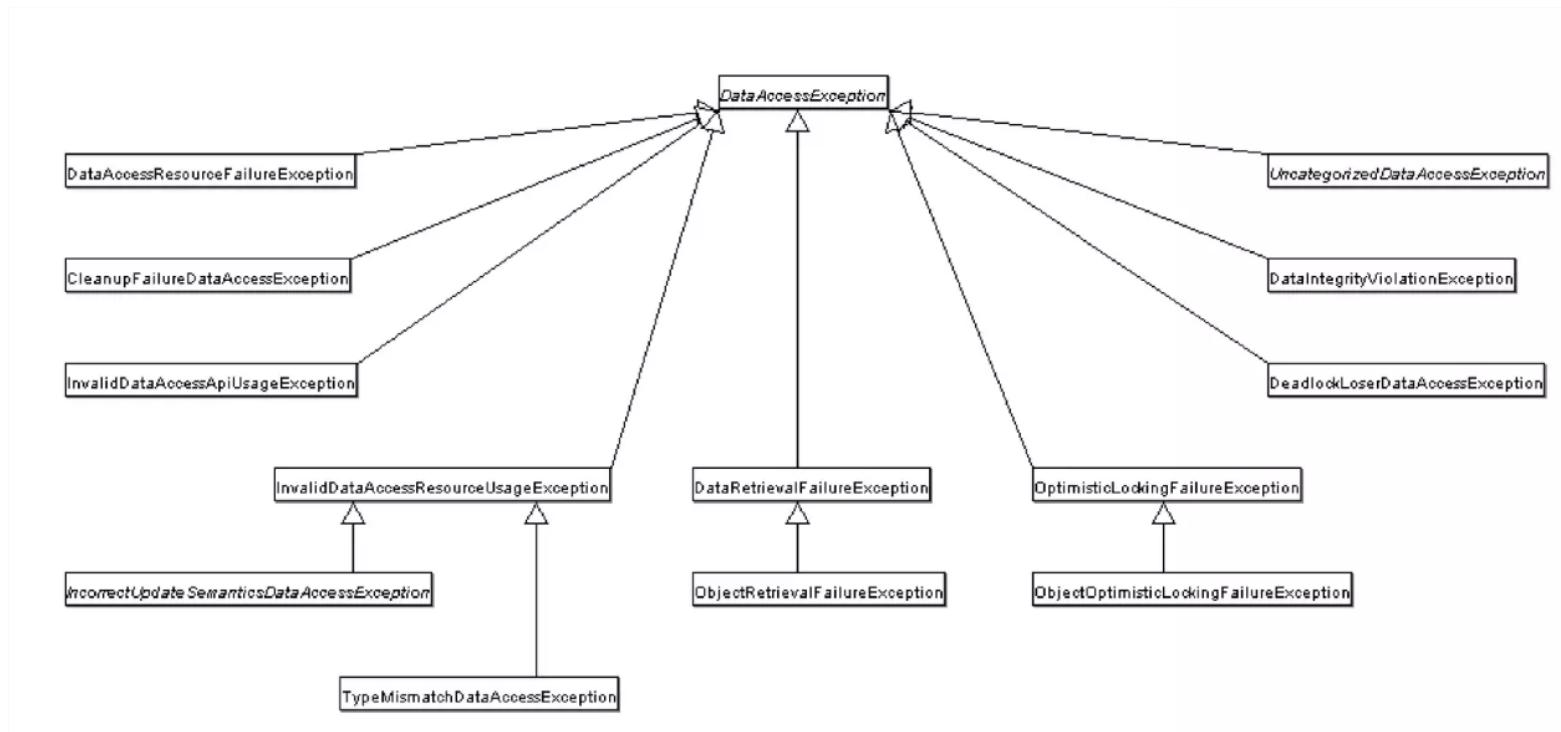
Spring traduit les exceptions spécifiques au modèle relationnel en ses propres classes d'exceptions

Ces classes de Spring englobent l'exception originale et contiendront toutes les informations pertinentes

DataAccessException est la classe principale pour les exceptions de la DAO

SPRING JDBC

Exceptions JDBC



SPRING JDBC

Annotation de la couche DAO

Spring propose l'annotation **@Repository** pour les classes de la couche DAO

Cette annotation hérite de **@Component**

Permet à Spring de traduire les exceptions JDBC

SPRING JDBC

Spring JDBC

Spring JDBC permet aux développeurs de se focaliser sur la logique métier

Il s'occupe de toutes les tâches bas niveau de l'accès aux bases de données comme : ouverture, fermeture des connexions, gestion des exceptions, itération des résultats, gestion des transactions...

SPRING JDBC

Responsabilités Spring JDBC et Développeur

	Spring JDBC	Développeur
Définir paramètres de connexion		x
Ouvrir connexion	x	
Spécifier requêtes sql		x
Déclarer et donner paramètres sql		x
Préparer et exécuter requête (statement)	x	
Mettre en place boucle pour itérer sur les résultats	x	
Traiter chaque résultat		x
Traiter les exceptions	x	
Fermer connexion, resultSet et Statement	x	

SPRING JDBC

JdbcTemplate

JDBCTemplate est l'interface de base pour utiliser Spring JDBC

C'est un classe qui englobe les fonctionnalités JDBC simples de Java et vous permet d'exécuter facilement des instructions SQL

Il a besoin d'une Datasource pour fonctionner correctement

SPRING JDBC

Datasource

Une datasource est une abstraction permettant un faible couplage entre une base de données et une application

Java propose l'interface Datasource présente dans le package `javax.sql` pour standardiser les accès à une base de donnée; les éditeurs de base données implémentent cette interface.

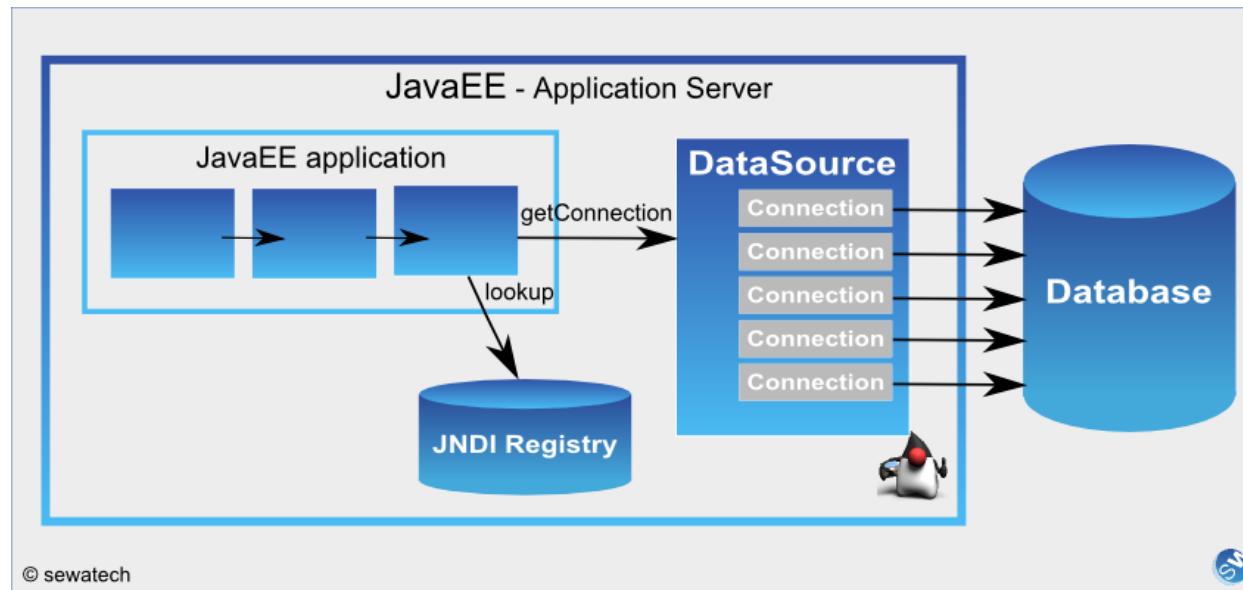
L'interface Datasource propose des méthodes basiques pour ouvrir/fermer une connexion

Les applications ne gèrent plus elles mêmes les ouvertures/fermetures mais délèguent cela à la classe de l'éditeur implémentant Datasource

SPRING JDBC

Datasource

Une datasource gère en interne un pool de connexion réutilisables



SPRING JDBC

Exemples de Datasource

MySql com.mysql.jdbc.jdbc2.optional.MysqlDataSource

Oracle oracle.jdbc.pool.OracleDataSource

PostGreSQL org.postgresql.jdbc3.Jdbc3SimpleDataSource

....

SPRING JDBC

Configuration exemple avec une base H2

Ajouter les dépendances Maven

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>2.1.214</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>6.0.9</version>
</dependency>
```

SPRING JDBC

Configuration exemple avec une base H2

Créer une datasource : H2 propose la classe **org.h2.jdbcx.JdbcDataSource** qui implémente l'interface Datasource de Java

```
import org.h2.jdbcx.JdbcDataSource;  
....  
  
@Bean  
public DataSource dataSource() {  
    JdbcDataSource ds = new JdbcDataSource();  
    ds.setURL("jdbc:h2:~/myFirstH2Database;INIT=RUNSCRIPT FROM  
'classpath:schema.sql'");  
    ds.setUser("sa");  
    ds.setPassword("sa");  
    return ds;  
}
```

SPRING JDBC

Configuration exemple avec une base H2

Créer le script d'initialisation de la base H2

Il faut initialiser la base de données avec au moins une table

Pour une application Maven, on peut créer un fichier SQL sur
src/main/resources contenant la requête de création de la base de données

Il faut mettre à jour la configuration de la datasource pour localiser le script (à faire en TP)

```
ds.setURL("jdbc:h2:~/ma_base;INIT=RUNSCRIPT FROM  
'classpath:schema.sql'");
```

SPRING JDBC

Configuration exemple avec une base H2

Créer le bean Spring JdbcTemplate (prend en paramètre la datasource)

```
@Bean public JdbcTemplate jdbcTemplate()  
{  
    return new JdbcTemplate(dataSource());  
}
```

SPRING JDBC

Configuration exemple avec une base H2

Exemple utilisation de JdbcTemplate

```
public List<Invoice> findAll() {  
  
    return jdbcTemplate.query("select id, user_id, pdf_url, amount from invoices",  
(resultSet, rowNum) -> {  
        Invoice invoice = new Invoice();  
        invoice.setId(resultSet.getObject("id").toString());  
        invoice.setPdfUrl(resultSet.getString("pdf_url"));  
        invoice.setUserId(resultSet.getString("user_id"));  
        invoice.setAmount(resultSet.getInt("amount"));  
        return invoice;  
});  
}
```

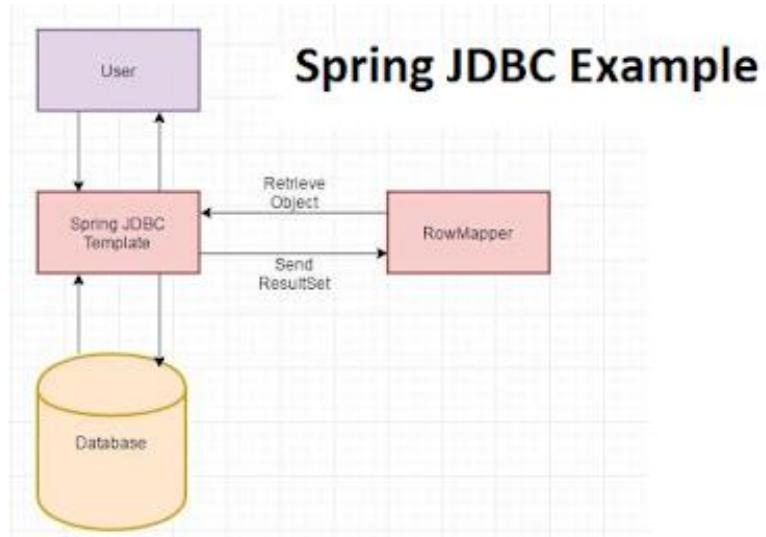
SPRING JDBC

Configuration exemple avec une base H2

Interface **RowMapper**

Pemet de mapper les résultats SQL sur des classes Java

A implémenter par le développeur pour transformer une ligne SQL en objet Java



SPRING JDBC

Exemples de query

Retourne le nombre de résultats

```
int rowCount = this.jdbcTemplate.queryForInt("select count(0) from t_article");
```

```
int nbreDeGeorges= this.jdbcTemplate.queryForInt("select count(0) from t_users  
where first_name = ?", new Object[]{"Georges"})
```

```
String prenom= (String) this.jdbcTemplate .queryForObject("select prenom from  
t_userwhere id = ?", new Object[]{new Long(1234)}, String.class);
```

SPRING JDBC

Exemples de query

Retourne un objet particulier

```
User user = this.jdbcTemplate.queryForObject("select first_name, last_name from  
t_user where id = ?",
new Object[] { 1234 }, new RowMapper<User>() {
    public User mapRow(ResultSet rs, int rowNum) throws SQLException {
        User user = new User();
        user.setFirstName(rs.getString("first_name"));
        user.setLastName(rs.getString("last_name"));
        return user;
    }
});
```

SPRING JDBC

Exemples de query

Retourne une liste d'objets

```
List<User> users = this.jdbcTemplate.query(  
    "select first_name, last_name from t_user",  
    new RowMapper<User>() {  
        public User mapRow(ResultSet rs, int rowNum) throws SQLException {  
            User user = new User();  
            user.setFirstName(rs.getString("first_name"));  
            user.setLastName(rs.getString("last_name"));  
            return user;  
        }  
    });
```

SPRING JDBC

Autres classes

NamedParameterJdbcTemplate permet l'utilisation de paramètres nommés à la différence des placeholders (?)

```
String sql = "select count(*) from T_User where first_name = :first_name";  
  
SqlParameterSource namedParameters = new MapSqlParameterSource("first_name",  
firstName);  
  
return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
```

SimpleJdbcTemplate profite des évolutions de Java 5

Transactions

TRANSACTIONS

Transactions

Les transactions sont une partie importante des SGBBs permettant d'assurer l'intégrité et la consistante des opérations. Elles suivent les principes ACID

Atomique : une transaction doit être traité comme une opération **unitaire**; soit tout réussi, soit on annule tout

Cohérente : cohérence dans l'intégrité des données

Isolée : isolation par rapport à aux d'autres opérations

Durable : les résultats d'une transaction sont durables

TRANSACTIONS

Transactions

Spring propose une abstraction pour la gestion des transactions avec notamment l'annotation `@Transactional`

Spring permet de gérer les transactions de manière déclarative ou programmatique

La manière programmatique permet une grande flexibilité mais est difficile à maintenir

La manière déclarative est à privilégier car tout se fait via des annotations ou XML

TRANSACTIONS

Gestion des transactions avec JDBC

```
import java.sql.Connection;  
  
Connection connection = dataSource.getConnection(); // (1)  
  
try (connection) {  
    connection.setAutoCommit(false); // (2)  
    // execute some SQL statements...  
    connection.commit(); // (3)  
  
} catch (SQLException e) {  
    connection.rollback(); // (4)  
}
```

TRANSACTIONS

@Transactional

Permet de faire exactement la même chose que les étapes manuelles
Exemple cette annotation :

```
@Transactional(propagation=TransactionDefinition.NESTED,  
isolation=TransactionDefinition.ISOLATION_READ_UNCOMMITTED)
```

Produira ce code :

```
import java.sql.Connection;  
  
// isolation=TransactionDefinition.ISOLATION_READ_UNCOMMITTED  
  
connection.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED); // (1)  
  
// propagation=TransactionDefinition.NESTED  
  
Savepoint savePoint = connection.setSavepoint(); // (2)  
...  
connection.rollback(savePoint);
```

TRANSACTIONS

Gestion programmatique des transactions

Utilisation de la classe **TransactionTemplate**

```
@Service
public class UserService {

    @Autowired
    private TransactionTemplate template;

    public Long registerUser(User user) {
        Long id = template.execute(status -> {
            // execute some SQL that e.g.
            // inserts the user into the db and returns the autogenerated id
            return id;
        });
    }
}
```

TRANSACTIONS

Gestion déclarative des transactions

- 1 - Ajouter `@EnableTransactionManagement` sur la classe de configuration Spring
- 2 - Ajouter un gestionnaire de transactions
- 3 - Annoter les classes ou méthodes avec `@Transactional`

```
public class UserService {  
  
    @Transactional  
    public Long registerUser(User user) {  
        // execute some SQL that e.g.  
        // inserts the user into the db and retrieves the autogenerated id  
        // userDao.save(user);  
        return id;  
    }  
}
```

TRANSACTIONS

Gestion déclarative des transactions

Classe de configuration

```
@Configuration  
@EnableTransactionManagement  
public class MySpringConfig {  
  
    @Bean  
    public PlatformTransactionManager txManager() {  
        return yourTxManager;  
    }  
}
```

TRANSACTIONS

Gestion déclarative des transactions

Code généré par Spring

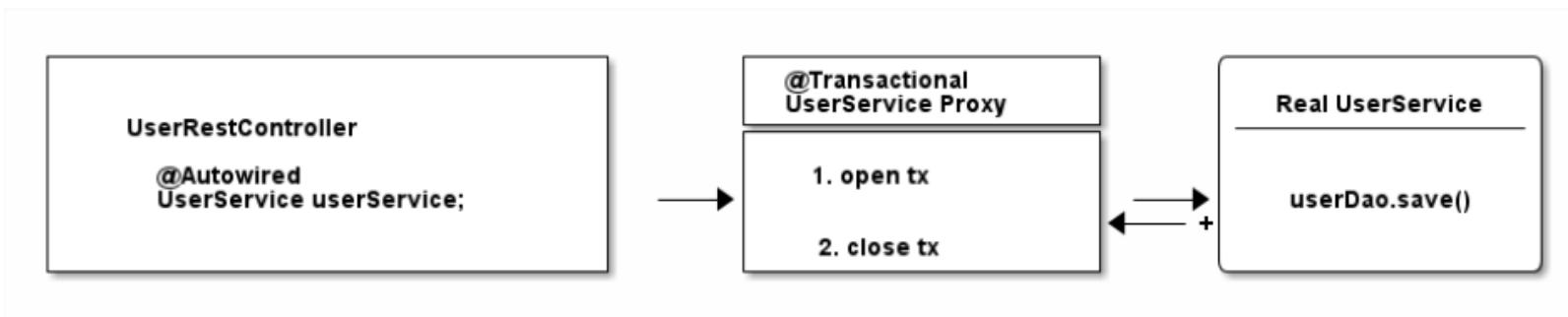
```
public class UserService {  
  
    public Long registerUser(User user) {  
        Connection connection = dataSource.getConnection(); // (1)  
        try (connection) {  
            connection.setAutoCommit(false); // (1)  
  
            // execute some SQL that e.g.  
            // inserts the user into the db and retrieves the autogenerated id  
            // userDao.save(user); <(2)>  
  
            connection.commit(); // (1)  
        } catch (SQLException e) {  
            connection.rollback(); // (1)  
        }  
    }  
}
```

TRANSACTIONS

Comment marche les transactions avec Spring ?

Spring utilise l'AOP pour générer un proxy autour de vos classes

Ce proxy ajoute les instructions pour gérer les transactions



TRANSACTIONS

Transaction Manager

Les proxy générés délèguent la gestion des transactions au transaction manager

Spring propose une interface **PlatformTransactionManager** avec quelques implémentations

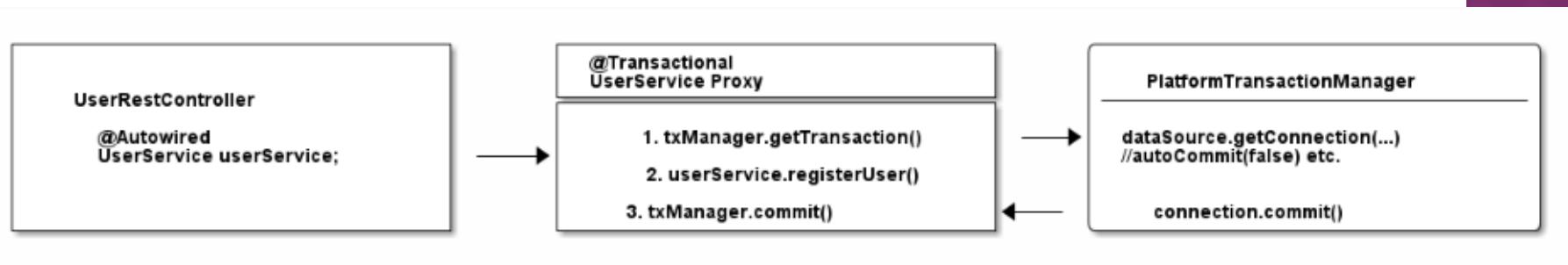
Exemple : **DataSourceTransactionManager**

```
@Bean  
public DataSource dataSource() {  
    return new MysqlDataSource(); // (1)  
}
```

```
@Bean  
public PlatformTransactionManager txManager() {  
    return new DataSourceTransactionManager(dataSource()); // (2)
```

TRANSACTIONS

Transaction Manager



TRANSACTIONS

Résumé :

1. Spring détecte l'annotation `@Transactional` sur un objet et crée un proxy dynamique sur l'objet
2. Le proxy a accès au transaction manager et lui demande d'ouvrir, de créer des transactions
3. Le transaction manager utilise simplement les APIs Java JDBC pour réaliser les actions

TRANSACTIONS

Propagation

Attribut de l'annotation `@Transactional` pour gérer les imbrications de transactions dans le code

```
@Transactional(propagation = Propagation.REQUIRED)  
@Transactional(propagation = Propagation.REQUIRES_NEW)
```

Liste des valeurs possibles

- REQUIRED
- SUPPORTS
- MANDATORY
- REQUIRES_NEW
- NOT_SUPPORTED
- NEVER
- NESTED

TRANSACTIONS

Propagation

Required (par défaut) : la méthode a besoin d'une transaction; utiliser une existante ou en créer une nouvelle

SUPPORTS : peu importe si une transaction existe ou pas, le code marchera bien

Mandatory : la méthode a besoin d'une transaction (une transaction doit exister à l'appel de la méthode)

Require_new : la méthode a besoin d'une nouvelle transaction

Not_Supported : la méthode ne supporte pas les transactions

Never : ne jamais ouvrir de transaction

Nested : concept assez avancée

TRANSACTIONS

Isolation

Le niveau d'isolation permet à une base de données de définir le degré de verrouillage des éléments lors d'une opération
Il est configurable et propre à chaque base de données

Quatre niveaux :

SERIALIZABLE

REPEATABLE READS

READ COMMITTED

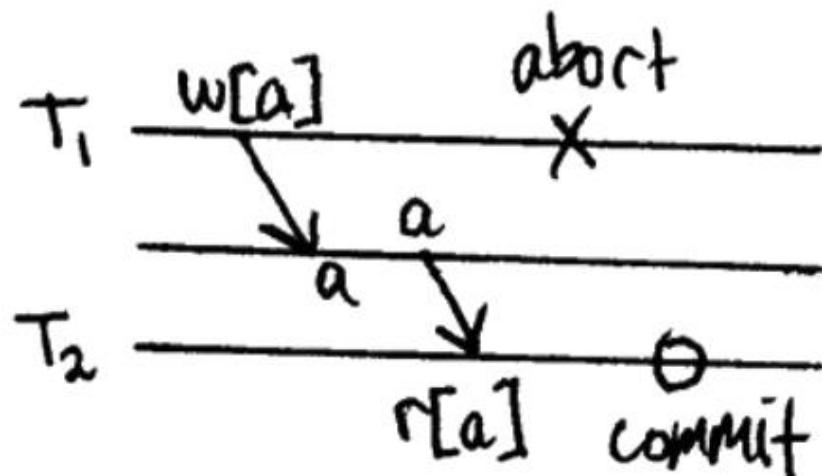
READ UNCOMMITTED

Ces 4 niveaux ont été définis en fonction des comportements notés lors de transactions concurrentes (**READ DIRTY, Non-Repeatable Reads, Phantom Read, Write Skew**)

TRANSACTIONS

Isolation

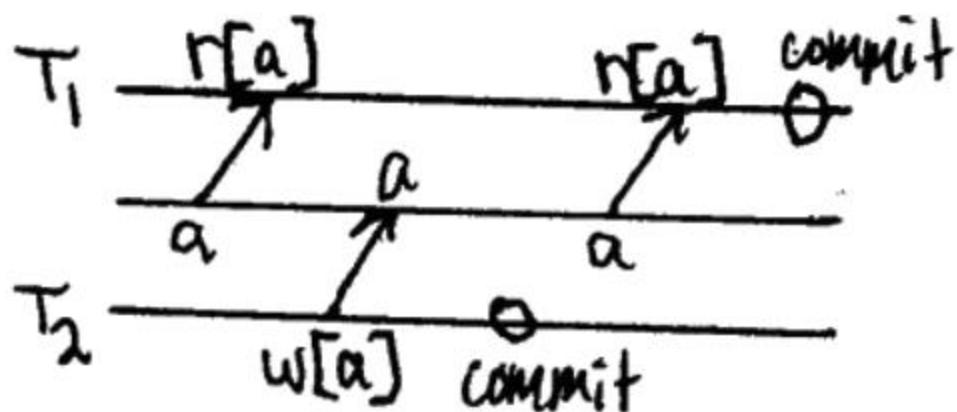
READ DIRTY



TRANSACTIONS

Isolation

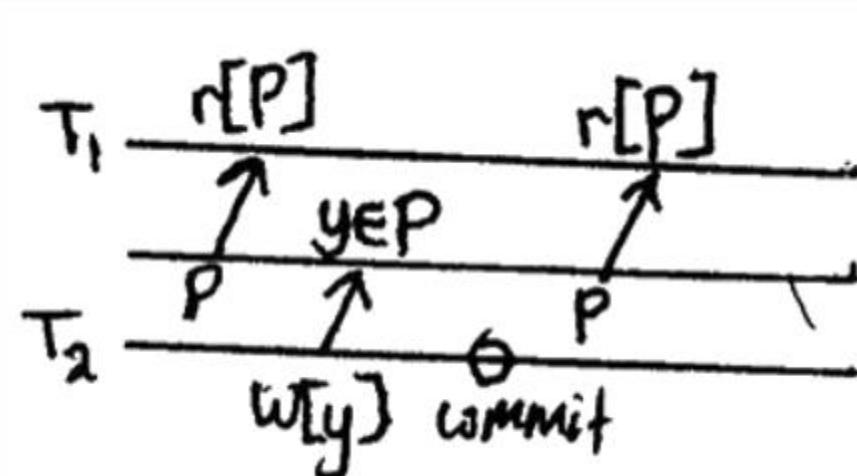
Non-Repeatable Reads



TRANSACTIONS

Isolation

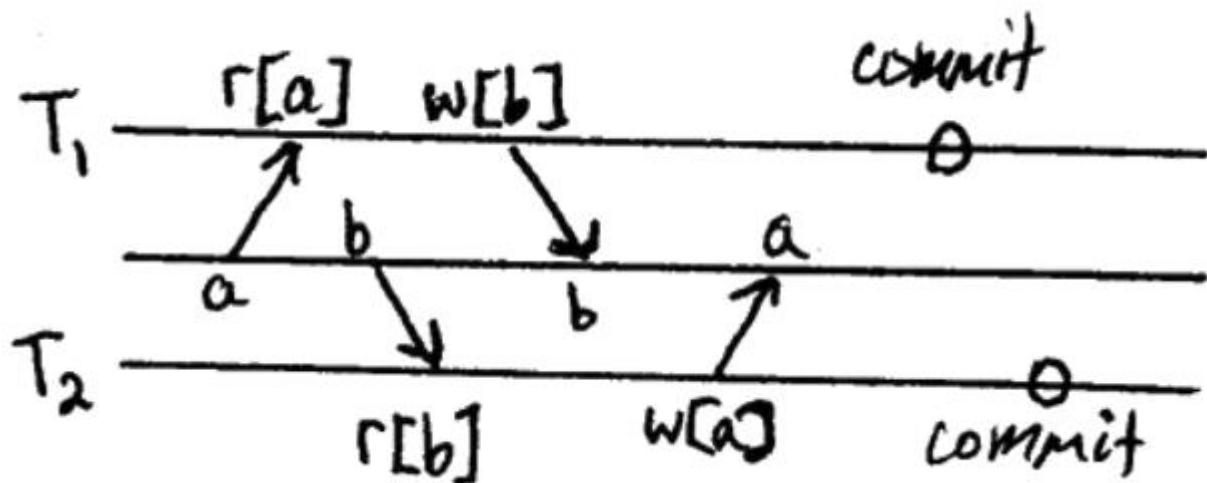
Phantom Read



TRANSACTIONS

Isolation

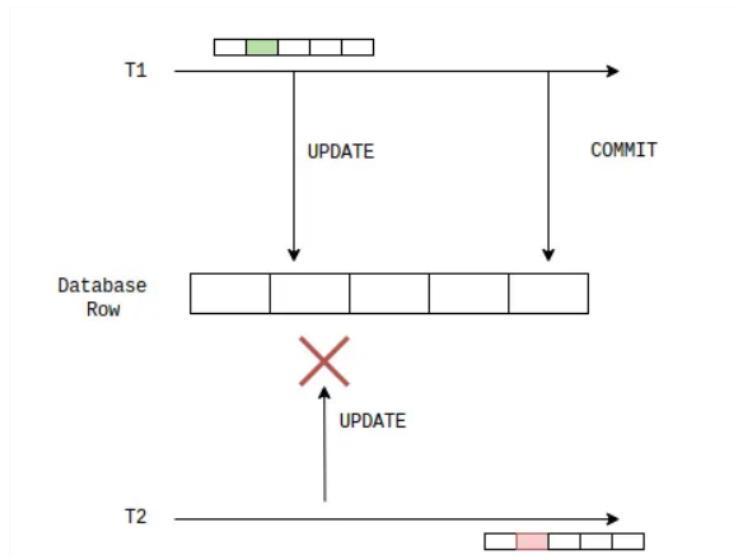
Write Skew



TRANSACTIONS

READ UNCOMMITTED

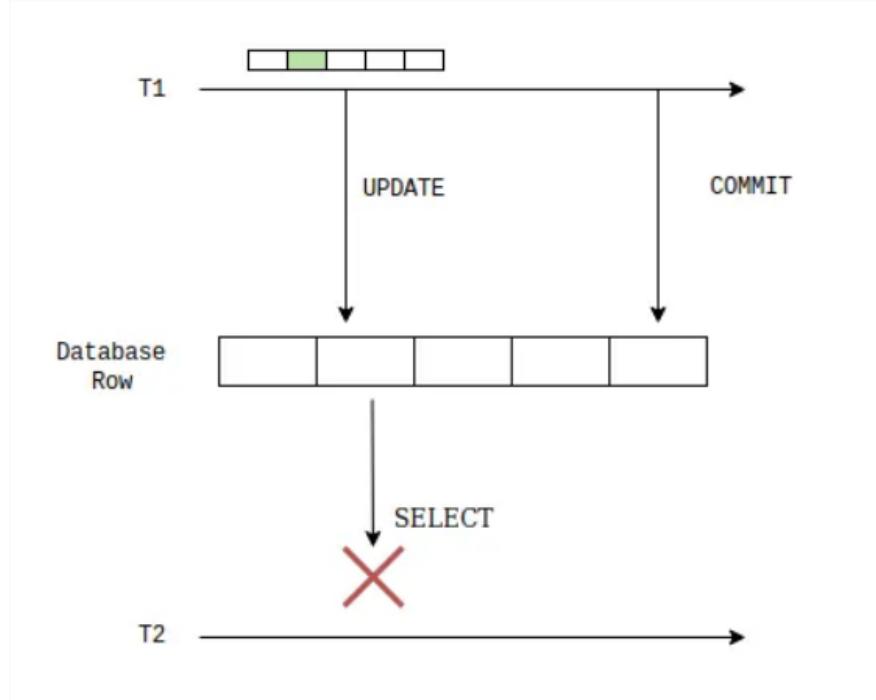
Niveau le plus bas. Mise à jour d'une ligne dans la base de données interdite si une autre transaction l'a déjà mise à jour et nom comitté. Cela protège contre les mises à jour perdues, mais ne protégera pas les lectures sales



TRANSACTIONS

READ COMMITTED

Mise à jour et lecture d'une ligne dans la base de données interdite si une autre transaction l'a déjà mise à jour et non comitté.



TRANSACTIONS

SERIALIZABLE

Niveau le plus restrictif; permet de verrouiller une table entière par une transaction

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	Yes	Yes	Yes
Read committed	No	Yes	Yes
Repeatable read	No	No	Yes
Snapshot	No	No	No
Serializable	No	No	No

TRANSACTIONS

Isolation avec Spring

Spring permet de changer le niveau d'isolement au cours d'une transaction. Vous devez vous assurer de consulter votre pilote/base de données JDBC pour comprendre quels scénarios sont pris en charge et lesquels ne le sont pas.

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
```

Spring MVC Rest Services

REST SERVICES

Spring MVC est bâti au dessus de l'API Servlet de Jakarta

Permet de réaliser services REST et propose des fonctionnalités avancées de Parsing et de conversion

Dépendances Maven

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>6.0.9</version>
</dependency>
```

REST SERVICES

Controller

Spring propose une abstraction au dessus de HttpServlet
appelés **Controller**

Les contrôleurs reçoivent et traitent les requêtes HTTP

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;
```

@Controller

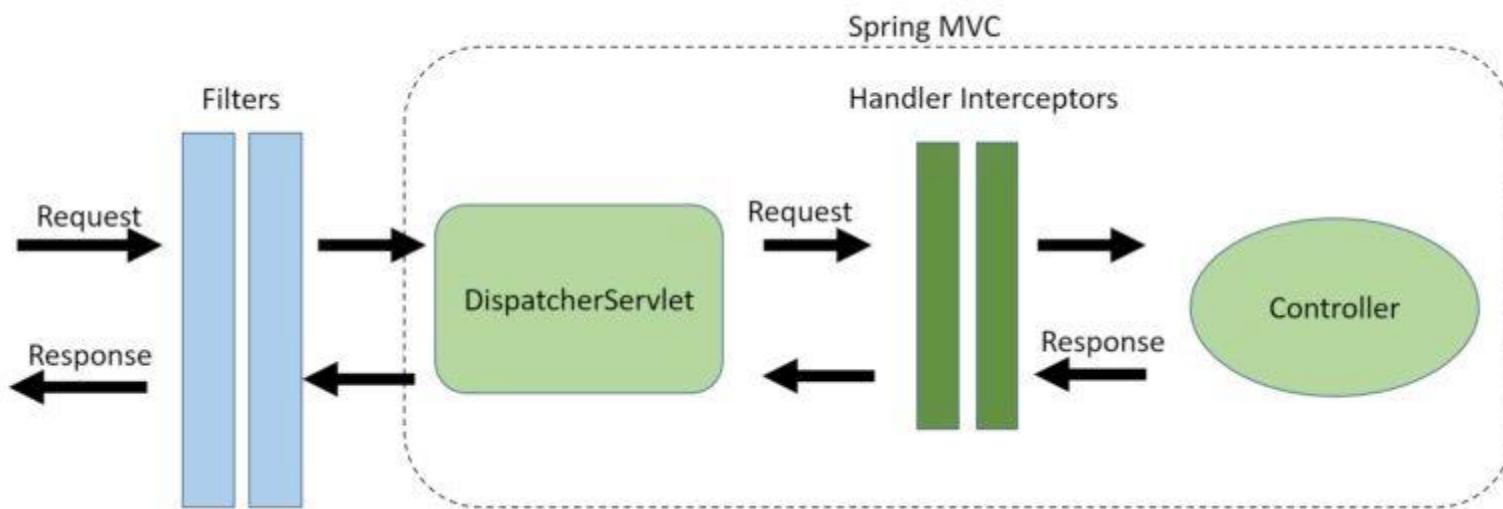
```
public class MyFirstSpringController {
```

```
    @GetMapping("/")
    @ResponseBody
    public String index() {
        return "Hello World";
    }
}
```

REST SERVICES

DispatcherServlet

Spring propose une classe DispatcherServlet pour qui va être le frontal qui reçoit les requêtes et les dispatche aux contrôleurs



REST SERVICES

DispatcherServlet

Configuration : Dans le cas d'une application web, il faut configurer cette servlet dans le fichier web.xml (ou annotations)

Exemple d'une application avec un tomcat embedded

```
WebApplicationContext appCtx = createApplicationContext(tomcatCtx.getServletContext());
DispatcherServlet dispatcherServlet = new DispatcherServlet(appCtx);
Wrapper servlet = Tomcat.addServlet(tomcatCtx, "dispatcherServlet",
dispatcherServlet);
servlet.setLoadOnStartup(1);
servlet.addMapping("/*");

tomcat.start();
```

REST SERVICES

@RestController

@Controller permet de notifier à Spring qu'une classe peut traiter des requêtes HTTP et retourner le résultat sous format HTML

@RestController est une annotation qui implémente **@Controller** et **@ResponseBody**; ce dernier permet d'écrire directement dans le body de la réponse du JSON, XML ou texte sans passer par un moteur de templates (ce que attend Spring par défaut)

REST SERVICES

@GETMAPPING

```
@GetMapping("/users")
public List<Users> users()
```

A chaque appel de /users, la DispatcherServlet va transférer cette appel à la method

```
@GetMapping("/users")
```

est un raccourci de :

```
@RequestMapping(value = "/users", method =
RequestMethod.GET)
```

REST SERVICES

@EnableWebMvc

Permet d'activer WebMVC et les conversions associées

```
import org.springframework.web.servlet.config.annotation.EnableWebMvc;

@Configuration
@ComponentScan(basePackageClasses = ApplicationLauncher.class)
@PropertySource("classpath:/application.properties")
@PropertySource(value = "classpath:/application-${spring.profiles.active}.properties"
, ignoreResourceNotFound = true)
@EnableWebMvc
public class MyApplicationConfiguration {

    @Bean
    public ObjectMapper objectMapper() {
        return new ObjectMapper();
    }
}
```

REST SERVICES

@EnableWebMvc

Permet d'activer WebMVC et les conversions associées (Ajouter la dépendance Jackson qui permet de réaliser des conversions Json entre autres)

```
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
```

```
@Configuration  
@ComponentScan(basePackageClasses = ApplicationLauncher.class)  
@PropertySource("classpath:/application.properties")  
@PropertySource(value = "classpath:/application-${spring.profiles.active}.properties"  
, ignoreResourceNotFound = true)
```

```
@EnableWebMvc  
public class MyApplicationConfiguration {
```

```
@Bean  
public ObjectMapper objectMapper() {  
    return new ObjectMapper();  
}
```

REST SERVICES

@PostMapping

Permet de réaliser des traitements POST

@RequestParam

Permet de mapper des paramètres de requêtes

```
public User createInvoice(@RequestParam("user_id") String userId, @RequestParam Integer age)
```

@PathVariables

Permet de chercher les paramètres le path de la requête

```
@PostMapping("/invoices/{userId}/{amount}")
public Invoice createUser(@PathVariable String userId, @PathVariable Integer age) {
    return invoiceAge.create(userId, age);
}
```

REST SERVICES

Utilisation de DTOs (Data Transfer Object)

Permet d'envoyer du json en entrée d'un contrôleur, qui le récupérera avec une classe Java appelé DTO

POST http://localhost:8080/users

Content-Type: application/json

####

{

 "age": 32,

 "user_id": "helene "

}

REST SERVICES

Utilisation de DTOs (Data Transfer Object)

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class UserDto {

    @JsonProperty("user_id")
    private String userId;

    @Min(10)
    @Max(50)
    private Integer age;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

REST SERVICES

Utilisation de DTOs (Data Transfer Object)

```
@PostMapping("/users")
public Invoice createUser(@RequestBody UserDto userDto) {
    return usereService.create(userDto.getUserId(), userDto.getAge());
}
```

Bean Validation

JBoss Seam 3.0

BEAN VALIDATION

Spring propose des fonctionnalités de validation rudimentaires

Exemple : conversion String en Integer pour des paramètres de service

La JSR 303 est le standard Java pour faire de la validation de beans

Hibernate-validator est l'implémentation de référence

BEAN VALIDATION

Dépendances Maven Hibernate-validator

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>8.0.0.Final</version>
</dependency>
```

```
<dependency>
    <groupId>org.glassfish.expressly</groupId>
    <artifactId>expressly</artifactId>
    <version>5.0.0</version>
</dependency>
```

BEAN VALIDATION ANNOTATIONS

```
import com.fasterxml.jackson.annotation.JsonProperty;

public class UserDto {

    @JsonProperty("user_id")
    @NotNull
    private String userId;

    @Min(10)
    @Max(50)
    private Integer age;

    public String getUserId() {
        return userId;
    }

    public void setUserId(String userId) {
        this.userId = userId;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }
}
```

BEAN VALIDATION ANNOTATIONS

- @AssertFalse / @AssertTrue
- @DecimalMin / @DecimalMax
- @Digits
- @Email
- @Min / @Max
- @Negative / @NegativeOrZero
- @NotBlank / @NotEmpty
- @Null / @NotNull
- @Past / @PastOrPresent
- @Pattern
- @Positive / @PositiveOrZero
- @Size

BEAN VALIDATION ANNOTATIONS

Validation DTO avec `@Valid`

```
public Invoice createUser (@RequestBody @Valid UserDto userDto)
```

BEAN VALIDATION ANNOTATIONS

Validation des paramètres de requêtes

```
@PostMapping("/users")
public User createUser(@RequestParam("user_id") @NotBlank String
userId, @RequestParam @Min(1) @Max(130) Integer age) {
    return userService.create(userId, age);
}
```

Il faut au préalable ajouter quelques configurations ...

BEAN VALIDATION ANNOTATIONS

Configuration pour activer la validation sur les paramètres de requêtes

1 - Annoter la classe de configuration avec `@Validated`

2 – Ajouter un nouveau bean `MethodValidationPostProcessor`

```
@Bean
public MethodValidationPostProcessor methodValidationPostProcessor() {
    return new MethodValidationPostProcessor();
}
```

Spring MVC Exceptions

EXCEPTIONS

Spring permet de remonter ses propres exceptions lors d'erreurs dans les appels REST

Il créer une classe annotée avec `@RestControllerAdvice` ou `@ControllerAdvice`

EXCEPTIONS

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import jakarta.validation.ConstraintViolationException;

@RestControllerAdvice
public class GestionnaireGlobalExceptionHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public String handleMethodArgumentNotValid(MethodArgumentNotValidException exception) { //
        // TODO on peut retourner tout ce qu'on veut .
        return "Désolé erreur dans l'appel: " + exception.getMessage();
    }

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(ConstraintViolationException.class)
    public String handleConstraintViolation(ConstraintViolationException exception) { //
        // TODO on peut retourner tout ce qu'on veut ..
        return "Désolé erreur dans l'appel : " + exception.getMessage();
    }
}
```

EXCEPTIONS

@ResponseStatus(HttpStatus.BAD_REQUEST) : permet de préciser le code de retour

@ExceptionHandler(ConstraintViolationException.class) : permet de déterminer le type d'exception à personnaliser

Spring MVC WEB

SPRING MVC

Spring MVC permet de retourner des pages HTML à l'aide d'un moteur de vue

Spring supporte plusieurs moteurs :

Thymeleaf

Velocity

FreeMarker

JSP

...

Les contrôleurs devant retourner du html via un moteur de vues doivent être annotés avec `@Controller`

SPRING MVC

Exemple avec Thymleaf

Dépendances

```
<dependency>
    <groupId>org.thymeleaf</groupId>
    <artifactId>thymeleaf-spring6</artifactId>
<version>3.1.1.RELEASE</version>
</dependency>
```

SPRING MVC

Exemple avec Thymleaf

Il faut définir un certain nombre de beans pour permettre à Thymeleaf de s'intégrer correctement avec Spring

Un Bean permettant à Spring de trouver les vues Thymeleaf
ThymeleafViewResolver

ThymeleafViewResolver doit être configuré avec une instance de la classe **SpringTemplateEngine**. Ce dernier fera le lien Thymeleaf et Spring

Un résolveur de Templates

SPRING MVC

```
@Bean  
public ThymeleafViewResolver viewResolver() {  
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();  
    viewResolver.setTemplateEngine(templateEngine());  
  
    viewResolver.setOrder(1); // optional  
    viewResolver.setViewNames(new String[] {"*.html", "*.xhtml"}); //  
    optional  
    return viewResolver;  
}
```

SPRING MVC

```
@Bean
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    return templateEngine;
}
```

SPRING MVC

```
@Bean
public SpringResourceTemplateResolver templateResolver() {
    _____
    SpringResourceTemplateResolver templateResolver = new
    SpringResourceTemplateResolver();
    templateResolver.setPrefix("classpath:/templates/");
    templateResolver.setCacheable(false);
    return templateResolver;
}
```

SPRING MVC

Exemple de template thymeleaf

```
<!DOCTYPE_html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Title</title>
</head>
<body>
<p>Hello <span th:text="${username}" th:remove="tag">[Username]</span>,
this is the current date <span
th:text="#{temporals.format(currentDate, 'dd-MM-yyyy HH:mm')}">
[currentDate]</span>
</p>
<div th:if="${username.startsWith('z')}">Your name starts with a
lower-case z, unusual!</div>
</body>
</html>
```

SPRING MVC

Mise à jour du modèle

```
@GetMapping("/")
public String homepage(Model model, @RequestParam(required = false,
defaultValue = "stranger") String username) {
    model.addAttribute("username", username);
    model.addAttribute("currentDate", LocalDateTime.now());
    return "index.html";
}
}
```

SPRING MVC

Mise à jour du modèle

Model est un Map de paramètres contenant toutes les variables qui seront disponibles dans la vue

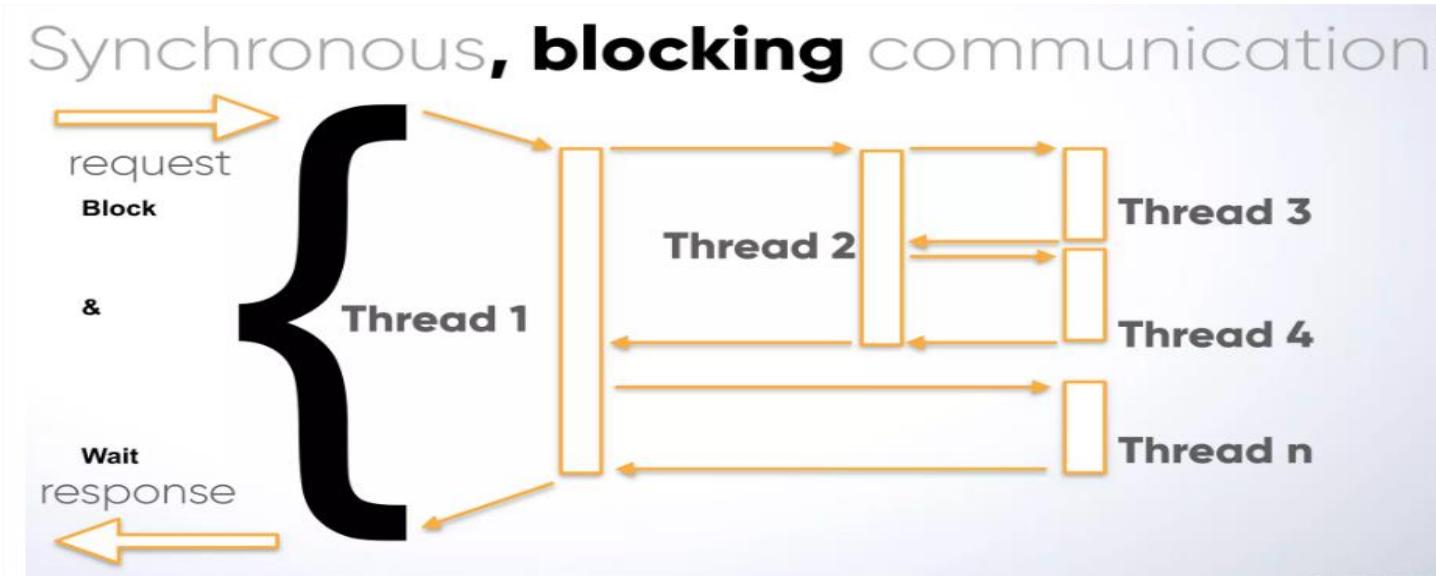
```
@GetMapping("/")
public String homepage(Model model, @RequestParam(required = false,
defaultValue = "stranger") String username) {
    model.addAttribute("username", username);
    model.addAttribute("currentDate", LocalDateTime.now());
    return "index.html";
}
}
```



Spring WEBFLUX

SPRING WEBFLUX

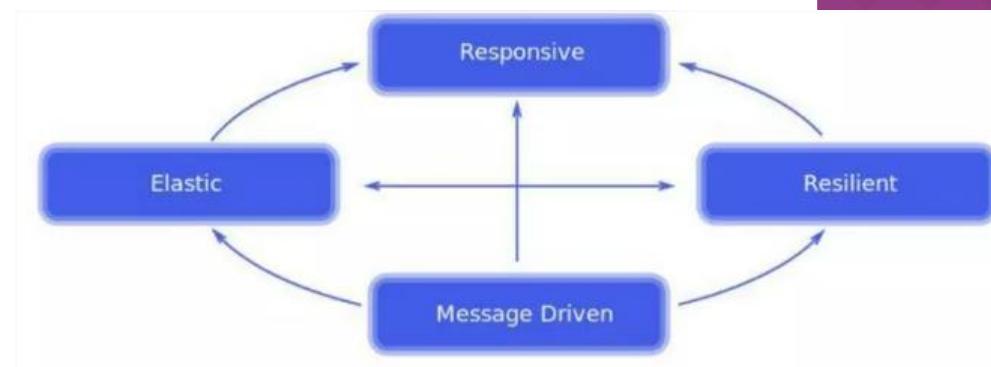
Paradigme de gestion classique des requêtes



SPRING WEBFLUX

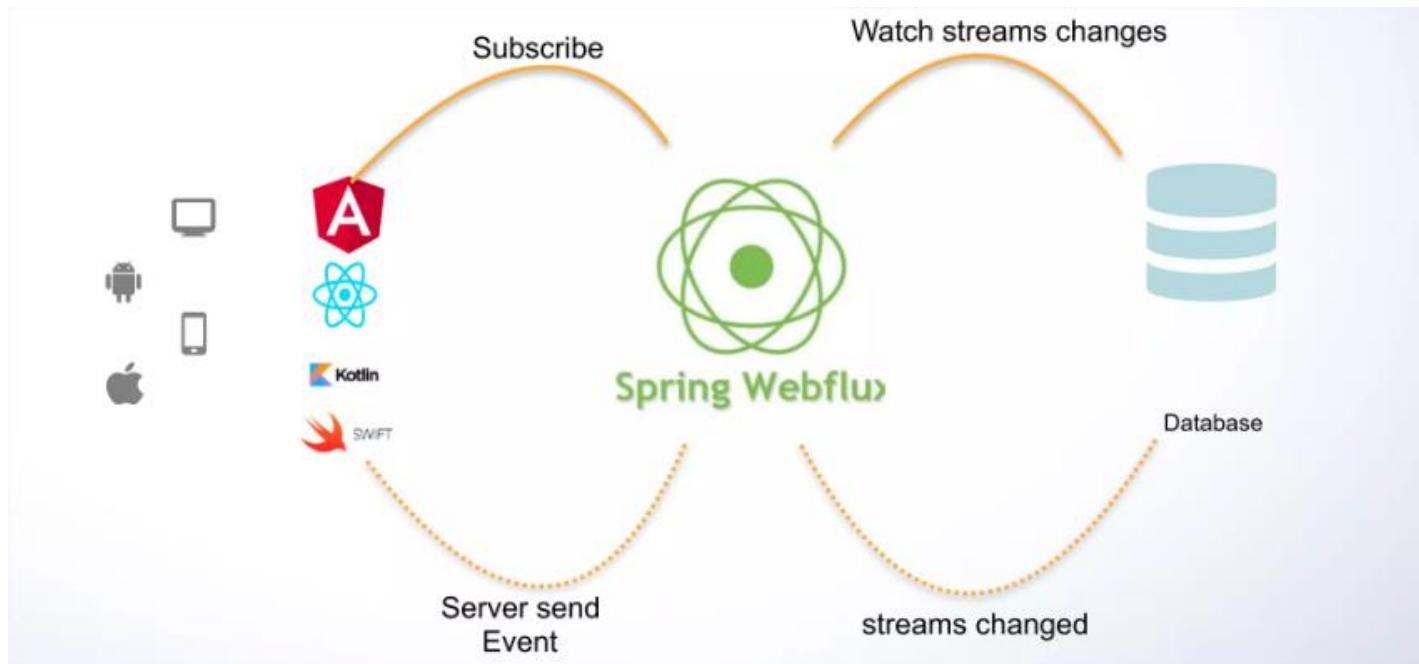
Permet de faire de la programmation réactive :

- Applications non bloquantes
- Asynchrones
- Piloté par les événements
- Minimum de threads pour monter en charge verticalement (JVM) plutôt que de manière horizontale (par clustering)



SPRING WEBFLUX

Propagation des changements via des événements



SPRING WEBFLUX

Exemples d'utilisation de la programmation réactive

- Appels à des services externes
- Feuilles de calcul
- Applications de type « feed » avec une consommation importante et concurrente de données
- ...

SPRING WEBFLUX

Spring webflux introduit dans Spring 5

Framework Web réactif et non bloquant pour la création d'applications Web modernes et évolutives en Java

Utilise la bibliothèque Reactor

- Coexiste avec Spring MVC mais pour des utilisations différentes

PROGRAMMATION RÉACTIVE

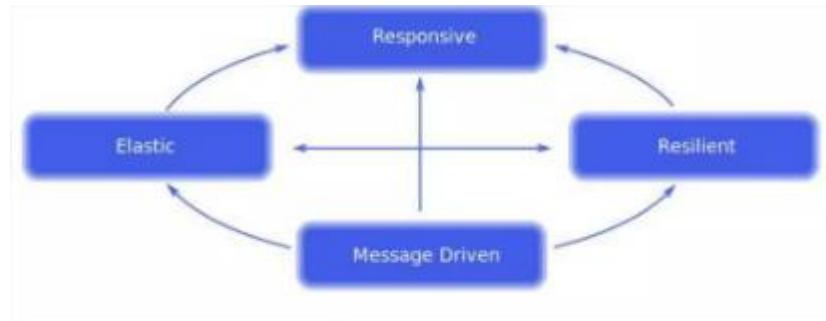
Paradigme de programme permettant la conception d'applications :

- Asynchrone
- Piloté par les événements
- avec un minimum de threads et une mise à l'échelle verticale (au sein de la JVM même)

PROGRAMMATION RÉACTIVE

Une application réactive

- réagit aux événements (piloté par les événements)
- réagit à la fluctuation de la charge (scalabilité)
- réagit aux pannes (résilience)
- réagit aux actions des utilisateurs (responsive)



PROGRAMMATION RÉACTIVE

Opère sur des streams de données

Les streams de données peuvent être :

- une action d'un utilisateur (ex : click sur un bouton)
- une réponse à un appel d'une API (ex : un flux facebook)
- les données d'une base
- ...

Inspiré du manifeste réactive <https://reactivemanifesto.org/>

SPÉCIFICATION STREAMS JVM

Spécification pour la JVM afin de construire des applications réactives

Consiste en deux parties :

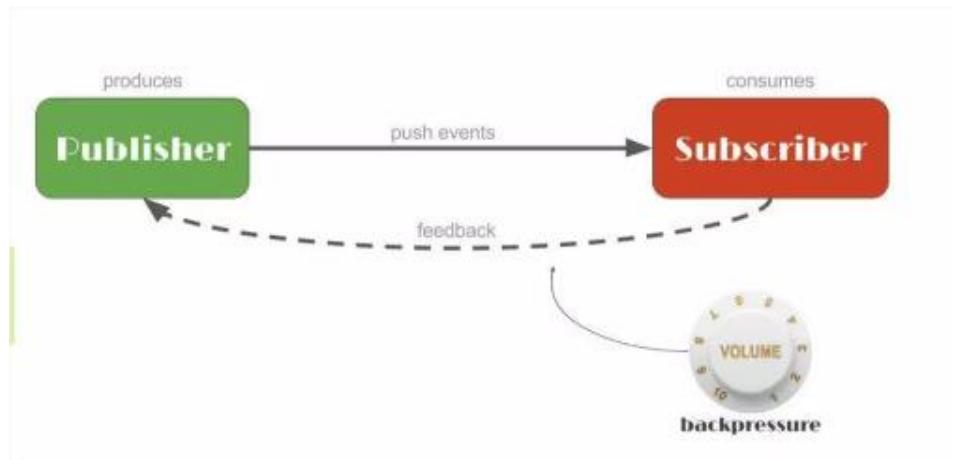
- Une API pour les types à implémenter
- Un kit de testing pour vérifier la conformité selon le standard (TCK Technology Compatibility Kit)

SPÉCIFICATION STREAMS - BACK PRESSURE

Permet de contrôler la quantité de données lors d'un échange

Régulation entre :

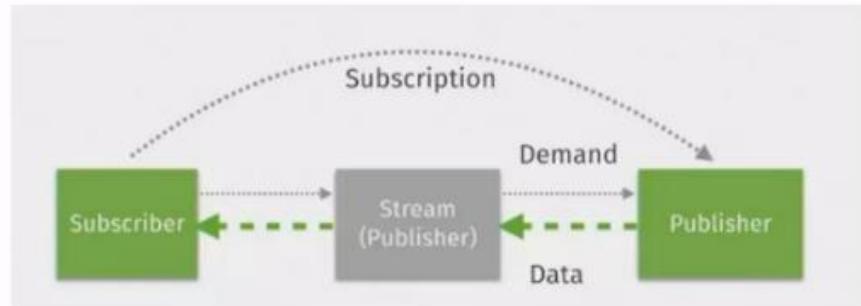
- un producteur lent et un consommateur rapide
- Un producteur rapide et un consommateur lent



SPÉCIFICATION STREAMS

API

```
public interface Publisher<T> {  
    void subscribe(Subscriber<? super T> s);  
}  
  
public interface Subscriber<T> {  
    void onSubscribe(Subscription s);  
    void onNext(T t);  
    void onError(Throwable t);  
    void onComplete();  
}  
  
public interface Subscription {  
    void request(long n);  
    void cancel();  
}  
  
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
}
```



Interfaces ajouté à Java 9
(sous `java.util.concurrent`)

:

- `Flow.Publisher`
- `Flow.Subscriber`
- `Flow.Subscription`
- `Flow.Processor`

SPÉCIFICATION STREAMS

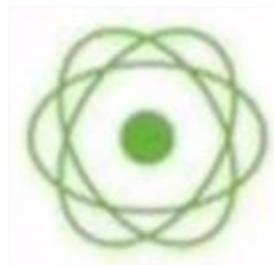
Librairies Reactive pour la JVM :



LIBRAIRIE REACTOR

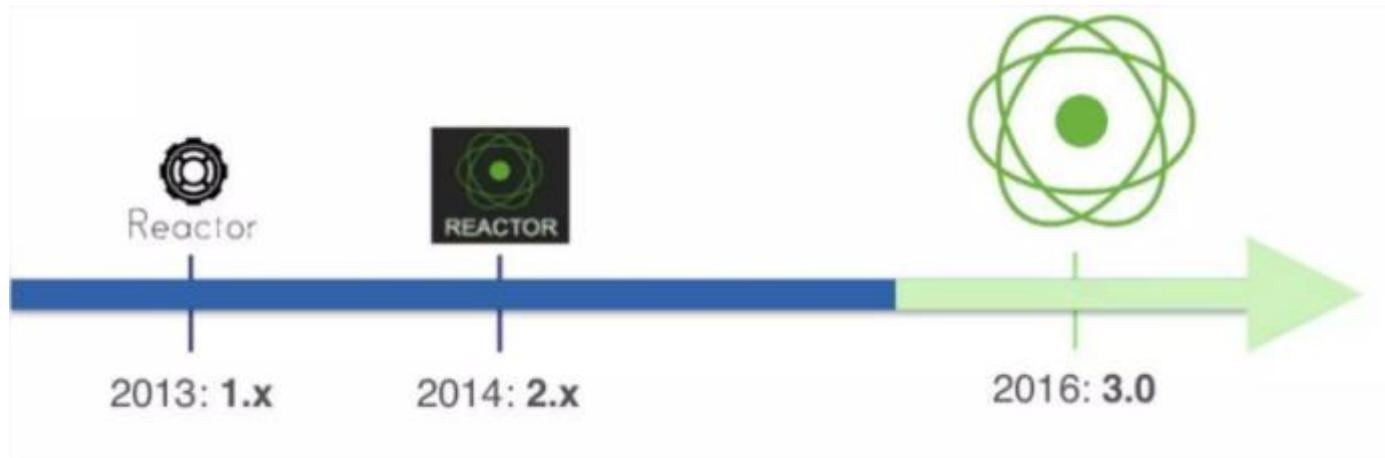
Projet soutenu et contribué par Spring pour la conception d'applications non bloquants pour la JVM (basé sur la spécification Reactive)

- Etend les streams Publisher avec les types Flux et Mono
- Ajoute le concept d'opérateur : chainage des opérateurs pour décrire les traitements à appliquer à chaque étape de données
- L'application d'un opérateur retourne un Publisher intermédiaire



LIBRAIRIE REACTOR

Evolution



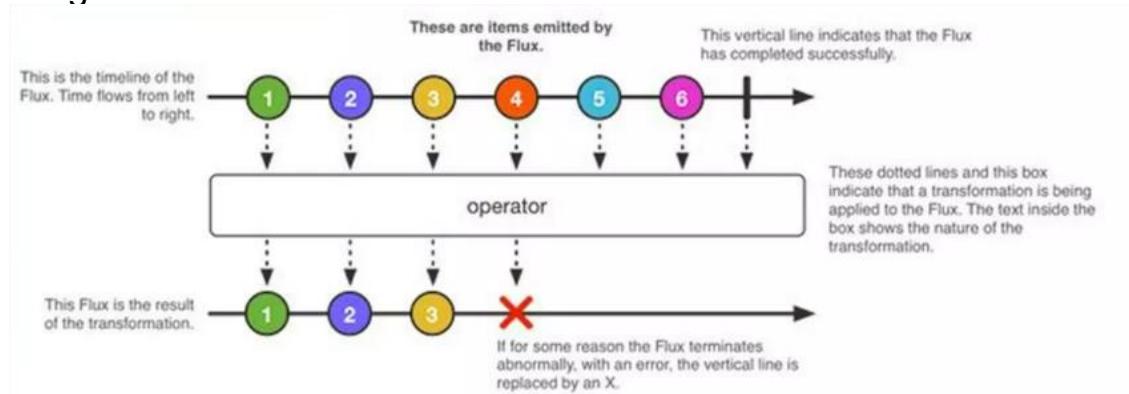
LIBRAIRIE REACTOR

Flux

Un objet de type Flux<T> est un Publisher<T> standard représentant une séquence asynchrone 0...N d'éléments, terminée soit par un signal de compléTION, soit par une erreur

Valeurs possibles d'un Flux : une valeur, un signal de compléTION, une erreur

Diagramme de Marble



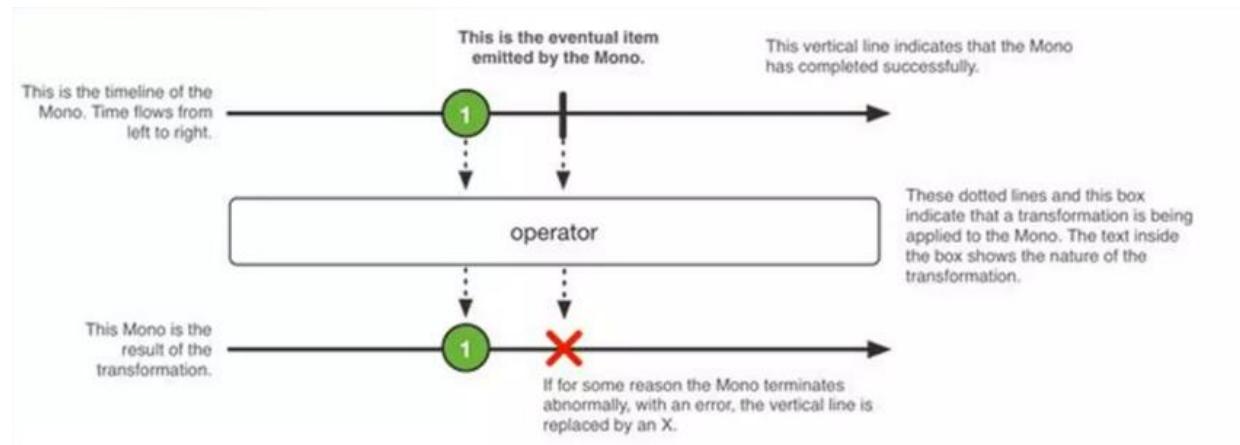
LIBRAIRIE REACTOR

Mono

Un objet de type Mono<T> est un Publisher<T> standard représentant au plus un élément, et terminé soit par un signal onComplete, soit par onError

Seul une sous partie des opérateurs des Flux est applicable sur les Monos

Diagramme de Marble



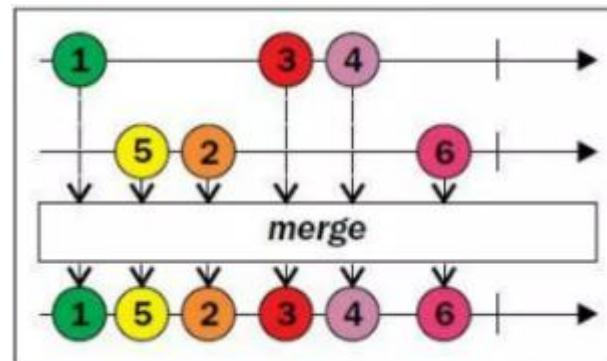
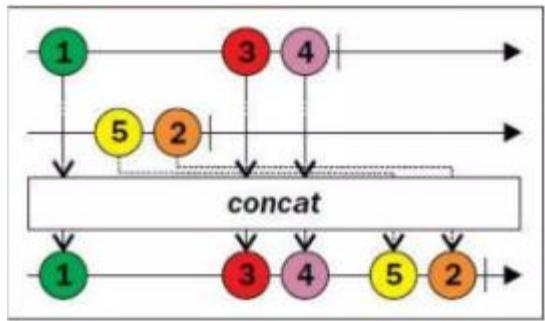
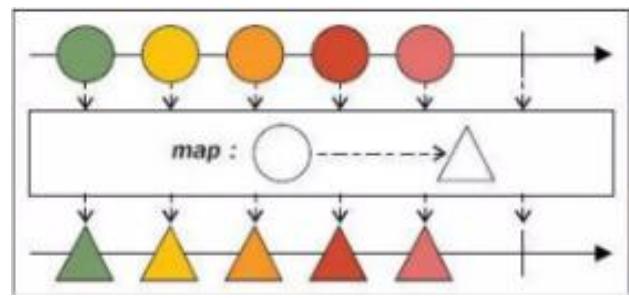
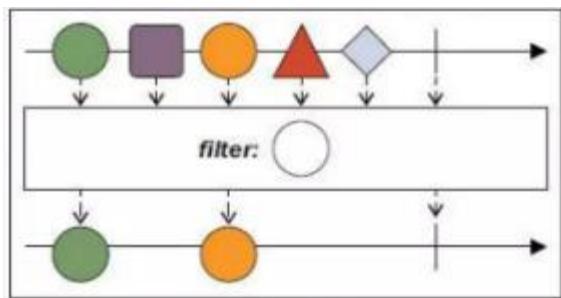
LIBRAIRIE REACTOR

Création de Flux ou Mono

- `Flux<String> seq1 = Flux.just("foo", "bar", "foobar");`
- `private static List<String> iterable = Arrays.asList ("foo", "bar", "foobar");
Flux<String> fluxFromIterable = Flux.fromIterable(iterable);`
- `Mono<String> noData = Mono.empty();`
- `Mono<String> data = Mono.just("foo");`
- `Flux<Integer> numbersFromFiveToSeven = Flux.range(5, 3); // 1st parameter is start of the range, 2nd is number of items`

LIBRAIRIE REACTOR

Quelques Opérateurs



LIBRAIRIE REACTOR

Testing

API **StepVerifier** : permet de tester le comportement d'un Flux ou d'un Mono après souscription

- expectNext()
- expectError()
- expectErrorMessage()
- expectComplete()
- Verify(), VerifyComplete(), VerifyError(), VerifyErrorMessage()

Example:

```
// empty Flux
Flux<String> emptyFlux = Flux.empty();
StepVerifier.create(emptyFlux).verifyComplete();
```

```
// non-empty Flux
Flux<String> nonEmptyFlux = Flux.just("John", "Mike", "Sarah");
StepVerifier.create(nonEmptyFlux).expectNext("John", "Mike", "Sarah").verify();
```

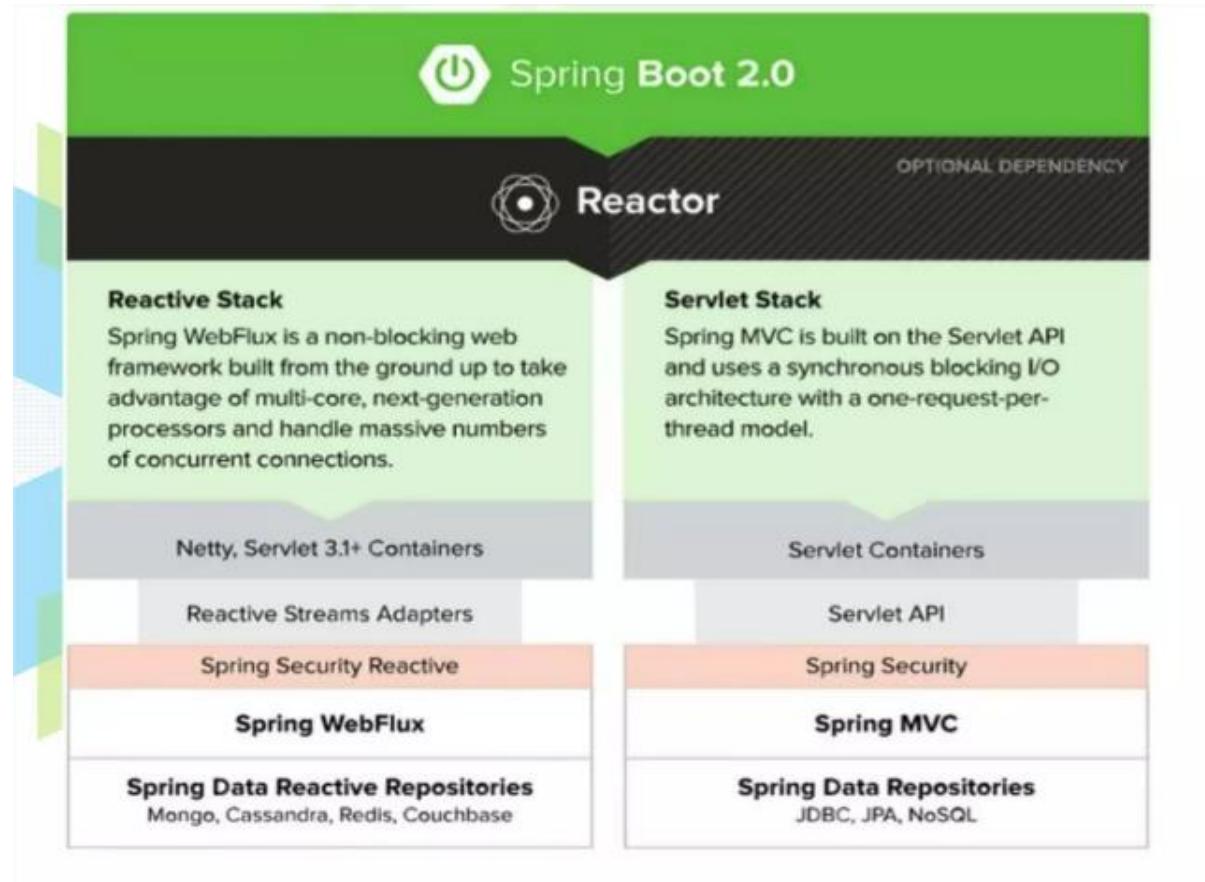
SPRING WEBFLUX

S'appuie sur la librairie Reactor pour proposer un module pour la programmation reactive entre serveur et client



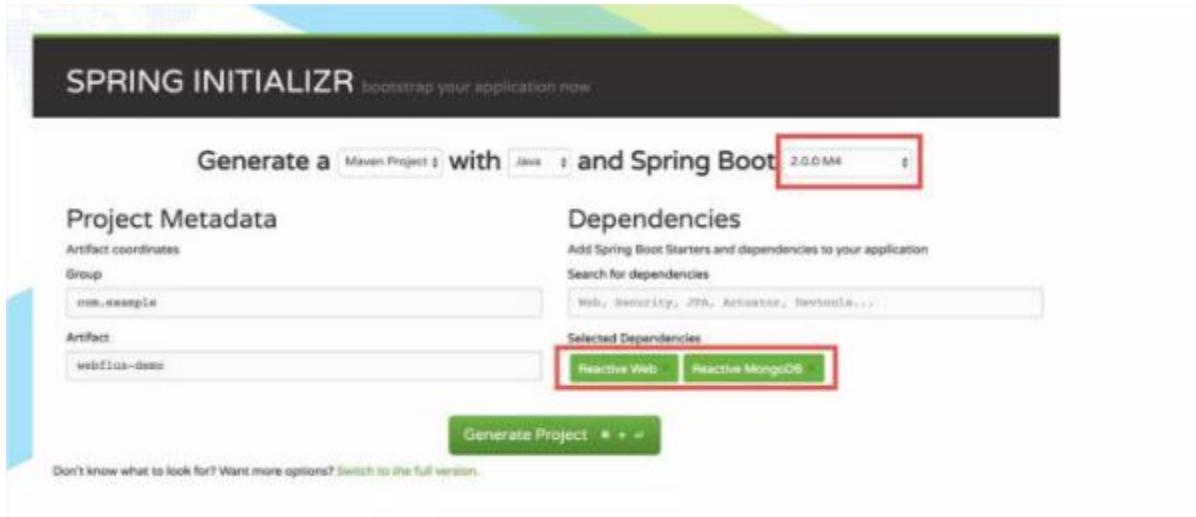
SPRING WEBFLUX

Spring WebFLux vs Spring MVC



SPRING WEBFLUX

Dépendances



SPRING WEBFLUX

Spring Webflux supporte d'un modèle de programmation côté serveur :

- Basé sur les annotations avec `@Controller` et les autres annotations de Spring MVC
- Basé sur la syntaxe fonctionnelle en utilisant les lambdas de Java 8

SPRING WEBFLUX

Exemple avec les annotations :

Example:

```
@RestController
@RequestMapping(value="/api/customer")
public class RestControllerAPIs {
    @GetMapping("/")
    public Flux<Customer> getAll() {
        ...
    }

    @GetMapping("/{id}")
    public Mono<Customer> getCustomer(@PathVariable Long id) {
        ...
    }
}
```

SPRING WEBFLUX

Exemple avec les annotations :

```
Map<Long, Customer> custStores = new HashMap<Long, Customer>();  
...  
@GetMapping  
public Flux<Customer> getAll() {  
    return Flux.fromIterable(custStores.entrySet().stream()  
        .map(entry -> entry.getValue())  
        .collect(Collectors.toList()));  
}
```

```
@GetMapping("/{id}")  
public Mono<Customer> getCustomer(@PathVariable Long id) {  
    return Mono.justOrEmpty(custStores.get(id));  
}
```

SPRING WEBFLUX

Exemple avec les annotations :

```
@PostMapping("/post")
public Mono<ResponseEntity<String>> postCustomer(@RequestBody Customer customer) {
    // do post
    custStores.put(customer.getCustId(), customer);
    return Mono.just(new ResponseEntity<>("Post Successfully!", HttpStatus.CREATED));
}
```

SPRING WEBFLUX

Avec la programmation fonctionnelle

Introduction de deux composants fondamentales:

- RouterFunction qui permet de router une requête HTTP(alternative à @RequestMapping)
- HandlerFunction qui permet de traiter une requête (alternative aux méthodes des classes @Controller)

RouterFunction.route(RequestPredicate, HandlerFunction) =>
exécute le HandlerFunction si la RequestPredicate est satisfaite

```
RouterFunction router = route( GET("/test"),
    request -> ok().build() );
```

SPRING WEBFLUX

Exemple avec la programmation fonctionnelle

```
RouterFunction router = route( GET("/test"),
    request -> ok().build() );
```

- **GET("/test")** is the *RequestPredicate*.
- **req -> ok().build()** is the handler producing the response.

SPRING WEBFLUX

Coté Client

Utilisation de WebClient : client réactive pour faire des requêtes HTTP avec les streams Reactive

WebClient est un équivalent de RestTemplate

WebClient est une interface avec une implémentation unique DefaultWebClient

Usage :

- Créer une instance
- Faire une requête
- Traiter la réponse

SPRING WEBFLUX

Coté Client – Crédit à WebClient avec 3 possibilités :

- With default settings:
 - `WebClient client1 = WebClient.create();`
- With a given base URI:
 - `WebClient client2 = WebClient.create("http://localhost:8080 (http://localhost:8080)");`
- Using the *DefaultWebClientBuilder* class:
 - `WebClient client3 = WebClient
.builder()
.baseUrl("http://localhost:8080 (http://localhost:8080)")
.defaultCookie("cookieKey", "cookieValue")
.defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
.defaultUriVariables(Collections.singletonMap("url", "http://localhost:8080 (http://localhost:8080)"))
.build();`

SPRING WEBFLUX

Coté Client – Exemple :

```
WebClient.RequestHeadersSpec requestSpec1
    = WebClient.create()
    .method(HttpMethod.POST)
    .uri("/resource")
    .body(BodyInapters.fromPublisher(Mono.just("data")), String.class);

WebClient.RequestHeadersSpec<?> requestSpec2
    = WebClient.create("http://localhost:8080 (http://localhost:8080)")
    .post()
    .uri(URI.create("/resource"))
    .body(BodyInapters.fromObject("data"));
```

SPRING WEBFLUX

Coté Client – Exemple réponse

Utilisation de :

- `exchange()` => fournit un objet de type `ClientResponse` avec le status, les headers ...
- ou `retrieve()` => fournit directement le body

Example:

```
String response2 = request1.exchange()
    .block()
    .bodyToMono(String.class)
    .block();
```

```
String response3 = request2.retrieve()
    .bodyToMono(String.class)
    .block();
```

SPRING WEBFLUX

Coté Client – WebTestClient

Client reactive pour gérer le testing

Même API que WebClient

Peut se connecter à n'importe quel serveur via HTTP

Fournit les méthodes pour vérifier la réponse

SPRING WEBFLUX

Choix d'un serveur :

Spring Webflux est supporté par les serveurs suivantes :

- Netty
- Undertow
- Jetty
- Tomcat
- Conteneurs de Servlet 3.1 +

Par défaut Spring boot utilise Netty avec Webflux

Spring Security

SPRING SECURITY

La sécurité d'une application repose sur deux piliers :

Authentification =>

Vérification de l'identité de la personne ou application qui se connecte sur le système

Autorisation =>

Vérification des droits d'accès ou privilèges aux différentes parties de l'application



SPRING SECURITY

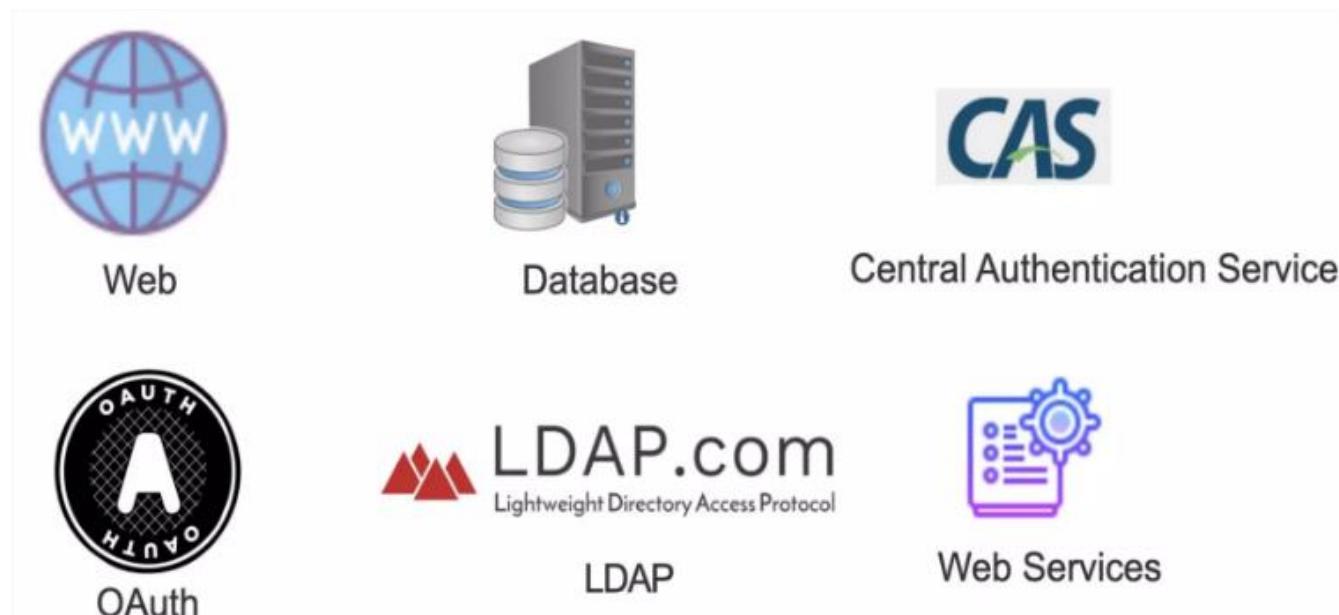
Spring Security est un module de Spring permettant de gérer l'authentification et le contrôle d'accès pour les applications Java

Spring Security est le standard de facto pour la sécurité des applications Spring

Peut être étendu pour satisfaire les exigences particulières

SPRING SECURITY

Intégration



SPRING SECURITY

Dépendances

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

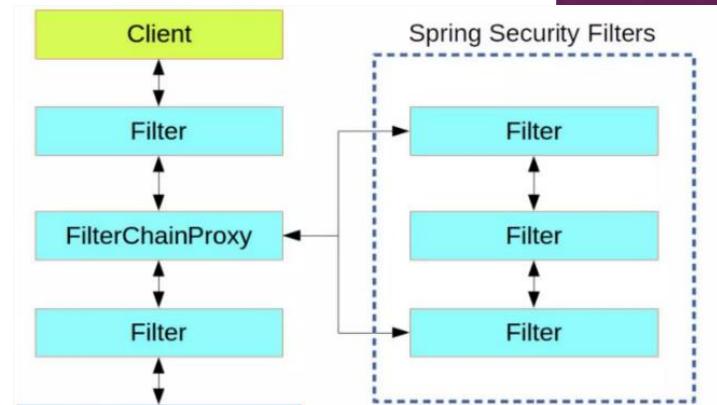
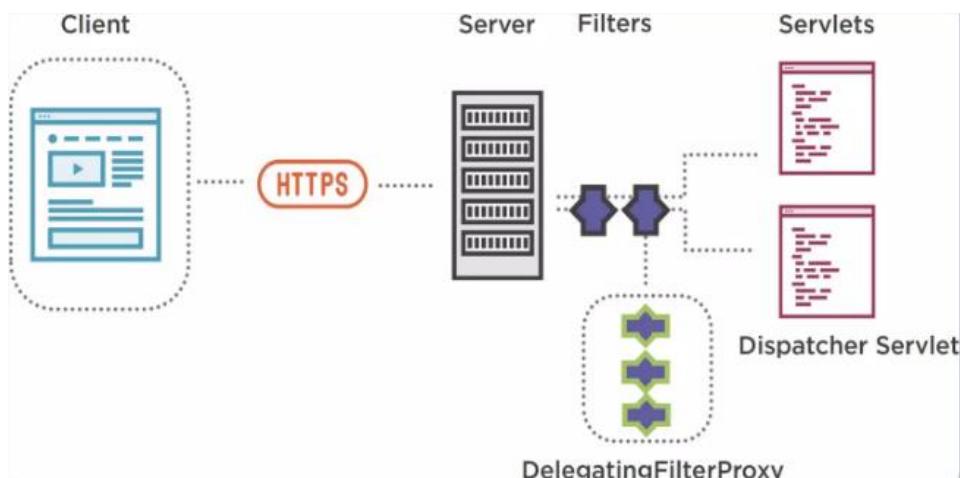
SPRING SECURITY

Fonctionnement

S'appuie sur le concept de filtre

Déclaration du composant Spring Security Filter, DelegatingFilterProxy dans le descripteur web

DelegatingFilterProxy va transférer toutes les requêtes aux autres filtres Spring Security pour faire l'authentification ou l'autorisation



SPRING SECURITY

Interface Filter

```
public interface Filter {  
  
    /** <p>Called by the web container ...*</p>  
     * @param filterConfig  
     * @throws ServletException  
    default public void init(FilterConfig filterConfig) throws ServletException {}  
  
    /** <p>The <code>doFilter</code> method of the Filter is called by the ...*</p>  
     * @param request  
     * @param response  
     * @param chain  
     * @throws IOException  
     * @throws ServletException  
    public void doFilter(ServletRequest request, ServletResponse response,  
                        FilterChain chain)  
        throws IOException, ServletException;  
  
    /** <p>Called by the web container ...*</p>  
     * @throws ServletException  
    default public void destroy() {}  
}
```

SPRING SECURITY

DelegatingFilterProxy délègue la requête à l'objet FilterChainProxy (FCP), qui à son tour la délègue à un objet de type SecurityFilterChain (SFC)

SFC est un wrapper sur une collection de filtres Spring qui contiennent la logique de la sécurité

Quand une requête arrive, FCP itère sur les filtres SFC pour trouver un qui matche avec l'url de la requête

Le filtre sélectionnée applique alors les règles

NB : L'ordre de SFC à une importance cruciale

SPRING SECURITY

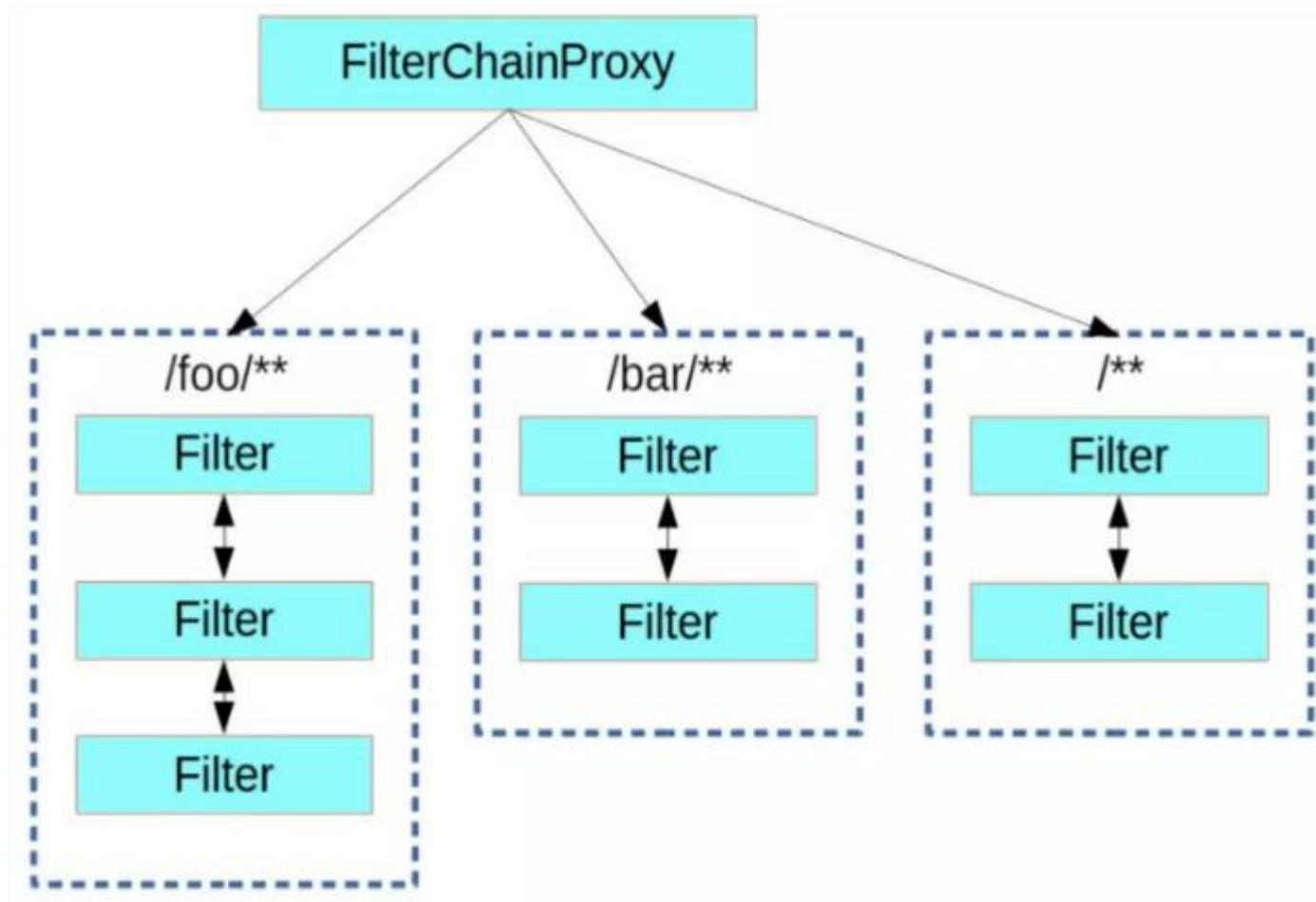
Interface SecurityFilterChain

```
public interface SecurityFilterChain {  
  
    boolean matches(HttpServletRequest request);  
  
    List<Filter> getFilters();  
  
}
```

matches() : retourne un booléen pour vérifier si la requête matches le filterchain

getFilters() : retourne une liste de filtres pour la requête

SPRING SECURITY



SPRING SECURITY

Concepts

Authentication

Authorization

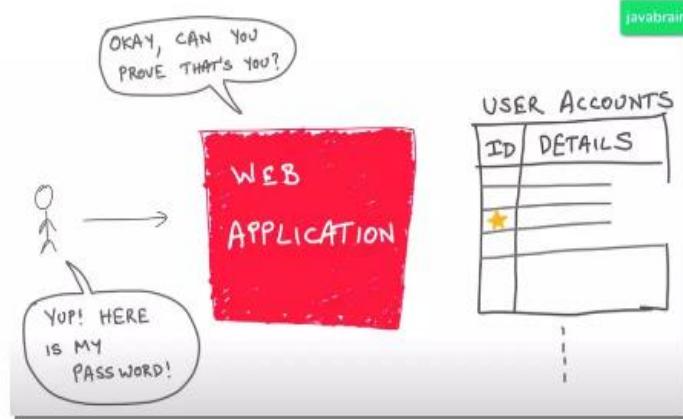
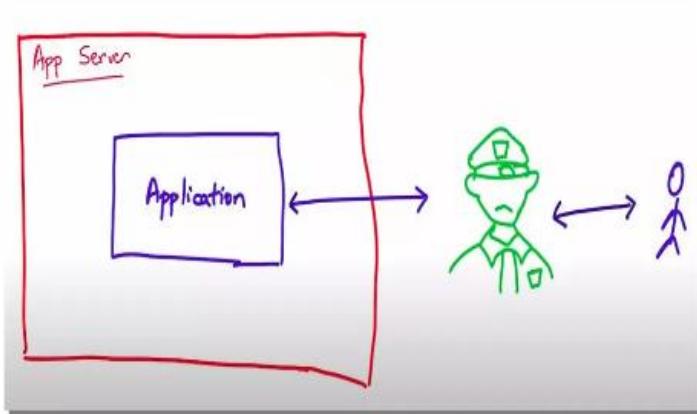
Principal

Granted Authority

Roles

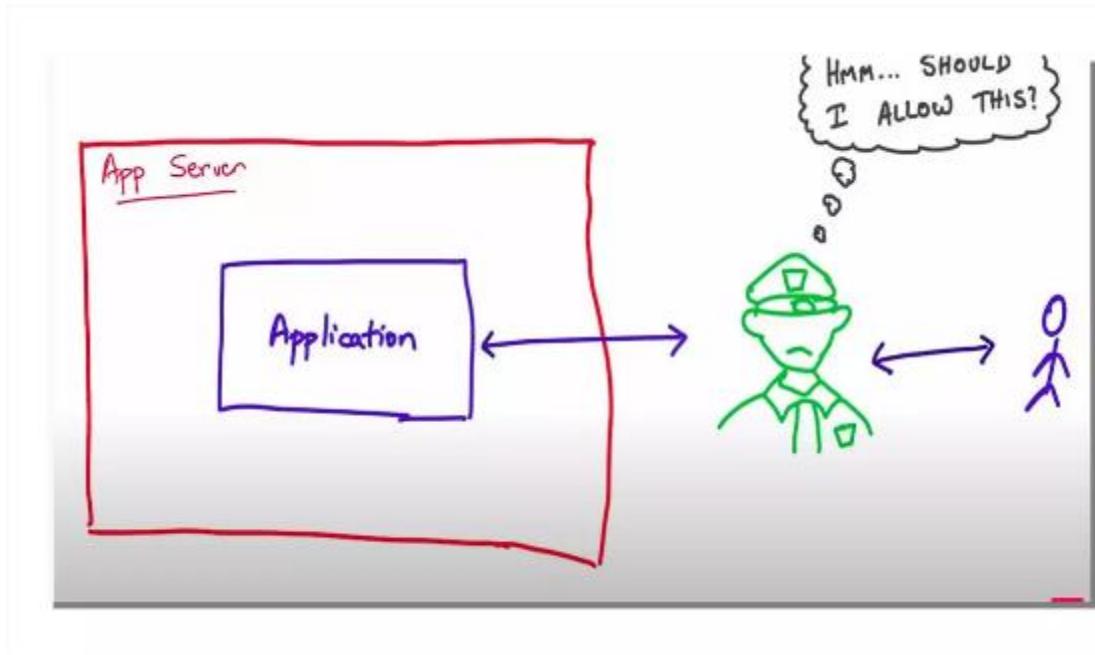
SPRING SECURITY

Authentification



SPRING SECURITY

Autorisation

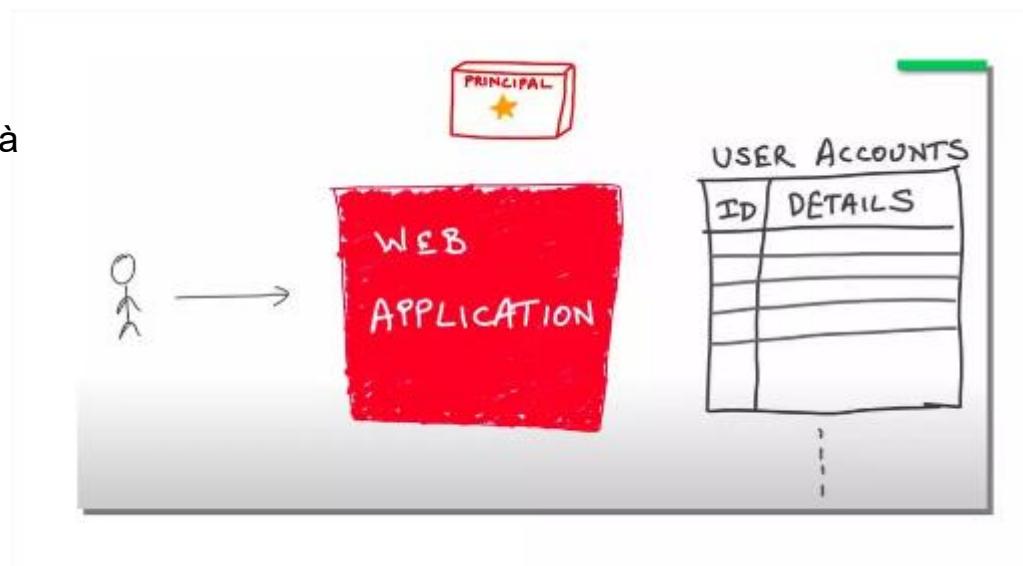


SPRING SECURITY

Principal

Un utilisateur n'a pas à entrer à chaque fois ses login/pwd à chaque nouvel requête ou changement de page

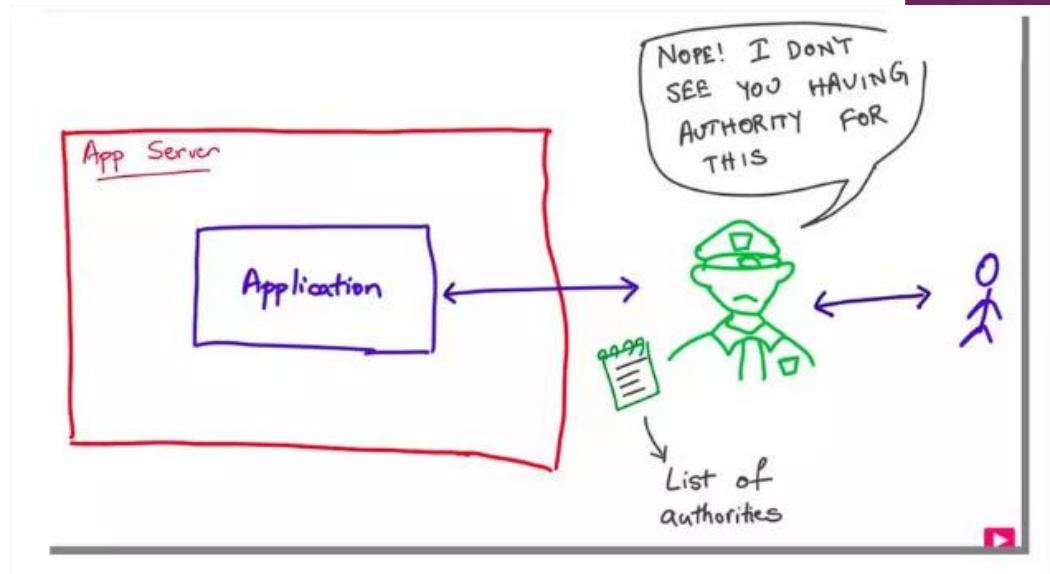
Le principal est un objet utilisé pour gérer cette fonctionnalité



SPRING SECURITY

Granted Authority (Privilège)

Permet de s'assurer que l'utilisateur a les privilèges nécessaires pour effectuer des actions

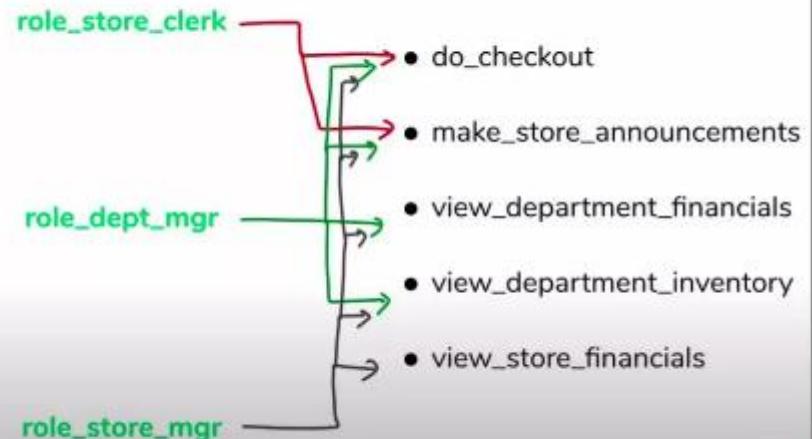


SPRING SECURITY

Rôles

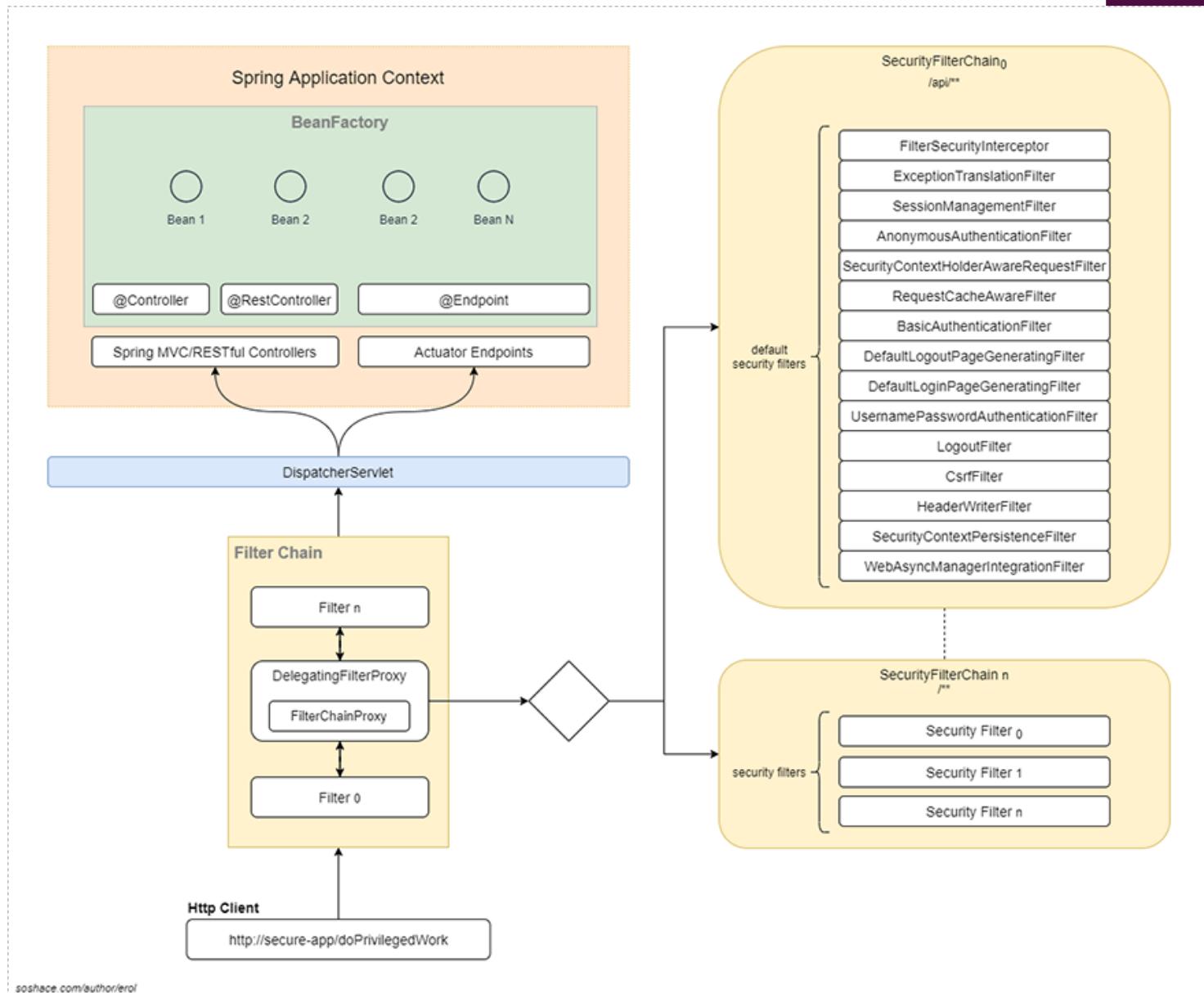
Groupe de privilèges

NB : Selon les projets, privilèges et rôles peuvent être interchangeables; Le niveau de granularité dépendra de chaque projet



SPRING SECURITY

OVERVIEW



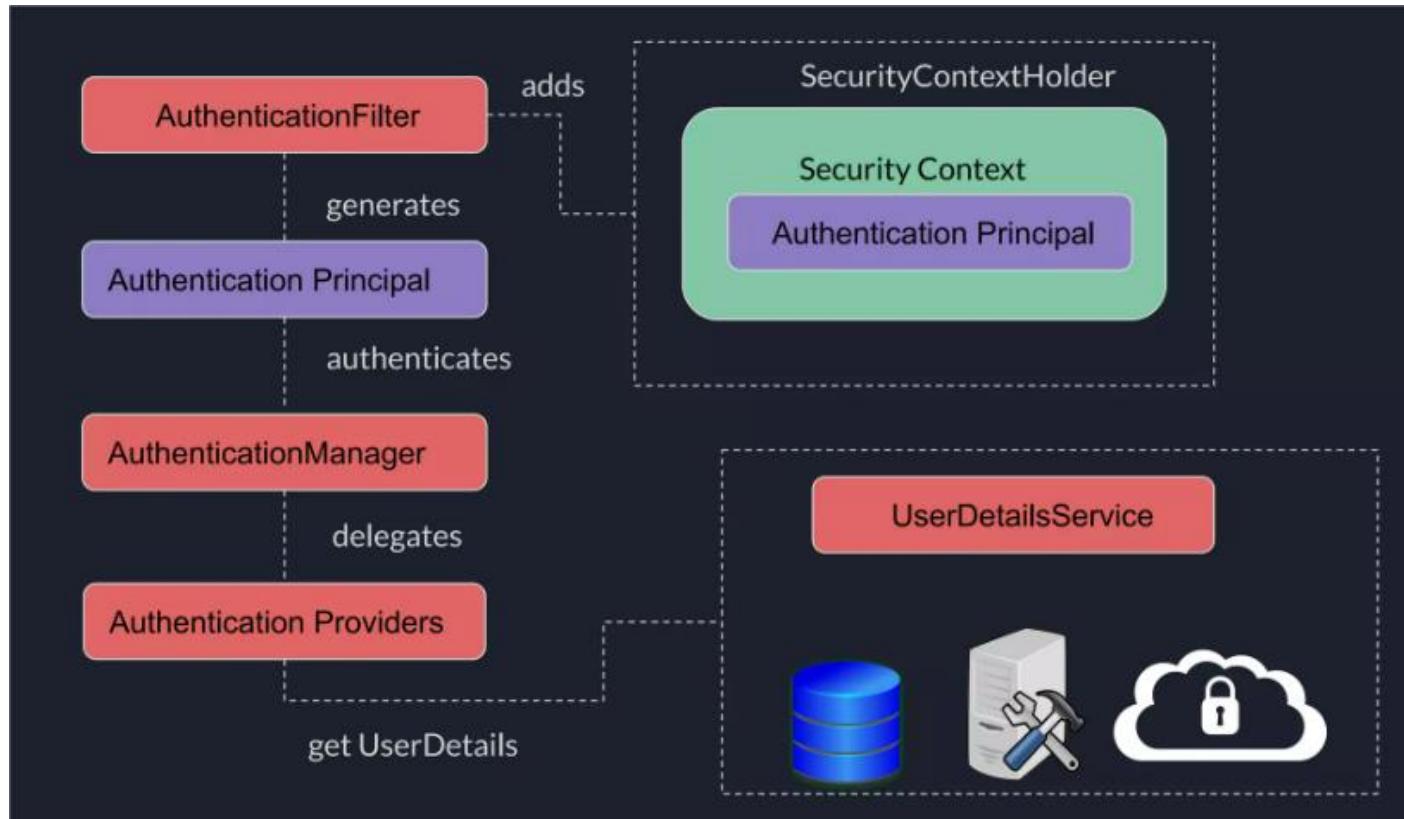
SPRING SECURITY

Flux d'authentification classique

1. Le filtre d'authentification crée une requête d'authentification et le passe à l'Authentification Manager
2. L'Authentification Manager délègue la requête à l'Authentification Provider
3. L'Authentification Provider utilise UserDetailsService pour charger les informations dans un objet de type UserDetails et retourne un Authenticated Principal
4. Le filtre d'Authentification met ces informations dans l'objet SecurityContext

SPRING SECURITY

Flux d'authentification classique



SPRING SECURITY

Flux d'autorisation

1. FilterSecurityInterceptor obtient un « Security Metadata » en matchant la requête courante
2. FilterSecurityInterceptor récupère l'objet Authentication actuelle
3. L'objet Authentication, le Security Metadata et la requête sont passé à AccessDecisionManager
4. AccessDecisionManager délègue AccesDecisionVoter(s) pour la décision finale

SPRING SECURITY

Ordre des filtres natives

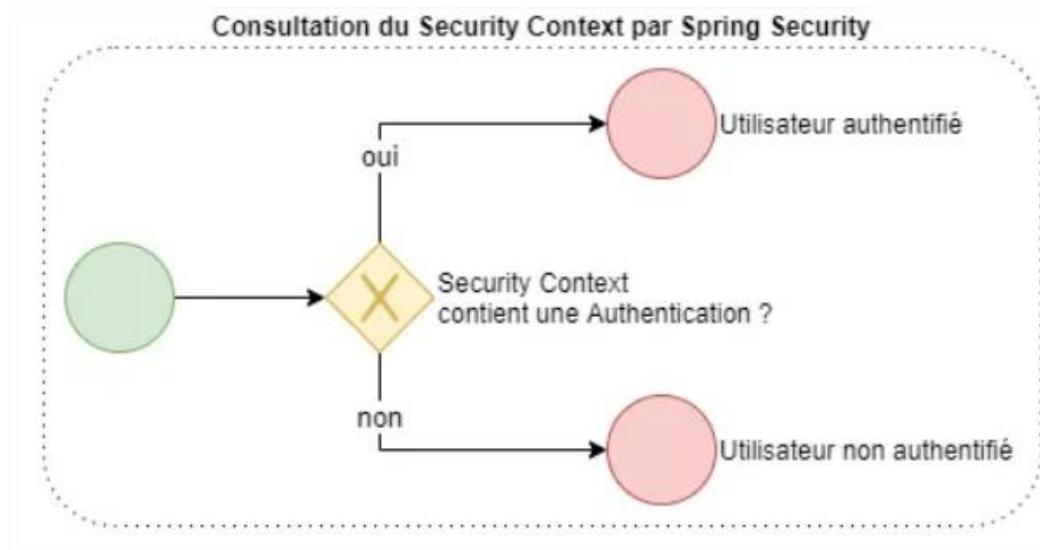
- | | |
|---|--|
| 1. ChannelProcessingFilter | 10. X509AuthenticationFilter |
| 2. WebAsyncManagerIntegrationFilter | 11. AbstractPreAuthenticatedProcessingFilter |
| 3. SecurityContextPersistenceFilter | 12. CasAuthenticationFilter |
| 4. HeaderWriterFilter | 13. OAuth2LoginAuthenticationFilter |
| 5. CorsFilter | 14. Saml2WebSsoAuthenticationFilter |
| 6. CsrfFilter | 15. UsernamePasswordAuthenticationFilter |
| 7. LogoutFilter | 16. OpenIDAuthenticationFilter |
| 8. OAuth2AuthorizationRequestRedirectFilter | 17. DefaultLoginPageGeneratingFilter |
| 9. Saml2WebSsoAuthenticationRequestFilter | |

- | | |
|---|--|
| 18. DefaultLogoutPageGeneratingFilter | 26. RememberMeAuthenticationFilter |
| 19. ConcurrentSessionFilter | 27. AnonymousAuthenticationFilter |
| 20. DigestAuthenticationFilter | 28. OAuth2AuthorizationCodeGrantFilter |
| 21. BearerTokenAuthenticationFilter | 29. SessionManagementFilter |
| 22. BasicAuthenticationFilter | 30. ExceptionTranslationFilter |
| 23. RequestCacheAwareFilter | 31. FilterSecurityInterceptor |
| 24. SecurityContextHolderAwareRequestFilter | 32. SwitchUserFilter |
| 25. JaasApiIntegrationFilter | |

SPRING SECURITY

Sessions avec Spring security

Spring Security regarde si son Security Context contient un objet de type Authentication pour vérifier si un utilisateur est connecté



SPRING SECURITY

Sessions avec Spring security

Si l'utilisateur n'est pas authentifié, il est redirigé vers la page de login

Si le Security Context possède un objet Authentication, l'utilisateur peut accéder aux endpoints sécurisés, selon ses droits. Jusqu'à ce qu'il soit déconnecté ... dans ce cas, le Security Context est nettoyé

Spring Security permet la configuration des sessions :

```
55      @Override
56  ⚡ @
57      protected void configure(HttpSecurity http) throws Exception{
58          http
59              .cors().and()
60              .csrf()
61                  .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
62                  .ignoringAntMatchers("/api/auth/**")
63              .and()
64              .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
65              .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and()
66              .authorizeRequests().antMatchers( ...antPatterns: "/api/auth/**").permitAll() ExpressionUrlA
```



SPRING SECURITY

Quatre configurations de session sont possibles, :

SessionCreationPolicy.ALWAYS : application *stateful*, état sauvegardé à chaque requête. Le contenu de **Security Context** est conservé par Spring Security, et peut être réutilisé pour plusieurs requêtes sans que l'utilisateur doive s'authentifier à nouveau.

SessionCreationPolicy.IF_REQUIRED : utilisé par défaut par Spring Security. Le **Security Context** n'est sauvegardée que lorsque c'est nécessaire..

SessionCreationPolicy.NEVER : aucune sauvegarde de session, mais utilisation possible des sessions déjà existantes

SessionCreationPolicy.STATELESS : Spring Security ne sauvegarde aucune session, et ne consulte aucune session existante. Nettoyage du **Security Context** à chaque requête; chaque nouvelle requête doit donc être authentifié

SPRING SECURITY

Dans le cadre d'un gestion de session stateless, comment éviter à l'utilisateur de s'authentifier à chaque nouvelle requête ?

=> Utilisation entre autres de JWT



SPRING SECURITY AVEC JWT

Spring Security with JWT

JWT est une structure de données en json contenant des informations sur un token d'authentification : émetteur, le sujet, la date d'expiration, les droits ...

JWT est composé de trois parties distinctes (header, payload, signature) qui sont encodées en base64URL et séparées par des points (.)

Spring Security s'intègre facilement avec JWT

SPRING SECURITY AVEC JWT

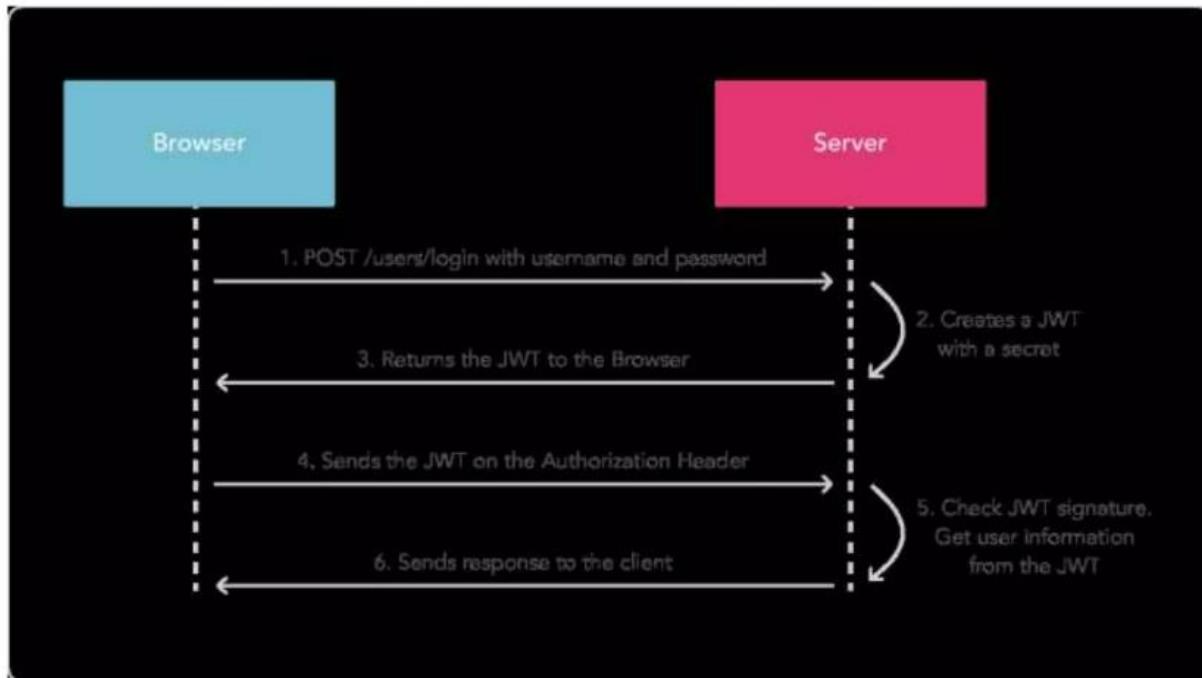
JWT

Header : première partie d'un JWT, contient des informations sur le type de jeton et l'algorithme de signature utilisé : "alg" : l'algorithme de signature utilisé pour signer le jeton "typ"

Payload : contient les revendications appelées "Claims" qui représentent les informations spécifiques à l'utilisateur et aux métadonnées supplémentaires.

Signature : la signature est utilisée pour garantir que le jeton n'a pas été modifié par des tiers non autorisés. La signature est calculée en utilisant le Header + les Claims encodées en Base64Url + une clé secrète (dans le cas d'un algorithme de signature symétrique) ou une paire de clés (dans le cas d'un algorithme de signature asymétrique).

SPRING SECURITY AVEC JWT



POST http://localhost:8080/api/login

Authorization	Headers	Body	Pre-request Script	Tests
Form-data	Raw	JSON	Text	File
{"username": "jdefrance", "password": "jdefrance"}				

Step 1: Request /api/login and get the JWT

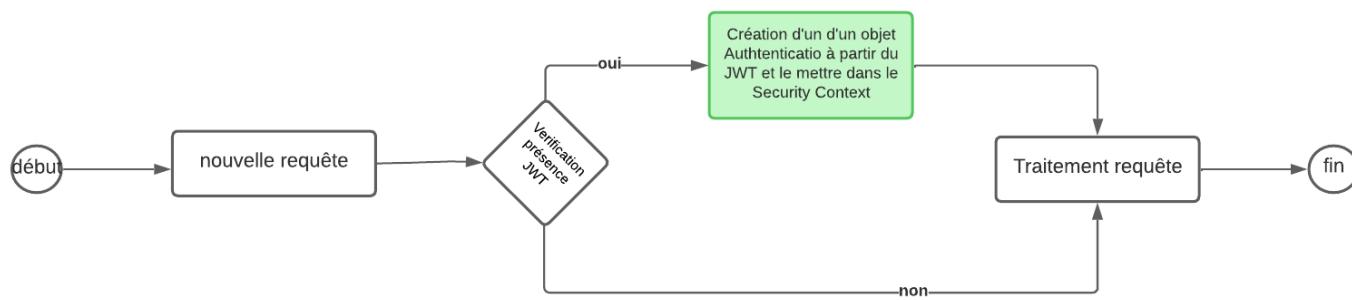
GET http://localhost:8080/api/thanks

Authorization	Headers (1)	Body	Pre-request Script	Tests
	Key	Value		
<input checked="" type="checkbox"/> Authorization	eyJhbGciOiJFUzLURMjI9eyJzdWIiOiJ			

Step 2: Request API REST with JWT (token)

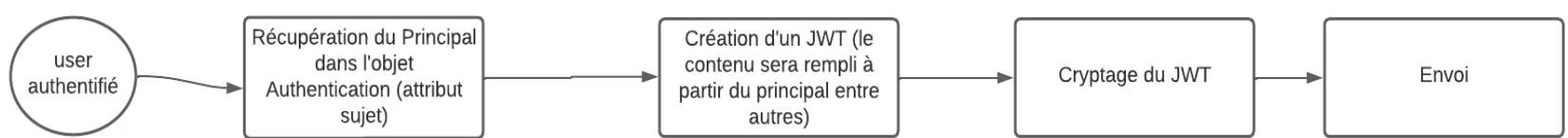
SPRING SECURITY

Utilisation JWT avec le Security Context



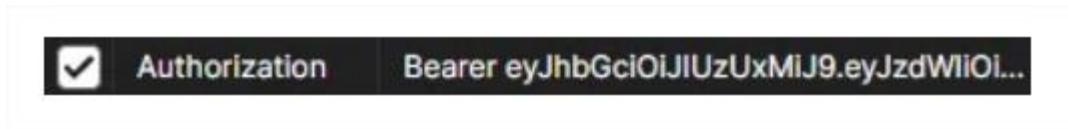
SPRING SECURITY

Création JWT à partir d'une Authentication présent dans le Security Context



SPRING SECURITY

A chaque nouvelle requête, le JWT est envoyé :



Spring security vérifie plusieurs informations sur le JWT reçu :

Date d'expiration

Format

Contenu ...

La valorisation de la Security Session avec un JWT passe l'utilisation du filtre approprié

Spring WebSockets

WEBSOCKETS

Le protocole HTTP est synchrone par défaut

Il peut arriver que le client puisse vouloir recevoir des données de manière asynchrone du serveur

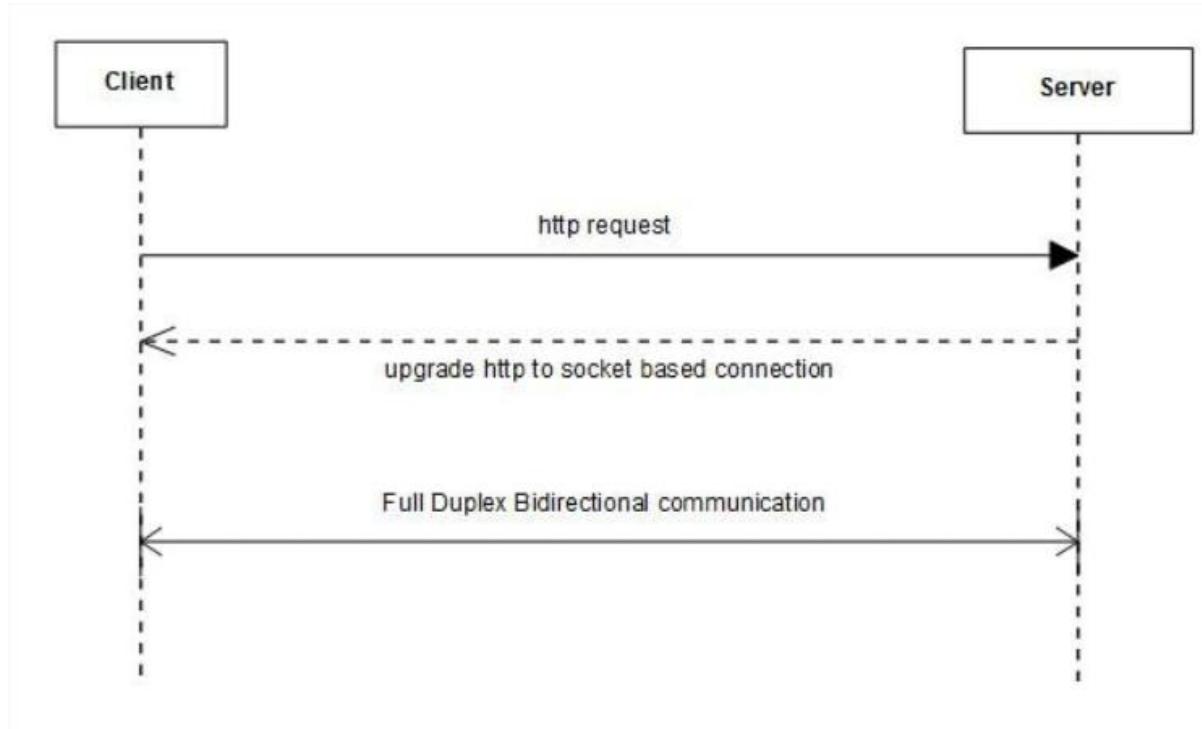


WEBSOCKETS

- Les requêtes HTTP sont **unidirectionnelles**, le client initie toujours la requête.
- **Half Duplex** - L'utilisateur demande une ressource et le serveur la sert ensuite au client. La réponse n'est envoyée qu'après la requête. Ainsi, à la fois, une seule demande se produit
- **Connexions TCP multiples** - Pour chaque demande, une nouvelle session TCP doit être établie puis fermée après réception de la réponse
- **Lourd** - La requête et la réponse HTTP nécessitent un échange de données supplémentaires entre le client et le serveur.

WEBSOCKETS

WebSocket est un protocole de communication, fournissant des canaux de communication en duplex intégral sur une seule connexion TCP.



WEBSOCKETS

- **Bidirectionnel** - En utilisant WebSocket, le client ou le serveur peut initier l'envoi d'un message.
- WebSocket est en **duplex intégral** - La communication entre client et serveur est indépendante l'une de l'autre.
- **Connexion TCP unique** - La connexion initiale utilise HTTP, puis cette connexion est mise à niveau vers une connexion basée sur un socket. Cette connexion unique est ensuite utilisée pour toutes les communications futures
- **Léger** - L'échange de données de messages WebSocket est beaucoup plus léger par rapport à http.

WEBSOCKETS - SOCKJS

Websocket est pris en charge par la plupart des navigateurs modernes (HTML 5) et les serveurs les plus populaires

SockJS est une librairie Javascript qui permet de faire des échanges de type websocket

Certains navigateurs mobiles ne supportent pas les websockets;
Certains équipements réseaux (proxies, routeurs, pare-feux...) peuvent aussi couper les connexions TCP trop longs ou empêcher l'upgrade de HTTP vers Websocket lors de la phase de handshake.
SockJS permet de résoudre ces problématiques en émulant Websocket.

SockJS essaie d'abord d'utiliser les WebSockets natifs. Si ça échoue, il peut utiliser des protocoles de transport spécifiques au navigateur pour simuler WebSocket.

WEBSOCKETS - STOMP

Contrairement à HTTP, Websocket ne spécifie pas de format de message; le client et le serveur peuvent convenir d'un sous-protocole lors du handshake.

Ce sous-protocole définirait la façon les messages envoyés sont formatés et reçus.

STOMP est un protocole utilisé pour cela

WEBSOCKETS - STOMP

STOMP (simple text orientated messaging protocol) est un sous-protocole similaire à HTTP. Chaque fois que l'une des parties envoie des données, elle doit les envoyer sous la forme d'une **Frame**. Une **Frame** a une structure comme avec une requête HTTP et s'exécute au dessus de TCP.

Chaque Frame a un verbe qui lui est associé selon l'intention ((ex. CONNECT, DISCONNECT, SEND ...)); Il contient également un en-tête pour donner des informations supplémentaires et un corps pour le contenu principal.

Spring gestion du cache

SPRING CACHE

Le cache permet d'améliorer les performances d'une application

C'est une mémoire entre l'application et la base de données

Le cache est utilisé pour garder les données les plus accédées afin de réduire le nombre d'appels vers la bases de donnée

Types de cache

- Cache en mémoire
- Cache de bases de données
- Cache de serveur web
- Cache de CDN

SPRING CACHE

La JSR 107 correspond à la spécification de l'API de caching en Java

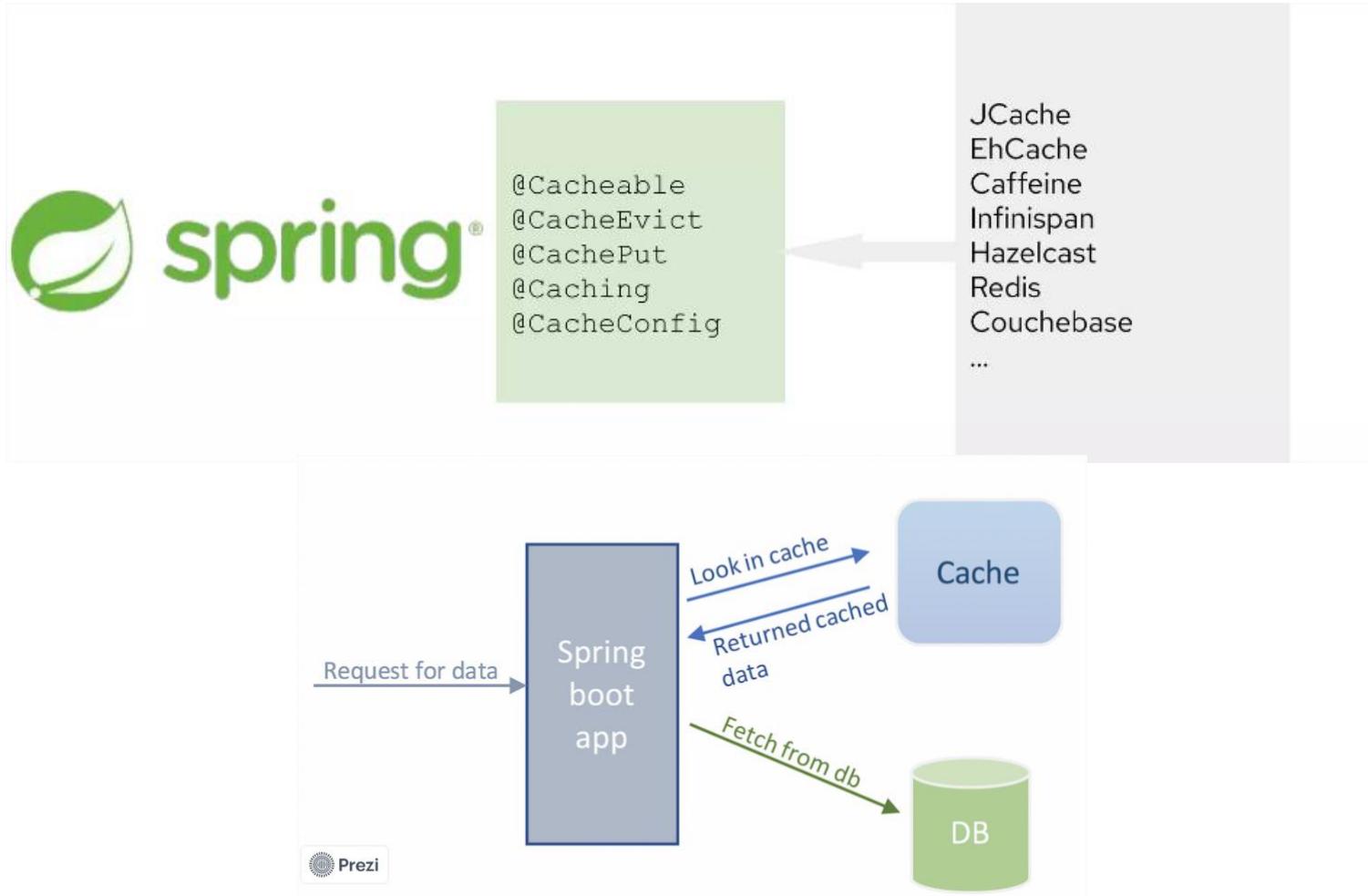
L'implémentation officielle de cette JSR est Jcache

Les solutions de caching actuelle implémentent tous cette API : EhCache, Hazelcast, Guava Cache, Caffeine, Gemfire cache,...

Ces implémentations fournissent un moyen de créer, d'accéder, de mettre à jour, et de supprimer les entrées d'un cache

SPRING CACHE

Providers de cache et annotations pour l'abstraction



SPRING CACHE

Annotation **@EnableCaching** permet d'activer le cache sur une classe avec Spring

Si Spring boot est utilisé, l'annotation **@EnableCaching** peut être mis au niveau de l'application main.

Les résultats des méthodes annotés **@Cacheable** seront automatiquement mis en cache avec comme clé de cache le paramètre de la méthode (si plusieurs paramètres, un hash sera utilisé)

La clé de cache peut aussi déterminé par un algorithme custom en implémentant KeyGenerator; ou en précisant avec le langage SpeL la clé à utiliser

SPRING CACHE

Annotations

@Cacheable peut définir les conditions pour lequel le retour de la méthode sera mis en cache.

@CachePut permet de mettre à jour la valeur d'une donnée en cache. Le contenu de la méthode annotée **@CachePut** sera systématiquement exécuté.

@CacheEvict permet de supprimer un ou tous les éléments présents dans le cache. Il est possible d'indiquer si la suppression est à effectuer avant ou après l'exécution de la méthode

@Caching permet de combiner les annotations **@Cacheable**, **@CachePut** et **@CacheEvict** sur une seule méthode. Ceci permet d'appliquer des règles différentes en fonctions du cache utilisés, si plusieurs caches sont utilisées pour stocker les mêmes données.

@CacheConfig permet de définir une configuration commune à toutes les méthodes d'une classe. Typiquement, cela peut être le nom du générateur de clé, le nom du cache à utiliser.

SPRING CACHE

Etapes de configuration

1. Ajouter les dépendances pour la gestion du cache
2. Activer les annotation Spring
3. Choisir l'implémentation (Spring fournit deux implémentations : une à base des ConcurrentHashMap de Java et la seconde avec ehcache.
4. Ajouter les annotations pour la gestion du cache

Schedulers avec Spring

SCHEDULERS

Spring propose un support permettant de planifier des tâches régulières

Pour activer le scheduling avec Spring

Configuration Java
`@EnableScheduling`

XML
`<task:annotation-driven>`

Privilégier la méthode par annotation car plus simple

SCEDULERS

L'annotation `@Scheduler` permet d'exécuter une méthode de manière régulière

`@Scheduler` a des arguments permettant de définir le comportement de la planification

`fixedDelay` : Délai fixe entre la fin de la dernière exécution et le début de la nouvelle

`fixedRate` : intervalle de temps fixe entre les exécutions, peut importe si la tâche précédente est finie ou pas. Par défaut, les tâches ne sont joués en parallèle. Pour jouer les tâches en parallèle, annoter la classe avec `@EnableAsync` et la méthode avec `@Async`

`initialDelay` : determine le délai avant la première exécution

SCEDULERS

```
package com.formation.schedule;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component
public class Planificateur{

    @Scheduled(fixedDelay = 3000)
    public void tacheADelaiFixe() {
        System.out.println("Hello scheduler");
    }
}
```

SCHEDULERS

Spring permet d'utiliser les expressions de type cron pour définir le timing

```
@Scheduled(cron = "0 5 10 30 * ?")
public void planificationAvecCron() {
    long now = System.currentTimeMillis() / 1000;
    System.out.println(" planification de taches avec cron ");
}
```

Par défaut c'est le fuseau horaire du serveur qui est utilisé; On peut changer ce fuseau :

```
@Scheduled(cron = "0 5 10 30 * ?", zone = "Europe/Paris")
```

Supervision avec Spring Actuator

SPRING ACTUATOR

Spring Actuator apporte des fonctionnalités de monitoring aux applications Spring boot

Actuator permet entre autres de :

- ❖ Monitorer l'application
- ❖ Afficher les métriques
- ❖ Afficher l'état des services, des beans
- ❖ Afficher les variables d'environnement ...

SPRING ACTUATOR

Spring Actuator est une simple dépendance maven à déclarer

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

Les fonctionnalités par défaut sont alors disponibles et permettent un débogage facilité

SPRING ACTUATOR

Spring Actuator expose des endpoints pour visualiser l'état de certains éléments

Un endpoint de discovery permet de voir les endpoints activés

<http://localhost:8080/actuator>

```
// 20230820213624
// http://localhost:8080/actuator

{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8080/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8080/actuator/caches",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8080/actuator/health/{*path}",
      "templated": true
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    }
  }
}
```

SPRING ACTUATOR

Pour activer tous les endpoints disponibles, il faut ajouter dans le fichier application.properties :

```
management.endpoints.web.exposure.include=*
```

On peut aussi n'activer qu'un sous ensemble de endpoints

```
management.endpoints.web.exposure.include=health,caches,  
heapdump
```

SPRING ACTUATOR

Spring Actuator permet d'enrichir les métriques

Le endpoint `/health` permet de voir l'état des services

Il analyse le retour de toutes les classes héritant de **HealthIndicator**. Chaque renvoyant si les services sont up ou pas

On peut ajouter des règles métiers qui déterminent si un service est up ou pas en implémentant HealthIndicator

Fin