



# Spring Boot

---

David THIBAU – 2020

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

- **Introduction**

- Rappels Spring Core
- L'offre Spring Boot
- Pré-requis, installation, IDE

- **Développer avec SpringBoot**

- Gestion des dépendances
- Structure projet
- Exécution, *DevTools* et Debug
- Propriétés de configuration
- Profils
- Configuration des traces

- **Persistance**

- Principes de SpringData
- Configuration base SQL
- Spring Boot et JPA
- NoSQL. L'exemple MongoDB
- Spring Data Rest

- **Applications Web**

- Rappels Spring MVC
- Spring MVC et les APIs Rest
- Spring Boot et APIs Rest

- **Spring Security**

- Rappels Spring Security
- Apports de Spring Boot
- Modèle stateful/stateless
- OAuth2

- **Spring Boot et les tests**

- Rappels Spring Test
- Apports de Spring Boot
- Tests auto-configurés

- **Déploiement Spring Boot**

- Actuator
- Déploiement d'applications Spring Boot
- Spring Boot et Docker

- **Annexes**

- Détails Spring Data Rest
- Spring MVC, HTML backend
- Spring WebFlux
- Mettre en place une auto-configuration



# Rappels Spring



# Historique

---

- ❖ *Spring* est un projet **OpenSource** avec un modèle économique basé sur le service (Support, Conseil, Formation, Partenariat et certifications)
- ❖ La société **SpringSource** fondée par les créateurs de Spring (Rod Johnson et Juergen Hoeller) a été rachetée par **VmWare**, puis intégrée dans la joint-venture **Pivotal Software**
- ❖ Succès de la solution : Perçu comme une alternative aux serveurs Java EE



# Projets Spring

---

*Spring* est en fait un ensemble de projets adaptés à toutes les problématiques actuelles.

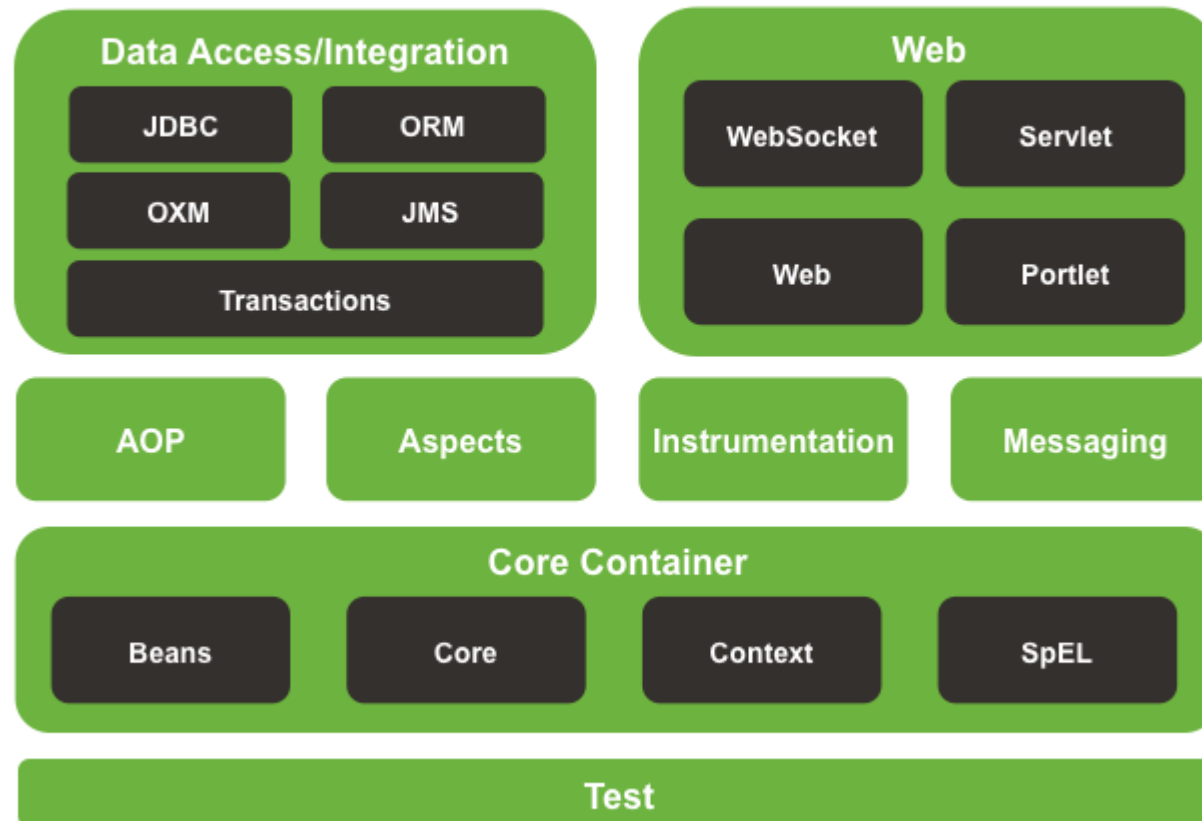
Tous ces projets ont comme objectifs :

- ✓ Permettre d'écrire du code propre, modulaire et testable
- ✓ Éviter d'avoir à coder les aspects techniques (plomberie)
- ✓ Être portable : déployer une application stand-alone, sur un serveur en Paas du moment qu'il y ait une JVM
- ✓ Favoriser le *stateless* : Client/serveur sans conservation d'états

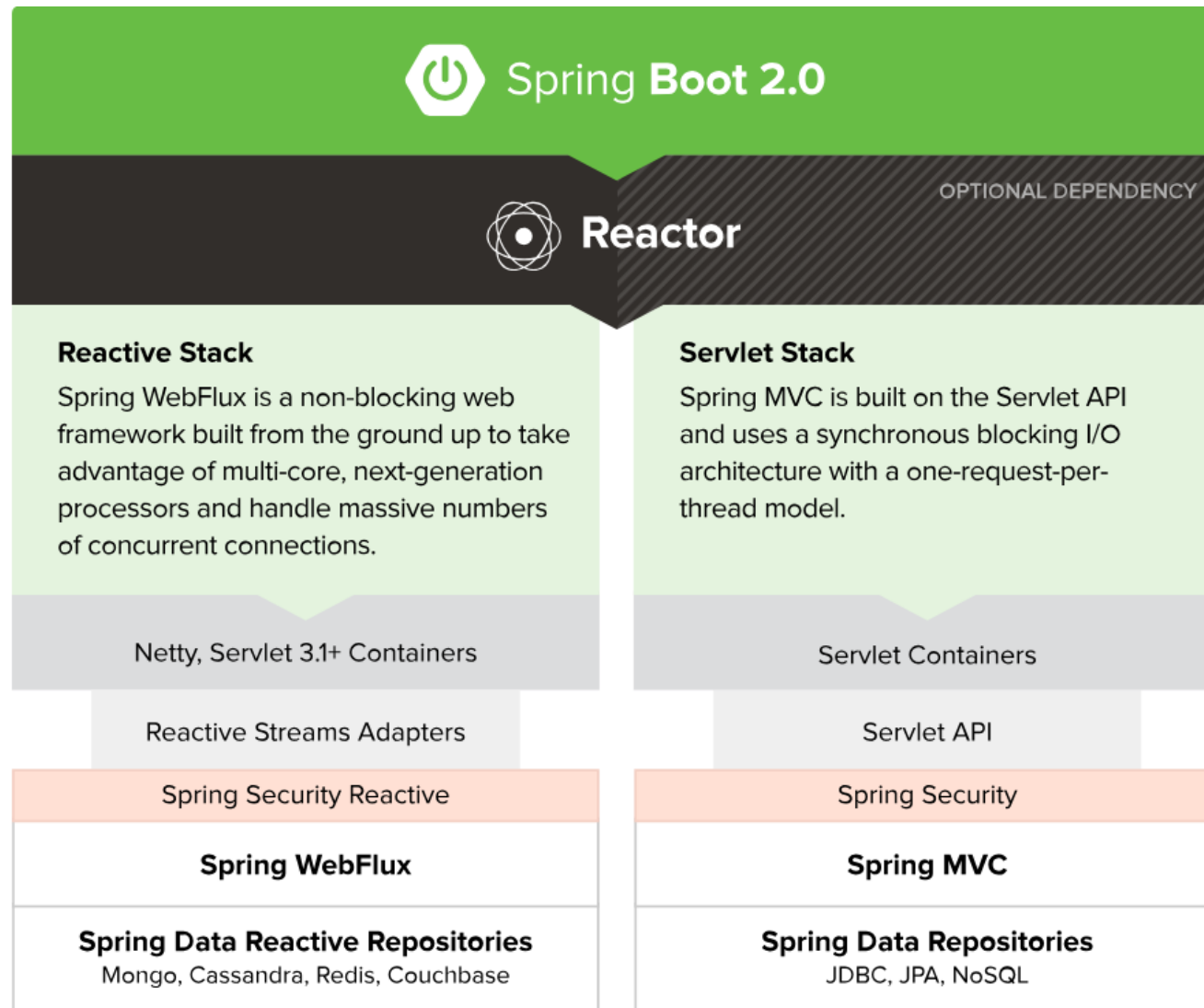
# Spring framework 4



## Spring Framework Runtime



# Spring 5.x et SB 2.0





# Pattern IoC et Injection de dépendance

---

- ❖ Le cœur de Spring se base sur le pattern **IoC**
- ❖ Illustration : Un contrôleur s'appuie sur une couche service qui implémente une interface :

```
public class MovieLister...
    public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = finder.findAll();
        List<Movie> ret = new ArrayList<Movie>() ;
        for (Movie movie : allMovies ) {
            if (!movie.getDirector().equals(arg))
                ret.add(movie);
        }
        return ret;
    }
...
    public interface MovieFinder {
        List<Movie> findAll();
    }
```





# Implémentation ?

---

- ❖ Même si le code est bien découplé via l'utilisation d'interface, comment peut-on insérer une classe concrète qui implémente l'interface *finder* ?

Par exemple, dans le constructeur de la classe *MovieLister*.

```
class MovieLister...  
    private MovieFinder finder;  
    public MovieLister() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```

=> Argh !!! Le contrôleur est alors dépendant de l'implémentation !!



# Solution

---

Pour enlever la dépendance, on délègue l'instanciation de l'implémentation à un framework (Spring, Serveur JavaEE).

Le framework utilise des données de configuration pour trouver l'implémentation à instancier

=> C'est le framework qui prend le contrôle des instanciations

=> Il **injecte** l'implémentation dans la classe contrôleur



# Types d'injection de dépendances

---

❖ Il y a 3 principaux types d'injections de dépendances :

– Par constructeur

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder;  
}
```

– Par setter

```
public void setFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

– Par interface :

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}
```



# Configuration XML

---

```
<beans>
```

```
  <bean id="MovieLister" class="spring.MovieLister">
```

```
    <property name="finder">
```

```
      <ref local="MovieFinder"/>
```

```
    </property>
```

```
  </bean>
```

```
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
```

```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```



# Configuration via annotations

---

```
public class MovieLister {  
    MovieFinder finder ;
```

**@Autowired**

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder ;  
}
```

```
public List<Movie> moviesDirectedBy(String arg) {  
    List<Movie> allMovies = finder.findAll();  
    List<Movie> ret = new ArrayList<Movie>() ;  
    for (Movie movie : allMovies ) {  
        if (!movie.getDirector().equals(arg))  
            ret.add(movie);  
    }  
    return ret;  
}
```



# Injection implicite

---

```
@Controller
public class MovieLister {
    private final MovieFinder finder ;

    public MovieLister(MovieFinder finder) {
        this.finder = finder ;
    }

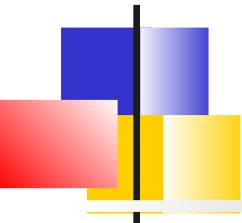
    public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = finder.findAll();
        List<Movie> ret = new ArrayList<Movie>() ;
        for (Movie movie : allMovies ) {
            if (!movie.getDirector().equals(arg))
                ret.add(movie);
        }
        return ret;
    }
}
```



# Avantages de l'injection de dépendances

---

- ❖ L'injection de dépendance apporte d'importants bénéfices :
  - Les composants applicatifs sont **plus faciles à écrire**
  - Ils sont **plus faciles à tester**.  
Les implémentations mock peuvent être injectés lors des tests.
  - Le **typage** des objets est **préservé**.
  - Les **dépendances sont explicites** (à la différence d'une initialisation à partir d'un fichier *properties* ou d'une base de données)
  - La plupart des classes applicatives **ne dépendent pas de l'API du conteneur** et peuvent donc être utilisées avec ou sans le container.
  - Le framework peut injecter des implémentations ayant des **services techniques transverses** (Transaction, sécurité, Trace, Profiling)



# Exemples, services transverses

---

Avec un framework *IoC* comme Spring, un développeur peut :

- Écrire une méthode s'exécutant dans une transaction base de données sans utiliser l'API de transaction
- Rendre une méthode accessible à distance sans utiliser une API remote
- Définir une méthode de gestion applicative sans utiliser JMX
- Définir une méthode gestionnaire de message sans utiliser JMS





# Types de configuration

---

- ❖ Différents choix sont possibles pour configurer le container :
  - **Fichier de configuration XML** :  
Changement sans recompilation, Utilisation de *namespace* spécifique
  - **Fichier properties** : Si la configuration est vraiment simpliste
  - **Annotations dans les sources Java** : Utile lorsque les changements sont peu fréquents
  - **Classe de configuration Java** : Moins verbeux que le XML

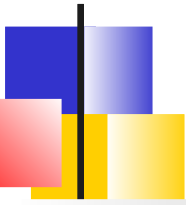


# Cycles de vie des objets gérés

---

❖ Les objets instanciés et gérés par Spring peuvent suivre 3 types de cycle de vie :

- **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des services « stateless »
- **Prototype** : A chaque fois que l'objet est demandé via son nom, une instance est créée.
- **Custom object “scopes”** : Ce sont des objets qui interagissent avec des éléments ne faisant pas partie du container.
  - Certains sont fournis par Spring en particulier les objets liés à la requête ou à la session HTTP
  - Il est assez aisé de mettre en place un système de scope pour les objets (Implémentation de `org.springframework.beans.factory.config.Scope`).



# Spring et les Design Patterns

---

Le framework Spring utilise de nombreux design patterns. Citons :

- **Singleton** : Les beans sont des singletons par défaut
- **DI/IOC** : Pattern central
- **Proxy** : Spring AOP et Remoting
- **Builder** : Pour construire les définitions de Beans (BeanDefinitions) via les classes "BeanDefinitionBuilder"
- **Adapter** : Implémentation par défaut d'une interface (Disparait avec Spring5 et Jdk8)
- **Factory** : Utilisé pour le chargement de Beans via *BeanFactory* et Application context
- **MVC** et **FrontController** : Spring MVC
- **Template** : Pour isoler le code technique (fermeture de connexions et autres)
- ...



# Offre Spring Boot



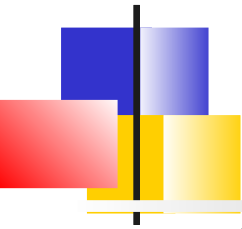
# Introduction

---

Spring Boot a été conçu pour **simplifier le démarrage** et le développement de nouvelles applications Spring

- ne nécessite aucune configuration XML
- Dès la première ligne de code, on a une application fonctionnelle

=> Offrir une expérience de développement simplifiant à l'extrême l'utilisation des technologies existantes



# Essence

---

Spring Boot est un ensemble de bibliothèques qui sont exploitées par un système de build et de gestion de dépendances ( ***Maven*** ou ***Gradle*** )

Il offre également une interface en ligne de commande, qui peut être utilisée pour exécuter et tester des applications : ***Spring CLI***



# Auto-configuration

---

Le concept principal de *SpringBoot* est l'**auto-configuration**

*SpringBoot* est capable de détecter automatiquement la nature de l'application et de configurer les beans Spring nécessaires

- Cela permet de démarrer rapidement et de graduellement surcharger la configuration par défaut pour les besoins de l'application

Les mécanismes sont différents en fonction du langage : Groovy, Java ou Kotlin



# *Groovy*

---

L'écosystème Spring Boot propose un support pour le langage Groovy.

Lors de l'exécution de code Groovy,  
Spring Boot décore le bytecode généré  
en y ajoutant les imports nécessaires





# Hello – Twitter

---

```
@RestController
```

```
class App {
```

```
    @RequestMapping("/")
```

```
    String home() {
```

```
        "hello"
```

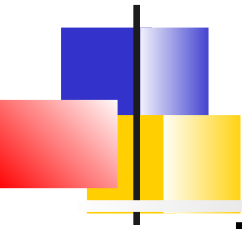
```
    }
```

```
}
```

```
spring run App.groovy
```

Spring Boot analyse le fichier et détermine que l'application est de type web.

Il démarre alors un contexte Spring à l'intérieur d'un conteneur Tomcat embarqué (port par défaut 8080).



# Java

## Gestion des dépendances

---

Dans un environnement Java, Spring Boot simplifie la gestion de dépendances et de leurs versions :

- Il organise les fonctionnalités de Spring en modules.  
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des **"starter" modules**.
- Il fournit un POM parent dont les projets héritent qui gère les versions des dépendances.
- Il propose l'interface web **"Spring Initializr"**, qui peut être utilisée pour générer des configurations Maven ou Gradle



# Starter Modules

---

***spring-boot-starter-web***: librairies des Spring MVC + configuration automatique d'un serveur embarqué (Tomcat, Undertow, Netty).

***spring-boot-starter-data-\**** remonte les dépendances nécessaires pour l'accès aux données avec une technologie particulière (JPA, NoSQL, ...). Il configure automatiquement les datasources ou les factories nécessaires pour la connexion au système de persistance

***spring-boot-starter-security*** librairies de SpringSecurity + configure automatiquement Spring Security pour ajouter à l'application un système d'authentification basique par défaut

***spring-boot-starter-actuator*** remonte un ensemble de dépendances et de beans permettant de surveiller une application en production (métriques, audit sécurité, traces HTTP).



# Projet Java

---

```
package com.infoq.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello";
    }

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



# *@EnableAutoConfiguration*

---

**@EnableAutoConfiguration** sur la classe *Application* informe Boot qu'il doit adopter une approche dogmatique de la configuration pour l'application

La classe *Application* est exécutable, ce qui veut dire que l'application, et son conteneur embarqué, peuvent être démarrés en tant qu'application Java

- Les plugins Maven et Gradle de Boot permettent de produire un "fat jar" exécutable (*mvn package, gradle build*)  
=> Idéal pour les micro-services

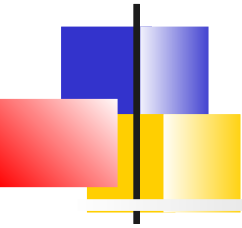


# Personnalisation de la configuration

---

La configuration par défaut peut être surchargée par différents moyens

- Des **fichiers de configuration externe** (*.properties* ou *.yml*) permettent de fixer des valeurs des variables de configuration.  
On peut mettre en place différents fichiers en fonction de profils (correspondant aux environnements)
- Utiliser des **classes spécifiques** au bean que l'on veut surcharger (exemple *AuthenticationManagerBuilder*)
- La **définition de Beans en Java** qui remplacent les beans par défaut
- L'auto-configuration peut également être **désactivée** pour une partie de l'application



# Installation (IDE)



# Versions

---

## Spring Boot 1.x

- Java 6 à Java8
- Spring Framework 4.x
- Maven 3.2+ ou Gradle 2.9 ou 3.x

## Spring Boot 2.x

- Java 8 à Java 11
- Spring Framework 5.x
- Maven 3.3+ Gradle 4.4+





# Spring Tool Suite

---

Distribution basée sur Eclipse,  
*VSCode* ou *Atom* depuis la version 4  
compatible Spring Boot 2

Offrant :

- Assistant de création de projet connecté avec *Spring Initializr*
- Complétion sur les propriétés de configuration.
- Boot Dashboard permettant de facilement démarrer arrêter les applis Spring Boot



# Exemple Eclipse

---

- Assistant pour la création de projet :  
*File → New → Spring starter Project*
- *Clic-droit → Run As Spring Boot App* : Ajoute des configurations spécifiques à Spring Boot pour le démarrage d'application
- Éditeur dédié aux propriétés Spring  
(*application.properties*)

## New Spring Starter Project

Name:

Type:  Packaging:

Java Version:  Language:

Boot Version:

Group:

Artifact:

Version:

Description:

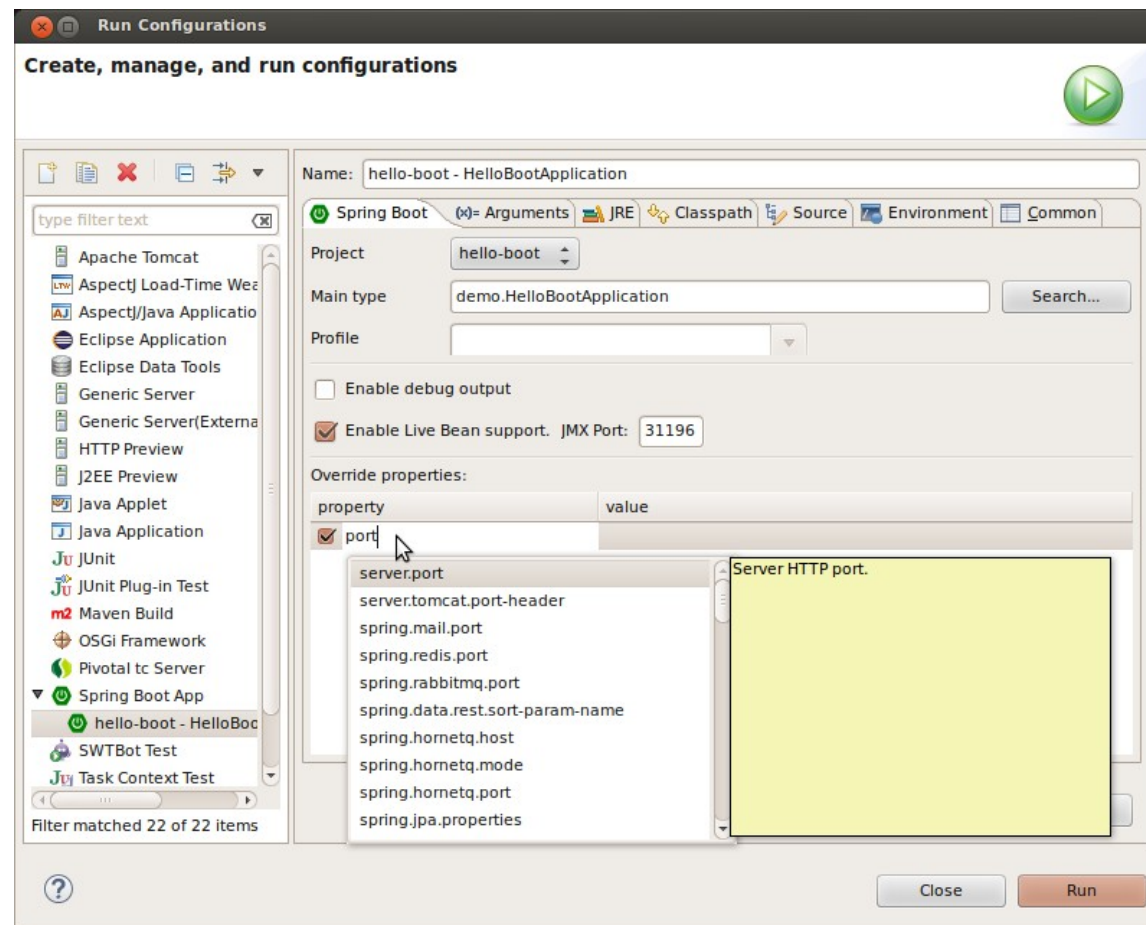
Package:

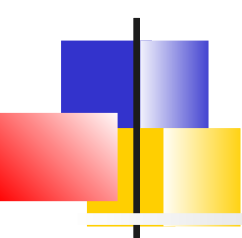
### Dependencies

<input type="checkbox"/> AMQP	<input type="checkbox"/> AOP	<input type="checkbox"/> Actuator	<input type="checkbox"/> Apache Derby
<input type="checkbox"/> Atomikos (JTA)	<input type="checkbox"/> Batch	<input type="checkbox"/> Bitronix (JTA)	<input type="checkbox"/> Cloud Connectors
<input type="checkbox"/> Elasticsearch	<input type="checkbox"/> Facebook	<input type="checkbox"/> Freemarker	<input type="checkbox"/> Gemfire
<input type="checkbox"/> Groovy Templates	<input type="checkbox"/> H2	<input type="checkbox"/> HATEOAS	<input type="checkbox"/> HSQLDB
<input type="checkbox"/> Integration	<input type="checkbox"/> JDBC	<input type="checkbox"/> JMS	<input type="checkbox"/> JPA
<input type="checkbox"/> Jersey (JAX-RS)	<input type="checkbox"/> LinkedIn	<input type="checkbox"/> Mail	<input type="checkbox"/> Mobile
<input type="checkbox"/> MongoDB	<input type="checkbox"/> Mustache	<input type="checkbox"/> MySQL	<input type="checkbox"/> Redis
<input type="checkbox"/> Remote Shell	<input type="checkbox"/> Rest Repositories	<input type="checkbox"/> Security	<input type="checkbox"/> Solr
<input type="checkbox"/> Thymeleaf	<input type="checkbox"/> Twitter	<input type="checkbox"/> Velocity	<input type="checkbox"/> WS
<input checked="" type="checkbox"/> Web	<input type="checkbox"/> Websocket		

Support for full-stack web development, including Tomcat and spring-webmvc

# Propriétés Spring Boot





# Développer avec Spring Boot

---

Gestion des dépendances et starter modules  
Structure projet et principales annotations  
Exécution, Debug  
Propriétés de configuration  
Profils  
Configuration des traces



# Introduction

---

Chaque version de Spring Boot fournit sa liste de dépendances.

=> 1 seule numéro de version fixe l'ensemble des versions des dépendances utilisées .

Lors d'une mise à jour de SpringBoot, toutes les dépendances seront également mises à niveau de manière cohérente.

La liste des dépendances est importée dans le projet via le pom parent



# Fichier *pom* Parent

---

Les projets spring-boot hérite du pom parent ***spring-boot-starter-parent*** qui contient :

- Un niveau de compilation Java en Java6 ou Java8
- L'encodage UTF-8
- L'importation des dépendances
- Des filtres de ressources. En particulier pour les fichiers de configuration *application-\** servant aux configurations spécifiques à des *profils* de build
- La configuration de plugins Maven (*exec plugin, surefire, Git commit ID, shade*)



# Exemple d'utilisation

---

```
<!-- Héritage de Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
<properties>
  <java.version>1.8</java.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```





# Starters

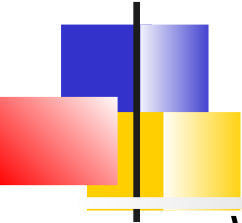
---

Les starters sont des descripteurs de dépendances très pratiques.

- Par exemple un projet voulant utiliser Spring et JPA ajoutera la dépendance sur le starter *spring-boot-starter-data-jpa*

Ils s'appellent tous :

***spring-boot-starter-\****



# Starters les + importants

---

## Web

- \*-**web** : Application web avec *SpringMVC* et API Restful
- \*-**reactive-web** : Application web avec *SpringMVC* et API Restful
- \*-**web-services** : Services Web SOAP

## Cœurs :

- \*-**logging** : Utilisation de logback (Starter par défaut)
- \*-**test** : Test avec Junit, Hamcrest et Mockito
- \*-**security** : Spring Security
- \*-**actuator** : Permet de surveiller et gérer une application SpringBoot
- \*-**devtools** : Fonctionnalités pour le développement



# Autres Starters

---

## Accès aux données en utilisant Spring Data

- \*-**jdbc** : JDBC avec pool de connexions Tomcat
- \*-**jpa** : Accès avec Hibernate et JPA
- \*-**<drivers>** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic)
- \*-**data-cassandra** : Base distribuée Cassandra
- \*-**data-neo4j** : Base de données orienté graphe de Neo4j
- \*-**data-couchbase** : Base NoSQL CouchBase
- \*-**data-redis** : Base NoSQL Redis
- \*-**data-gemfire** : Stockage de données via GemFire
- \*-**data-elasticsearch** : Base documentaire indexée ElasticSearch
- \*-**data-solr** : Base indexée SolR
- \*-**data-mongodb** : Base NoSQL MongoDB
- \*-**jooq** : Accès à des BD relationnelles
- \*-**data-rest** : Utilisation de Spring Data Rest



# Autres starters

---

## Interfaces Web, Mobile REST

- \*-**thymeleaf** : Création d'applications web MVC avec des vues *Thymeleaf*
- \*-**mobile** : Spring Mobile
- \*-**hateoas** : Application RESTful avec Spring Hateoas
- \*-**jersey** : API Restful avec JAX-RS et Jersey
- \*-**websocket** : Spring WebSocket
- \*-**mustache** : Spring MVC avec Mustache
- \*-**groovy-templates** : MVC avec gabarits Groovy
- \*-**freemarker** : MVC avec freemarker



# Autres Starters

---

## I/O

- \*-**batch** : Gestion de batches
- \*-**mail** : Envois de mails
- \*-**cache**: Support pour un cache
- \*-**quartz** : Planification de jobs

## Messaging et JMS

- \*-**integration**: Spring Integration (Couche de + haut niveau)
- \*-**kafka**: Spring et Apache Kafka
- \*-**amqp**: Spring AMQP et Rabbit MQ
- \*-**activemq** : JMS avec Apache ActiveMQ
- \*-**artemis** : JMS messaging avec Apache Artemis



# Autres Starters

---

## Services cloud

*Amazon, Google Cloud, Azure, Cloud Foundry*

## Micro-services, SpringCloud

Services de discovery, de config, de répartition de charge, de proxy, de monitoring, de tracing, de messagerie distribuée, etc ...



# Structure projet et principales annotations



# Structure projet

---

Aucune obligation mais des recommandations :

- Placer la classe *Main* dans le package racine
- L'annoter avec :
  - Les annotations
    - **@EnableAutoConfiguration**
    - **@ComponentScan**
    - **@Configuration**
  - Ou tout simplement :  
**@SpringBootApplication**





# Structure typique

---

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



# @Configuration

---

**@Configuration** indique à Spring que le classe peut définir des beans Spring

- La classe *Main* peut être un bon emplacement pour la configuration
- Mais celle-ci peut être dispersée dans plusieurs autres classes
- L'annotation **@Import** peut être utilisée pour importer les autres classes de configuration



# Exemple

---

```
@Import(DataSourceConfig.class)  
@Configuration  
public class SimpleConfiguration {  
    @Autowired  
    Connection connection;  
  
    @Bean  
    Database getDatabaseConnection(){  
        return connection.getDBConnection();  
    }  
    // Mode code here....  
}
```



# Auto-configuration

---

**@EnableAutoConfiguration** permet de configurer automatiquement des beans Spring en fonction des dépendances qui ont été spécifiées.

- Possibilité de désactiver l'auto-configuration pour certaines parties de l'application.

Ex :

```
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})
```



# Beans et Injection de dépendance

---

Il est possible d'utiliser les différentes techniques de Spring pour définir les beans et leurs injections de dépendances.

La technique la plus simple est généralement la combinaison de :

- L'annotation **@ComponentScan** afin que *Spring* trouve les beans (Inclut dans *@SpringBootApplication*)
- L'annotation **@Autowired** dans le constructeur d'un bean ou sur une déclaration
- L'utilisation de l'injection implicite, attribut final + paramètre du constructeur
- Les annotations **@Component**, **@Service**, **@Repository**, **@Controller** qui permettent de définir des beans



# Exemple

---

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

    @Autowired
    public DatabaseAccountService(RiskAssessor riskAssessor) {
        this.riskAssessor = riskAssessor;
    }

    // ...

}
```



# Exécution, *DevTools* et Debug



# Exécution

---

Le projet est en généralement packagé en un jar. Il peut être démarré par :

- `java -jar target/myproject-0.0.1-SNAPSHOT.jar`
- Ou pour permettre le debug :  
`java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000,suspend=n -jar target/myproject-0.0.1-SNAPSHOT.jar`

Les plugins Maven ou Gradle proposent également un objectif permettant de démarrer l'application (Cela peut être utile pour des tests d'intégration par exemple)

```
mvn spring-boot:run
gradle bootRun
```





# Rechargement de code

---

Les applications Spring Boot étant de simples applications Java, le rechargement de code à chaud doit être supporté.

- => Cela permet de ne pas avoir à redémarrer l'application à chaque modification

Pour des solutions plus complète à cet épineux problème, voir les projets *JRebel* ou *Spring Loaded*



# Outils de développement

---

Le module ***spring-boot-devtools*** peut être ajouté via une dépendance

Ce module apporte notamment :

- Ajout automatique de propriétés de configuration propre au développement. Ex : *spring.thymeleaf.cache=false*
- Redémarrage automatique rapide lors d'un changement de classe. (Un peu contradictoire avec le rechargement à chaud de classes).
- ...



# Redémarrage automatique

---

Il est possible d'affiner le comportement du redémarrage automatique, :

- Possibilité d'exclure des ressources provoquant le redémarrage.

*Ex application.properties :*

```
spring.devtools.restart.exclude=static/**,public/**
```

- Possibilité de le désactiver :

```
System.setProperty("spring.devtools.restart.enabled", "false");
```

- Possibilité d'utiliser un fichier déclencheur



# Propriétés de configuration



# Propriétés de configuration

---

Spring Boot permet d'externaliser la configuration afin de s'adapter à différents environnements

On peut utiliser des fichiers **properties** ou **YAML**, des variables d'environnement ou des arguments de commande en ligne.

Les valeurs des propriétés peuvent être injectées directement dans les beans

- via l'annotation **@Value**
- Via l'abstraction Spring : **Environment**
- Ou les mapper dans un objet structuré via l'annotation **@ConfigurationProperties**



# Priorités

De nombreux niveaux de propriétés différents mais en résumé l'ordre des propriétés est :

1. *spring-boot-devtools.properties* si *devtools* est activé
2. Les propriétés de test
3. **La ligne de commande. Ex : *--server.port=9000***
4. Environnement REST, Servlet, JNDI, JVM
5. **Variables d'environnement de l'OS**
6. Propriétés avec des valeurs aléatoires
7. **Propriétés spécifiques à un profil**
8. ***application.properties* , *yaml***
9. Annotation *@PropertySource* dans la configuration
10. Les propriétés par défaut spécifiées par *SpringApplication.setDefaultProperties*



# *application.properties (.yml)*

---

Spring recherche un fichier ***application.properties (.yml)*** dans les emplacements suivants :

- Un sous-répertoire *config*
- Le répertoire courant
- Un package *config* dans le classpath
- A la racine du classpath



# Valeur filtrée

---

Les valeurs d'une propriété sont filtrées.  
Elles peuvent ainsi faire référence à  
une propriété déjà définie.

```
app.name=MyApp
```

```
app.description=${app.name} is a Boot app.
```





# Valeurs aléatoires

---

Utile pour injecter des valeurs aléatoires  
(par exemple pour des données de test).

Peut produire des nombres entiers, des  
longs, des *uuid* ou des chaînes

```
my.secret=${random.value}  
my.number=${random.int}  
my.bignumber=${random.long}  
my.uuid=${random.uuid}  
my.number.less.than.ten=${random.int(10)}  
my.number.in.range=${random.int[1024,65536]}
```



# Format YAML

---

***YAML*** (*Yet Another Markup Language*)  
est une extension de JSON, il est très  
pratique et très compact pour spécifier  
des données de configuration  
hiérarchique.

... mais également très sensible, à  
l'indentation par exemple



# Exemple *.yml*

---

```
environments:  
  dev:  
    url: http://dev.bar.com  
    name: Developer Setup  
  prod:  
    url: http://foo.bar.com  
    name: My Cool App
```

## Produit :

```
environments.dev.url=http://dev.bar.com  
environments.dev.name=Developer Setup  
environments.prod.url=http://foo.bar.com  
environments.prod.name=My Cool App
```



# Listes

---

Les listes YAML sont représentées par des propriétés avec un index.

```
my:
  servers:
    - dev.bar.com
    - foo.bar.com
```

Devient :

```
my.servers[0]=dev.bar.com
my.servers[1]=foo.bar.com
```



# Injection de propriété : *@Value*

---

La première façon de lire une valeur configurée est d'utiliser l'annotation ***@Value***.

```
@Value("${my.property}")  
private String myProperty ;
```

Dans ce cas, aucun contrôle n'est effectué sur la valeur effective de la propriété



# Vérifier les propriétés

---

Il est possible de forcer la vérification des propriétés au démarrage.

- Utiliser une classe annotée par **@ConfigurationProperties** et **@Validated**
- Positionner des contraintes de *javax.validation* sur les attributs de la classes



# Exemple

---

**@ConfigurationProperties(prefix="connection")**

**@Validated**

```
public class ConnectionProperties {
```

```
    private String username;
```

**@NotNull**

```
    private RemoteAddress remoteAddress;
```

```
    // ... getters and setters
```

```
    public static class RemoteAddress {
```

**@NotEmpty**

```
        public String hostname;
```

```
        // ... getters and setters
```

```
    }
```

```
}
```



# Profils





# Introduction

---

Les **profils** fournissent un moyen de séparer des parties de la configuration et de les rendre disponible seulement dans certains environnements :  
Production, Test, Debug, etc.



# Annotations utilisant les profils

---

Tout *@Component* ou *@Configuration* peut être marqué avec **@Profile** pour limiter son chargement

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```



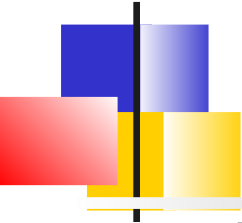
# Activation des profils

---

Les profils sont activés généralement par la propriété ***spring.profiles.active*** qui peut être positionnée :

- Dans un fichier de configuration
- En commande en ligne via :  
***--spring.profiles.active=dev,hsqldb***
- Programmatically, via :  
***SpringApplication.setAdditionalProfiles(...)***

Plusieurs profils peuvent être activés simultanément



# Propriétés spécifiques via *\*.properties*

---

Les propriétés spécifiques à un profil  
(ex : intégration, production) peuvent  
être dans des fichiers nommés :

***application-{profile}.properties***



# Propriété spécifiques à un profil via fichier *.yml*

---

Il est possible de définir plusieurs profils dans le même document.

```
server:  
  address: 192.168.1.100
```

---

```
spring:  
  profiles: development
```

```
server:  
  address: 127.0.0.1
```

---

```
spring:  
  profiles: production
```

```
server:  
  address: 192.168.1.120
```



# Inclusion de profils

---

La directive ***spring.profiles.include*** permet d'inclure des profils.

Par exemple :

```
---  
my.property: fromyamlfile  
---  
spring.profiles: prod  
spring.profiles.include:  
- proddb  
- prodm
```



# Journalisation et traces



# Introduction

---

Spring utilise *Common Logging* en interne mais permet de choisir son implémentation

Des configurations sont fournies pour :

- Java Util Logging
- Log4j2
- Logback (défaut)





# Format des traces

---

Une ligne contient les informations suivantes :

- Timestamp à la ms
- Niveau de trace : ERROR, WARN, INFO, DEBUG or TRACE.
- Process ID
- Un séparateur --- .
- Nom de la thread entouré de [].
- Le nom du Logger <=> Nom de la classe .
- Un message
- Une note entouré par des []



# Configurer les traces via Spring

---

Par défaut, Spring affiche les messages de niveau ERROR, WARN et INFO sur la console

- *java -jar myapp.jar -debug* : Active les messages de debug
- Propriétés **logging.file** et **logging.path** afin que les messages soient stockés dans un fichier
- Changer les niveaux de logs au niveau des logger via *application.properties*  
logging.level.root=WARN  
logging.level.org.springframework.web=DEBUG  
logging.level.org.hibernate=ERROR



# Configuration personnalisée via l'implémentation

---

Il est possible de personnaliser la configuration en utilisant les fichiers propres à chaque implémentation :

- Logback : *logback-spring.xml, logback-spring.groovy, logback.xml ou logback.groovy*
- Log4j2 : *log4j2-spring.xml ou log4j2.xml*
- JDK (Java Util Logging) : *logging.properties*



# Extensions Logback

---

Spring propose des extensions à Logback permettant des configurations avancées.

Il faut dans ce cas utiliser le fichier ***logback-spring.xml***

- Multi-configuration en fonction de profil
- Utilisation de propriétés dans la configuration



# Persistence

---

Principes de SpringData  
SpringBoot et SpringJPA  
Spring Boot et NoSQL, Exemple  
MongoDB  
Spring Data Rest



# Introduction

---

La mission de ***Spring Data*** est de fournir un modèle de programmation simple et cohérent pour l'accès aux données quelque soit la technologie sous-jacente (Relationnelle, NoSQL, Cloud, Moteur de recherche)

*Spring Data* est donc le projet qui encadre de nombreux sous-projets collaborant avec les sociétés éditrices de la solution de stockage



# Apports de *SpringData*

---

Les apports sont :

- Une abstraction de la notion de **repository** et de **mapping** objet
- La **génération dynamique de requêtes** basée sur des règles de nommage des méthodes
- Des classes **d'implémentations** de bases pouvant être utilisées : *\*Template.java*
- Un support pour **l'audit** (Date de création, dernier changement)
- La possibilité d'intégrer du code **spécifique** au repository
- Configuration **Java ou XML**
- Intégration avec les contrôleurs de **Spring MVC** via **SpringDataRest**



# Interfaces *Repository*

---

L'interface centrale de Spring Data est ***Repository***  
(C'est une classe marqueur)

L'interface prend en arguments de type

- la **classe persistante** du domaine
- son **id**.

La sous-interface ***CrudRepository*** ajoute les méthodes CRUD

Des abstractions spécifiques aux technologies sont également disponibles *JpaRepository*, *MongoRepository*, ...





# Interface *CrudRepository*

---

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // ... more functionality omitted.
}
```



# Déduction de la requête

---

Après avoir étendu l'interface, il est possible de définir des méthodes permettant d'effectuer des requêtes vers le repository

A l'exécution Spring fournit un bean implémentant l'interface et les méthodes fournies.

Spring doit déduire les requêtes à effectuer :

- Soit à partir du **nom** de la méthode
- Soit de l'annotation **@Query**



# Exemple

---

```
public interface MemberRepository
    extends JpaRepository<Member, Long> {

    /**
     * Tous les membres ayant un email particulier.
     * @param email
     * @return
     */
    public List<Member> findByEmail(String email);

    /**
     * Chargement de la jointure one2Many.
     * @param id
     * @return
     */
    @Query("from Member m left join fetch m.documents where m.id =:id")
    public Member fullLoad(Long id);
}
```



# Méthodes de sélection de données

---

Lors de l'utilisation du nom de la méthode, celles-ci doivent être préfixées comme suit :

- Recherche : *find\*By\**
- Comptage : *count\*By\**
- Suppression : *delete\*By\**
- Récupération : *get\*By\**

La première *\** peut indiquer un flag (comme *Distinct* par exemple)

Le terme **By** marque la fin de l'identification du type de requête

Le reste est parsé et spécifie la clause **where** et éventuellement **orderBy**



# Résultat du parsing

---

Les noms des méthodes consistent généralement de propriétés de l'entité combinées avec *AND* et *OR*

Des opérateurs peuvent également être précisés :  
*Between, LessThan, GreaterThan, Like*

Le flag *IgnoreCase* peut être attribué individuellement aux propriétés ou de façon globale

```
findByLastnameIgnoreCase(...)
```

```
findByLastnameAndFirstnameAllIgnoreCase(...)
```

La clause *order* de la requête peut être précisée en ajoutant *OrderBy(Asc/Desc)* à la fin de la méthode



# Expression des propriétés

---

Les propriétés ne peuvent faire référence qu'aux propriétés directes des entités

Il est cependant possible de référencer des propriétés imbriquées :

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Ou si ambiguïté

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```



# Gestion des paramètres

---

En plus des paramètres concernant les propriétés, SpringBoot est capable de reconnaître les paramètres de types **Pageable** ou **Sort** pour appliquer la pagination et le tri dynamiquement

Les valeurs de retours peuvent alors être :

- *Page* connaît le nombre total d'éléments en effectuant une requête *count*,
- *Slice* ne sait que si il y a une page suivante

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



# Limite

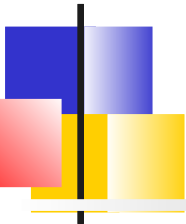
---

Les mots clés ***first*** et ***top*** permettent de limiter les entités retournées

Elles peuvent être précisées avec un numérique

```
User findFirstByOrderByLastnameAsc();  
Slice<User> findTop3ByLastname(String lastname,  
                                Pageable pageable);
```





# Mots-clés supportés pour JPA

---

And, Or Is, Equals, Between,  
LessThan, LessThanEqual,  
GreaterThan, GreaterThanEqual,  
After, Before, IsNull,  
IsNotNull, NotNull, Like,  
NotLike, StartingWith,  
EndingWith, Containing, OrderBy,  
Not, In, NotIn, True, False,  
IgnoreCase



# Utilisation des *NamedQuery* JPA

---

Avec JPA le nom de la méthode peut correspondre à une *NamedQuery*.

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {
}

public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmailAddress(String emailAddress);
}
```



# Utilisation de *@Query*

La requête peut également être exprimée dans le langage d'interrogation du repository via l'annotation ***@Query*** :

- Méthode la plus prioritaire
- A l'avantage de se situer sur la classe *Repository*

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
  
    @Query("select u from User u where u.firstname like %?1")  
    List<User> findByFirstnameEndsWith(String firstname);  
  
    @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")  
    List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);  
  
    @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")  
    User findByLastnameOrFirstname(@Param("lastname") String lastname,  
                                   @Param("firstname") String firstname);  
}
```



# SpringBoot et Spring Data-JPA



# Apports Spring Boot

---

*spring-boot-starter-data-jpa* fournit les dépendances suivantes :

- Hibernate
- Spring Data JPA .
- Spring ORMs

Par défaut, toutes les classes annotée par *@Entity*, *@Embeddable* ou *@MappedSuperclass* sont scannées et prises en compte

L'emplacement de départ du scan peut être réduit avec ***@EntityScan***



# Configuration source de données / Rappels

---

Pour accéder à une BD relationnelle, Java utilise la notion de ***DataSource*** (interface représentant un pool de connections BD)

Une data source se configure via :

- Une URL JDBC
- Un compte base de donnée
- Un driver JDBC
- Des paramètres de dimensionnement du pool



# Support pour une base embarquée

---

Spring Boot peut configurer automatiquement les bases de données H2, HSQL et Derby.

Il n'est pas nécessaire de fournir d'URL de connexion, la dépendance Maven suffit :

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



# Base de production

---

Les bases de production peuvent également être auto-configurées.

Les propriétés requises à configurer sont :

`spring.datasource.url=jdbc:mysql://localhost/test`

`spring.datasource.username=dbuser`

`spring.datasource.password=dbpass`

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

Voir *DataSourceProperties* pour l'ensemble des propriétés disponibles

L'implémentation du pool sous-jacent privilégie celle de Tomcat dans Spring Boot 1 et Hikari dans Spring Boot 2. Cela peut être surchargée par la propriété *spring.datasource.type*





# Configuration du pool

---

Des propriétés sont également spécifiques à l'implémentation de pool utilisée.

Par exemple pour Tomcat :

# Timeout en ms si pas de connexions dispo.

```
spring.datasource.tomcat.max-wait=10000
```

# Taille maximale du pool

```
spring.datasource.tomcat.max-active=50
```

# Validation de la connexion avant de la retourner.

```
spring.datasource.tomcat.test-on-borrow=true
```



# Propriétés

---

Les bases de données JPA embarquées sont créées automatiquement.

Pour les autres, il faut préciser la propriété ***spring.jpa.hibernate.ddl-auto***

Ou utiliser les propriétés natives d'Hibernate

- Elles peuvent être spécifiées en utilisant le préfixe *spring.jpa.properties.\**

Ex :

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```



# Configuration des Templates

---

Les beans ***JdbcTemplate*** et ***NamedParameterJdbcTemplate*** sont auto-configurés et peuvent donc être directement injectés

Leur comportement peut être personnalisé par les propriétés *spring.jdbc.template.\**

Ex :

```
spring.jdbc.template.max-rows=500
```



# Example

---

@Repository

```
public class UserDaoImpl implements UserDao {
```

```
    private final String INSERT_SQL = "INSERT INTO USERS(name, address, email) values(:name,:email)";
```

```
    private final String FETCH_SQL_BY_ID = "select * from users where record_id = :id";
```

@Autowired

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

```
public User create(final User user) {
```

```
    KeyHolder holder = new GeneratedKeyHolder();
```

```
    SqlParameterSource parameters = new MapSqlParameterSource()
```

```
        .addValue("name", user.getName())
```

```
        .addValue("email", user.getEmail());
```

```
    namedParameterJdbcTemplate.update(INSERT_SQL, parameters, holder);
```

```
    user.setId(holder.getKey().intValue());
```

```
    return user;
```

```
}
```

```
public User findUserById(int id) {
```

```
    Map parameters = new HashMap();
```

```
    parameters.put("id", id);
```

```
    return namedParameterJdbcTemplate.queryForObject(FETCH_SQL_BY_ID, parameters, new UserMapper());
```

```
}
```

```
}
```



# Code JDBC ou JPA

---

On peut également se faire injecter les beans permettant de coder à un niveau plus bas :

- Au niveau JDBC, en se faisant injecter la *DataSource*
- Au niveau JPA, en se faisant injecter l'*entityManager* ou l'*entityManagerFactory*



# *OpenInView*

---

Lors d'une application Web, Spring Boot enregistre par défaut l'intercepteur *OpenEntityManagerInViewInterceptor* afin d'appliquer le pattern “**Open EntityManager in View**” permettant d'éviter les *LazyException* dans les vues

Si ce n'est pas le comportement voulu :  
`spring.jpa.open-in-view = false`



# SpringBoot et NoSQL



# Introduction

---

Spring Boot fournit des configurations automatique pour *Redis*, *MongoDB*, *Neo4j*, *Elasticsearch*, *Solr* et *Cassandra*;

## Exemple MongoDB

`spring-boot-starter-data-mongodb`





# Connexion à une base MongoDB

---

SpringBoot créé un bean  
***MongoDbFactory*** se connectant à  
l'URL *mongodb://localhost/test*

La propriété ***spring.data.mongodb.uri***  
permet de changer l'URL

- L'autre alternative est de déclarer sa propre *MongoDbFactory* ou un bean de type *Mongo*



# Entité

---

Spring Data offre un ORM entre les documents MongoDB et les objets Java.

Une classe du domaine peut être annoté par **@Id**:

```
import org.springframework.data.annotation.Id;
public class Customer {
    @Id
    public String id;
    public String firstName;
    public String lastName;

    public Customer() {}
    ...
    // getters and setters
}
```



# Mongo Repository

---

SpringData propose également des implémentations de Repository pour les base NoSQL

- Il suffit d'avoir les bonnes dépendances dans le classpath :  
*spring-boot-starter-data-mongodb*

L'exemple pour JPA est alors également valable dans cet environnement



# Usage

---

```
@Controller
public class MyController {

    @Autowired
    private CustomerRepository repository;

    @Override
    public void doIt(throws Exception {

        repository.deleteAll();

        // save a couple of customers
        repository.save(new Customer("Alice", "Smith"));
        Repository.findByName("Smith") ;
        ...
    }
}
```



# *MongoTemplate*

---

Un bean ***MongoTemplate*** est également auto-configuré

- C'est cette classe qui implémente les méthodes de l'interface Repository
- Mais elle peut également être injectée et utilisée directement.



# Mongo Embarqué

---

Il est possible d'utiliser un Mongo embarqué

Il suffit d'avoir des dépendances vers :

```
de.flapdoodle.embed:de.flapdoodle.embed.mongo
```

Le port utilisé est soit déterminé

aléatoirement soit fixé par la propriété :

```
spring.data.mongodb.port
```

Les traces de MongoDB sont visibles si *slf4f*  
est dans le classpath



# Spring Data Rest



# Objectifs du projet

---

**Spring Data REST** est un sous-projet de Spring Data

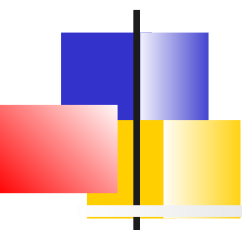
Il permet d'exporter automatiquement les Repositories Spring Data en des ressources REST

- En utilisant les concepts hypertexte, il permet à un client de découvrir automatiquement les méthodes proposées par le Repository et de les intégrer dans des ressources hypertexte
- En plus court : « *Traduire les concepts de domaine dans les concepts web les plus appropriés* »

*Spring Data Rest* combine plusieurs sous-projets de Spring :

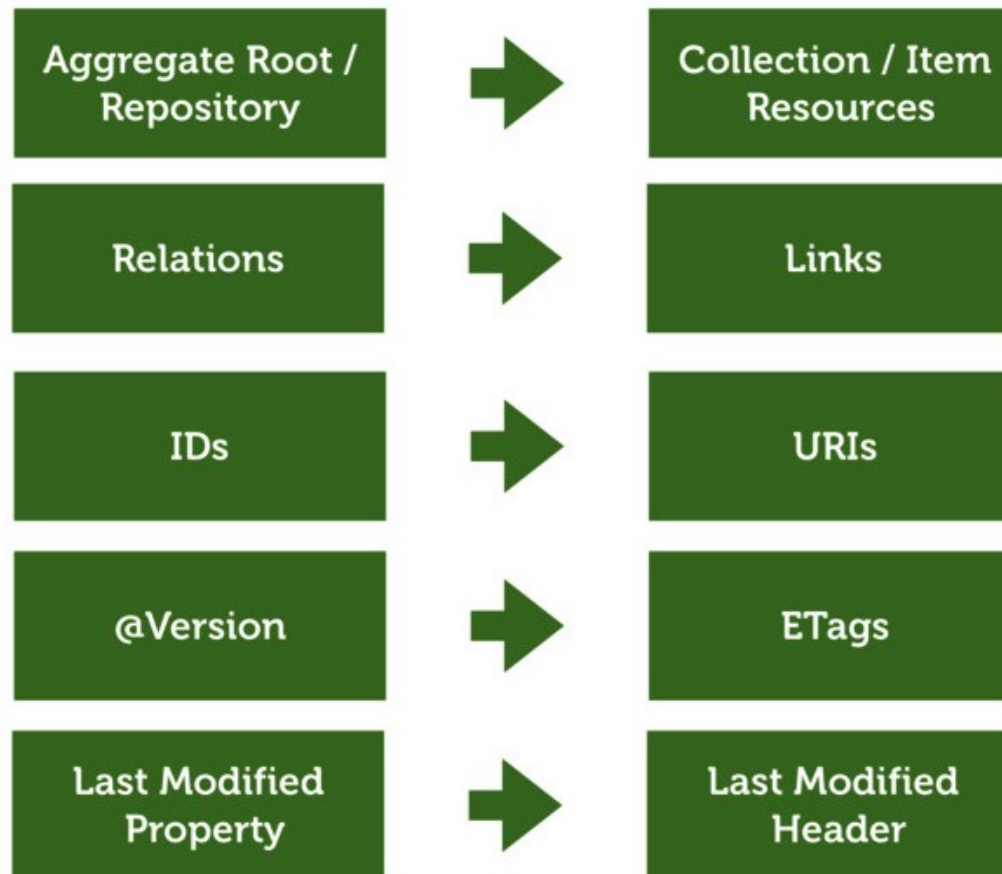
- *Spring Data*
- *Spring HATEOAS*
- *Spring MVC* (Génération de contrôleurs)





# Mapping Domain/Hypertexte

---





# RestFul API

URI	GET	PUT	POST	DELETE
Collection, <a href="http://api.example.com/resources/">http://api.example.com/resources/</a>	Liste les URI et autres détails des membres de la collection	Remplace la collection avec une autre collection	Crée une nouvelle entrée dans la collection. La nouvelle URI créé automatiquement et retournée par l'opération.	Supprime la collection.
Element, <a href="http://api.example.com/resources/item17">http://api.example.com/resources/item17</a>	Récupère une vue détaillé de l'élément dans le format approprié	Remplace ou créé l'élément	Traite l'élément comme une collection et y ajoute une entrée	Supprime l'élément



# HATEOAS

---

Spring Data REST suit les principes de ***HATEOAS***.

HATEOAS est une des contraintes de l'architecture REST : Les liens hypertextes doivent être utilisés pour naviguer à travers l'API.



# HAL Format

---

***Hypertext Application Language*** est un format descriptif simple permettant de lier (via des hyperlink) les différentes ressources de l'API.

- HAL est basée sur JSON
- HAL Browser (bibliothèque Javascript) transforme le format JSON en HTML

=> L'API devient alors explorable et auto-documentée.



# Exemple

```
"_links": {
  "self": { "href": "/orders" }, // Représente l'URI de la ressource : Une collection de order
  "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}", "templated": true }], // Compact URI
  "next": { "href": "/orders?page=2" }, // Navigation par page
  "ea:find": { "href": "/orders{id}", "templated": true }, // Un lien avec gabarit pour recherche un ordre
particulier
  "ea:admin": [{ "href": "/admins/2", "title": "Fred" }, { "href": "/admins/5", "title": "Kate" }] // 2 entités liées
},
"currentlyProcessing": 14, // Propriété simple de l'URI
"shippedToday": 20,
"_embedded": { // Ressources embarquées
  "ea:order": [{
    "_links": {
      "self": { "href": "/orders/123" },
      "ea:basket": { "href": "/baskets/98712" },
      "ea:customer": { "href": "/customers/7809" }
    },
    "total": 30.00,
    "status": "shipped"
  }, {
    "_links": {
      "self": { "href": "/orders/124" },
      "ea:basket": { "href": "/baskets/97213" },
      "ea:customer": { "href": "/customers/12369" }
    },
    "total": 20.00,
    "status": "processing"
  }]
}
}
```



# Spring Data REST

---

***Spring Data REST*** est construit au dessus de Spring MVC.

Il crée automatiquement les contrôleurs, les convertisseur JSON liés aux repositories de Spring Data.



# starter

---

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



# Configuration

---

Propriétés préfixées par ***spring.data.rest.*** dans *application.properties*

- ***basePath*** : Base URI
- ***defaultPageSize*** : Nombre d'items dans une page
- ***maxPageSize*** : Maximum dans une page
- ***pageParamName*** : Nom du paramètre pour délectionner une page
- ***limitParamName*** : Nom du paramètre pour le nombre d'items par page
- ***sortParamName*** : Nom du paramètre pour le tri
- ***defaultMediaType*** : Media par défaut si pas spécifié
- ***returnBodyOnCreate*** : Corps doit être retourné lors d'une création
- ***returnBodyOnUpdate*** : Corps doit être retourné lors d'une mise à jour





# Contrôle de l'API

---

Le contrôle de l'API Rest exposée s'effectue via l'interface repository

Par défaut, pour l'interface :

```
public interface OrderRepository extends CrudRepository<Order, Long> { }
```

Spring Data REST expose :

- une ressource collection à /orders.
- Une ressource détail avec le gabarit d'URI /orders/{id}.
- Les méthodes HTTP qui interagissent avec ces ressources sont associées à des méthodes de *CrudRepository*



# Découverte de l'API

---

```
curl -v http://localhost:8080/
```

```
< HTTP/1.1 200 OK
```

```
< Content-Type: application/hal+json
```

```
{ "_links" : {  
  "orders" : {  
    "href" : "http://localhost:8080/orders"  
  },  
  "profile" : {  
    "href" : "http://localhost:8080/api/alps"  
  }  
}  
}
```



# Installation HAL Browser

---

Pour profiter d'une interface web permettant de parcourir les endpoints exposés, il suffit d'une dépendance :

```
<dependency>  
  <groupId>org.springframework.data</groupId>  
  <artifactId>spring-data-rest-hal-browser</artifactId>  
</dependency>
```



# Application Web

---

Rappels Spring MVC  
Support pour les APIs Rest  
Spring Boot pour les APIs Rest



# Introduction

---

*SpringBoot* est adapté pour le développement web

Le module starter ***spring-boot-starter-web*** permet de charger le framework Spring MVC

Spring MVC permet de déclarer des beans de type

- ***@Controller*** ou ***@RestController***
- Dont les méthodes peuvent être associées à des requêtes http via ***@RequestMapping***



# Rappels Spring MVC

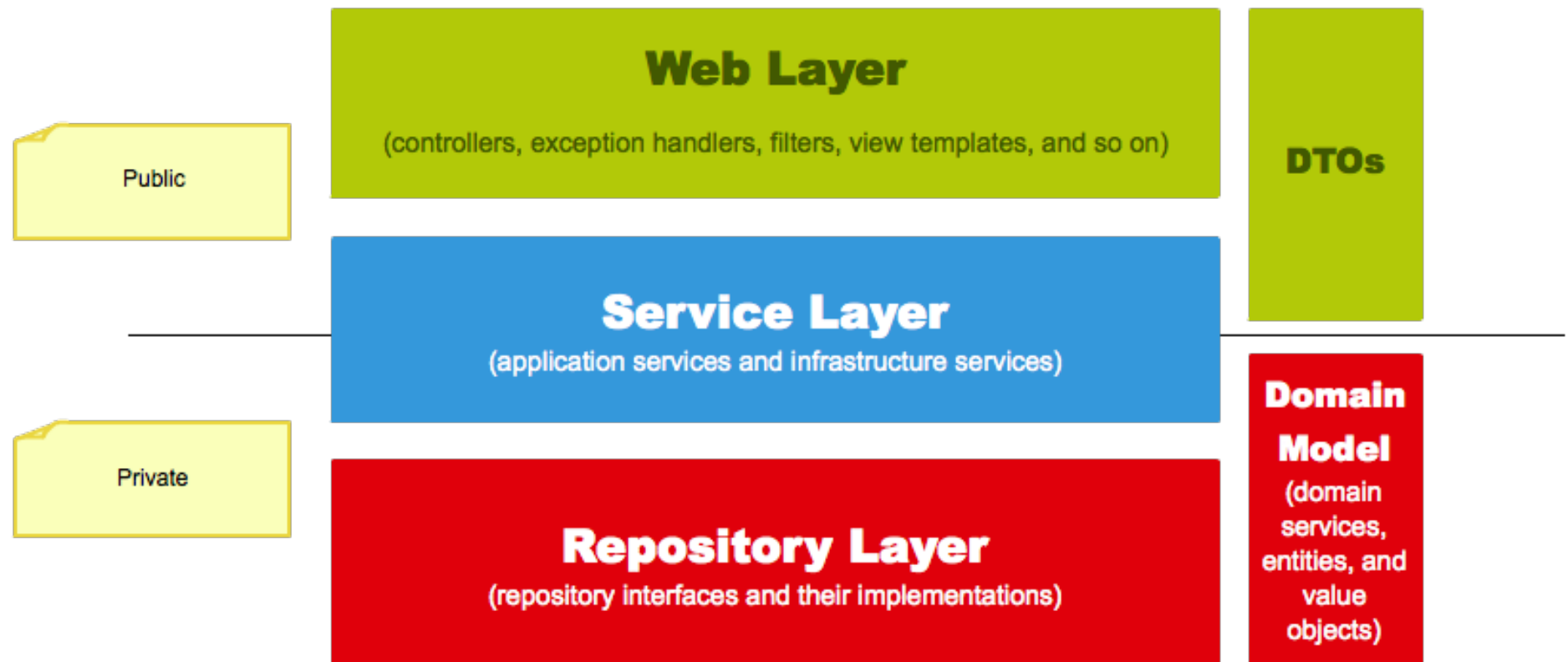
---

Le framework MVC (Spring-Web Model-View-Controller) est conçu autour du servlet ***DispatcherServlet*** qui distribue des requêtes aux contrôleurs

- L'association contrôleur / requête est effectuée via l'annotation ***@RequestMapping***
- Les contrôleurs classiques ont la responsabilité de préparer les données du modèle via des interfaces de type ***Map***.  
Le traitement de la requête est ensuite transféré à une technologie de rendu (*JSP, Velocity, Freemarker, Thymeleaf*) qui sélectionne un gabarit de page et génère du HTML
- Les contrôleurs REST ont la responsabilité de construire une réponse HTTP (code de retour, entêtes, ... ) dont le corps est généralement un document ***json***



# Architecture classique projet





# *@Controller, @RestController*

Les annotations **@Controller**, **@RestController** se positionnent sur de simples POJOs dont les méthodes publiques sont généralement accessible via HTTP

## **@Controller**

```
public class HelloWorldController {  
  
    @RequestMapping("/helloWorld")  
    public String helloWorld(Model model) {  
        model.addAttribute("message", "Hello World!");  
        return "helloWorld";  
    }  
}
```





# Types des arguments de méthode

---

Une méthode d'un contrôleur peut prendre des arguments de type :

- La requête ou réponse HTTP (ServletRequest, HttpServletRequest, spring.WebRequest, ...)
- La session HTTP (HttpSession)
- La locale, la time zone
- Les streams d'entrée/sortie
- La méthode HTTP
- L'utilisateur authentifié par HTTP (Principal)
- Une Map, org.springframework.ui.Model ou org.springframework.ui.ModelMap représentant le modèle exposé à la vue
- Errors ou validation.BindingResult : Les erreurs d'une précédente soumission de formulaire

Si l'argument est d'un autre type, il nécessite des **annotations** afin que Spring puisse effectuer les conversions nécessaires à partir de la requête HTTP

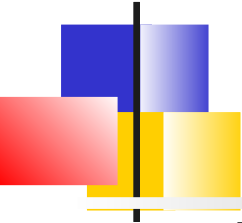


# Annotations sur les arguments

---

Les annotations sur les arguments permettent d'associer un argument à une valeur de la requête HTTP :

- **@PathVariable** : Une partie de l'URI
- **@RequestParam** : Un paramètre HTTP
- **@RequestHeader** : Une entête
- **@RequestBody** : Contenu de la requête en utilisant un *HttpMessageConverter*
- **@RequestPart** : Une partie d'une requête multi-part
- **@SessionAttribute** : Un attribut de session
- **@RequestAttribute** : Un attribut de requête
- **@ModelAttribute** : Un attribut du modèle (requête, session, etc.)
- **@Valid** : S'assure que les contraintes sur l'argument sont valides



# Types des valeurs de retours des méthodes

---

Les types possibles sont :

- Des modèles : *ModelAndView*, *Model*, *Map*
- Des Vues : *View*, *String*
- *void* : Si le contrôleur a lui-même généré la réponse
- Un corps de réponse HTTP converti via un *HttpConverter* (REST JSON)
- ...



# @RequestMapping

---

## @RequestMapping

- Placer au niveau de la classe indique que toutes les méthodes du gestionnaires seront relatives à ce chemin
- Au niveau d'une méthode, l'annotation précise :
  - **path** : Valeur fixe ou gabarit d'URI
  - **method** : Pour limiter la méthode à une action HTTP
  - **produce/consume** : Préciser le format des données d'entrée/sortie



# Compléments *@RequestMapping*

---

Des variantes pour limiter à une méthode :

*@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping*

Limiter à la valeur d'un paramètre ou d'une entête :

*@GetMapping(path = "/pets", headers = "myHeader=myValue", params = "myParam=myValue")*

Utiliser des expressions régulières

*@GetMapping(value = "/ex/bars/{numericId:[\\d]+}")*

Utiliser des propriétés de configuration

*@RequestMapping("\${context}")*



# Gabarits d'URI

---

Un gabarit d'URI permet de définir des noms de variable :

<http://www.example.com/users/{userId}>

L'annotation **@PathVariable** associe la variable à un argument de méthode

```
@GetMapping("/owners/{ownerId}")  
public String findOwner(@PathVariable String  
    ownerId, Model model) {
```



# Compléments

---

Un argument *@PathVariable* peut être de type simple, Spring fait la conversion automatiquement

Si *@PathVariable* est utilisée sur un argument *Map <String, String>*, l'argument est renseigné avec toutes les variables du gabarit

Un gabarit peut être construit à partir de la combinaison des annotations de type et de méthode



# Paramètres avec *@RequestParam*

---

```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...

}
```





# Formats d'entrée/sorties

---

Il est également de spécifier une liste de type de média permettant de filtrer sur l'entête *Content-type* de la requête HTTP

En entrée, précise le format attendu

```
@PostMapping(path = "/pets", consumes = "application/json")  
public void addPet(@RequestBody Pet pet, Model model) {
```

Ou en sortie, précise le format généré :

```
@GetMapping(path = "/pets/{petId}",  
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE)  
@ResponseBody  
public Pet getPet(@PathVariable String petId, Model model) {
```



# @RequestBody et convertisseur

---

L'annotation **@RequestBody** utilise des *HTTPMessageConverter* qui se basent sur l'entête *content-type* de la requête

- *StringHttpMessageConverter*
- *FormHttpMessageConverter*  
(*MultiValueMap<String, String>*)
- *ByteArrayHttpMessageConverter*
- *MappingJackson2HttpMessageConverter* : JSON
- *MappingJackson2XmlHttpMessageConverter* : XML
- ...



# Spring MVC pour les APIs Rest



# *@RestController*

---

Si les contrôleurs n'implémentent qu'une API Rest, ils peuvent être annotés par ***@RestController***

Ces contrôleurs ne produisent que des réponses au format JSON, XML  
=> Pas nécessaire de préciser *@ResponseBody*



# Exemple pour des données JSON

---

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}/customers", method=RequestMethod.GET)
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(value="/{user}", method=RequestMethod.DELETE)
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

    @RequestMapping(path = "/Members", method = RequestMethod.POST)
    public ResponseEntity<Member> register(@Valid @RequestBody Member member) {

        member = memberRepository.save(member);

        return new ResponseEntity<>(member,HttpStatus.CREATED);
    }
}
```



# Sérialisation JSON

---

Un des principales problématiques des interfaces REST et la conversion des objets du domaine Java au format JSON.

Des frameworks spécialisés sont utilisés (Jackson, Gson) mais en général le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bi-directionnelles entre entités
- Adaptation aux besoins de l'interface de front-end
- Optimisation du volume de données échangées



# API Jackson

---

Jackson propose 3 principales API :

- Une API de streaming capable de lire ou écrire du contenu JSON sur un modèle évènementiel (analogue au Stax Parser de XML)
- Une API basée sur un modèle d'arbre. Un contenu JSON est transformé ou produit à partir d'une représentation mémoire en arbre (analogue au parser DOM)
- Du Data Binding permettant de convertir des POJO via ses accesseurs ou via des annotations (analogue à JAXB)

Les sérialisations/désérialisations sont effectuées généralement par des **ObjectMapper**



# Annotations Jackson

---

**@JsonProperty, @JsonGetter, @JsonSetter,  
@JsonAnyGetter, @JsonAnySetter, @JsonIgnore,  
@JsonIgnoreProperty, @JsonIgnoreType :**

Permettant de définir les propriétés JSON

**@JsonRootName** : Arbre JSON

**@JsonSerialize, @JsonDeserialize** : Indique des  
dé/sérialiseurs spécialisés

**@JsonManagedReference, @JsonBackReference,  
@JsonIdentityInfo** : Gestion des relations  
bidirectionnelles

....





# Spring Boot et APIs Rest



# Auto-configuration

---

*SpringBoot* effectue des configurations automatiques pour Spring MVC. Les principaux apports sont :

- Démarrage automatique des serveur embarqué
- Configuration par défaut pour servir des ressources statiques (index.html, favicon, Webjars)
- Détection et configuration automatique du langage de templating
- Configuration automatique des *HttpMessageConverters* permettant un comportement par défaut des sérialiseurs



# Personnalisation de la configuration

---

- Ajouter un bean de type ***WebMvcConfigurer*** et implémenter les méthodes voulues :
  - Configuration MVC (ViewResolver, ViewControllers)
  - Configuration du CORS
  - Configuration d'intercepteurs
  - ...



# Exemple Cross-origin

Le *crosss-origin resource sharing*, i.e pouvoir faire des requêtes vers des serveurs différents que son serveur d'origine peut facilement s'implémenter via la définition d'un bean qui rajoute les entêtes nécessaires :

```
@Configuration
public class MyConfiguration {

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/api/**");
            }
        };
    }
}
```



# Exemple *Intercepteurs*

---

```
@SpringBootApplication
public class MyApplication {

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

    @Bean
    public WebMvcConfigurer adapter() {
        return new WebMvcConfigurer() {
            @Override
            public void addInterceptors(InterceptorRegistry registry) {
                System.out.println("Adding interceptors");
                registry.addInterceptor(new MyInterceptor()).addPathPatterns("/**");

                super.addInterceptors(registry);
            }
        };
    }
}
```



# Gestion des erreurs

---

*Spring Boot* associe ***/error*** à la page d'erreur globale de l'application

- Un comportement par défaut en REST ou en Web permet de visualiser la cause de l'erreur

Pour remplacer le comportement par défaut :

- Implémenter ***ErrorController*** et l'enregistrer comme Bean
- Ajouter un bean de type ***ErrorAttributes*** qui remplace le contenu de la page d'erreur
- Ajouter une classe annotée par ***@ControllerAdvice*** pour personnaliser le contenu renvoyé lors d'une exception



# Exemple *@ControllerAdvice*

## **@ControllerAdvice**

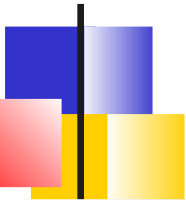
```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {
```

```
    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFoundException.class})
```

```
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }
```

## **@Override**

```
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```



# Sérialisation JSON spécifique

L'annotation **@JsonComponent** facilite l'enregistrement de sérialiseurs/désérialiseurs Jackson

Elle doit être placée sur des implémentations de *JsonSerializer* et *JsonDeserializer* ou sur des classes contenant des inner-class de ce type

## @JsonComponent

```
public class Example {  
    public static class Serializer extends JsonSerializer<SomeObject> {  
        // ...  
    }  
    public static class Deserializer extends  
JsonDeserializer<SomeObject> {  
        // ...  
    }  
}
```





# Appels de service REST

---

*Spring* fournit la classe ***RestTemplate*** facilitant les appels aux services REST.

*Spring Boot* ne fournit pas de bean auto-configuré de type *RestTemplate* mais il auto-configuré un ***RestTemplateBuilder*** permettant de les créer



# Exemple

---

```
@Service
public class MyBean {
    private final RestTemplate restTemplate;

    public MyBean(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate =
            restTemplateBuilder.basicAuthorization("user", "password")
                               .build();
    }

    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details",
                                         Details.class,
                                         name);
    }
}
```



# Mise en place de Swagger

---

La mise en place de Swagger dans un environnement *SpringBoot* est assez direct :

- Ajouter les dépendances dans *pom.xml*
- Se créer une classe configuration définissant un bean de type *Docket* (filtre des annotations) et autorisant les configurations par défaut
- Utiliser les annotations Swagger

=> Accéder à */swagger-ui.html*



# Dépendances

---

```
<dependency>
```

```
  <groupId>io.springfox</groupId>
```

```
  <artifactId>springfox-swagger2</artifactId>
```

```
  <version>2.9.2</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>io.springfox</groupId>
```

```
  <artifactId>springfox-swagger-ui</artifactId>
```

```
  <version>2.9.2</version>
```

```
</dependency>
```



# Configuration

---

```
@Configuration
@Profile("swagger")
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())

            .build();
    }
}
```



# Sécurité et Spring Boot

---

Rappels Spring Security  
Modèles stateful/stateless  
Apports de Spring Boot  
Exemple JWT  
*oAuth2*



# Rappels Spring Security



# Rappels *Spring Security*

---

*Spring Security* gère principalement 2 domaines de la sécurité :

- **L'authentification** : S'assurer de l'identité de l'utilisateur ou du système
- **L'autorisation** : Vérifier que l'utilisateur ou le système ait accès à une ressource.

*Spring Security* facilite la mise en place de la sécurité sur les applications Java EE en

- se basant sur des fournisseurs d'authentification :
  - Spécialisés
  - Ou s'intégrant avec des standards (LDAP, OpenID, Kerberos, PAM, CAS, OAuth2)
- permettant la configuration des contraintes d'accès





# Principe et mécanisme

---

Comme les autres modules de Spring, *Spring Security* nécessite une configuration de beans. Elle peut être provoquée par l'annotation **@EnableWebSecurity**

La configuration crée alors le filtre **springSecurityFilterChain** responsable de tous les aspects de la sécurité :

- Protéger les URLs
- Soumettre les login/mot de passe
- Rediriger vers le formulaire d'authentification,
- Etc ...

Il est en plus nécessaire de créer un initialiseur (*SecurityWebApplicationInitializer*) qui :

- enregistrer le filtre pour toutes les requêtes de l'application
- Ajoute un *ContextLoaderListener* capable de charger une objet de type **WebSecurityConfigurerAdapter** permettant de personnaliser la configuration

Si en plus on désire ajouter de la sécurité au niveau des méthodes : **@EnableGlobalMethodSecurity**



# Quelques Filtres de *springSecurityFilterChain*

---

***UsernamePasswordAuthenticationFilter*** : Répond par défaut à */login*, récupère les paramètres *username* et *password* et appelle le gestionnaire d'authentification

***SessionManagementFilter*** : Gestion de la collaboration entre la session *http* et la sécurité

***BasicAuthenticationFilter*** : Traite les entêtes d'autorisation d'une authentification basique

***SecurityContextPersistenceFilter*** : Responsable de stocker le contexte de sécurité (par exemple dans la session *http*)



# Personnalisation

---

La personnalisation consiste à implémenter une classe de type ***WebSecurityConfigurer*** et de surcharger le comportement par défaut en surchargeant les méthodes appropriées. En particulier :

- ***void configure(HttpSecurity http)*** : Permet de définir les ACLs et la gestion de la sécurité => Influe sur la chaîne de filtre
- ***void configure(AuthenticationManagerBuilder auth)*** : Permet de définir le gestionnaire d'authentification qui peut être fourni par Spring (inMemory, jdbc, ldap, ...) ou complètement personnalisé par l'implémentation d'un bean ***UserDetailsService***



# Exemple

## *WebSecurityConfigurerAdapter*

```
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .authorizeRequests() // ACLs  
        .antMatchers("/resources/**", "/signup", "/about").permitAll()  
        .antMatchers("/admin/**").hasRole("ADMIN")  
        .antMatchers("/db/**").access("hasRole('ADMIN') and hasRole('DBA')")  
        .anyRequest().authenticated()  
        .and()  
        .formLogin() // Page de login  
        .loginPage("/login")  
        .permitAll()  
        .and()  
        .logout() // Comportement du logout  
        .logoutUrl("/my/logout")  
        .logoutSuccessUrl("/my/index")  
        .invalidateHttpSession(true)  
        .addLogoutHandler(logoutHandler)  
        .deleteCookies(cookieNamesToClear) ;  
}
```



# Définition des filtres

---

L'ordre des filtres a une grosse importance.

A priori, les méthodes accessibles sur *HttpSecurity* permettent d'activer les filtres sans se soucier de l'ordre dans lequel il seront activés.

Pour modifier l'enchaînement des filtres à un plus bas niveau, il est possible d'utiliser directement les méthodes ***addFilter\****



# Debug de la sécurité

---

Pour debugger la configuration :

- Afficher le bean *springSecurityFilterChain* et visualiser la chaîne de filtre configurée

Pour debugger l'exécution :

- Activer les traces de DEBUG :

`logging.level.org.springframework.security=DEBUG`



# Exemple

## *AuthenticationManagerBuilder*

---

@Configuration

```
public class InMemorySecurityConfiguration extends  
    WebSecurityConfigurerAdapter {
```

@Override

```
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
```

```
    auth.inMemoryAuthentication().withUser("user").password("password").  
        roles("USER")  
        .and().withUser("admin").password("password").  
        roles("USER", "ADMIN");
```

```
    }  
}
```



# Personnalisation via *UserDetailsService*

---

La personnalisation de l'authentification peut s'effectuer en fournissant sa propre classe implémentant

## ***UserDetailsService***

L'interface contient une seule méthode :

```
public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException
```

- La méthode est responsable de retourner à partir d'un login, un objet de type *UserDetails* encapsulant le mot de passe et les rôles

C'est le framework qui vérifie que le mot de passe saisi correspond.

- La configuration s'effectue comme suit :

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





# Exemple

---

```
import org.springframework.security.core.userdetails.User ;
...
@Service
public class UserDetailsServiceImpl implements UserDetailsService{
    @Autowired
    private AccountRepository accountRepository;

    @Transactional(readOnly = true)
    public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {
        Account account = accountRepository.findByLogin(login);
        if ( account == null )
            throw new UsernameNotFoundException("Invalides login/mot de passe");
        Set<GrantedAuthority> grantedAuthorities = new HashSet<>();
        for (Role role : account.getRoles()){
            grantedAuthorities.add(new SimpleGrantedAuthority(role.getLibelle()));
        }
        return new User(account.getLogin(), account.getPassword(),
            grantedAuthorities);
    }
}
```



# Password Encoder

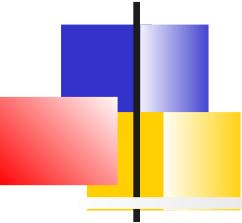
---

Spring Security 5 nécessite que les mots de passes soient encodés

Il faut alors définir un bean de type  
***PasswordEncoder***

L'implémentation recommandée est  
*BcryptPasswordEncoder*

```
@Bean
PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```



# *{noop}*

---

Si les mots de passes sont stockés en clair, il faut les préfixer par ***{noop}*** afin que Spring Security n'utilise pas d'encodeur

```
public UserDetails loadUserByUsername(String login) throws UsernameNotFoundException {  
    Member member = memberRepository.findByEmail(login);  
    if ( member == null )  
        throw new UsernameNotFoundException("Invalides login/mot de passe");  
    Set<GrantedAuthority> grantedAuthorities = new HashSet<>();  
  
    return new User(member.getEmail(), "{noop}" + member.getPassword(), grantedAuthorities);  
}
```



## Modèle stateful/stateless



# Application Web et API Rest

---

Les application web (stateful) et les APIs REST (stateless) n'ont pas la même stratégie pour la gestion de la sécurité.

- Dans une application stateful, les informations liées à l'authentification sont stockées dans la session utilisateur (cookie).
- Dans une application stateless, les droits de l'utilisateur sont transmis à chaque requête



# Processus d'authentification appli web back-end

---

1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier :
  1. En redirigeant vers une page de login
  2. En fournissant les entêtes pour une authentification basique du navigateur .
3. Le navigateur renvoie une réponse au serveur :
  1. Soit le POST de la page de login
  2. Soit les entêtes HTTP d'authentification.
4. Le serveur décide si les crédits sont valides :
  1. si oui. L'authentification est stockée dans la session, la requête originelle est réessayée, si les droits sont suffisants la page est retournée sinon un code 403
  2. Si non, le serveur redemande une authentification.
5. L'objet *Authentication* contenant l'utilisateur et ses rôles est présent dans la session. Il est récupérable à tout moment par `SecurityContextHolder.getContext().getAuthentication()`

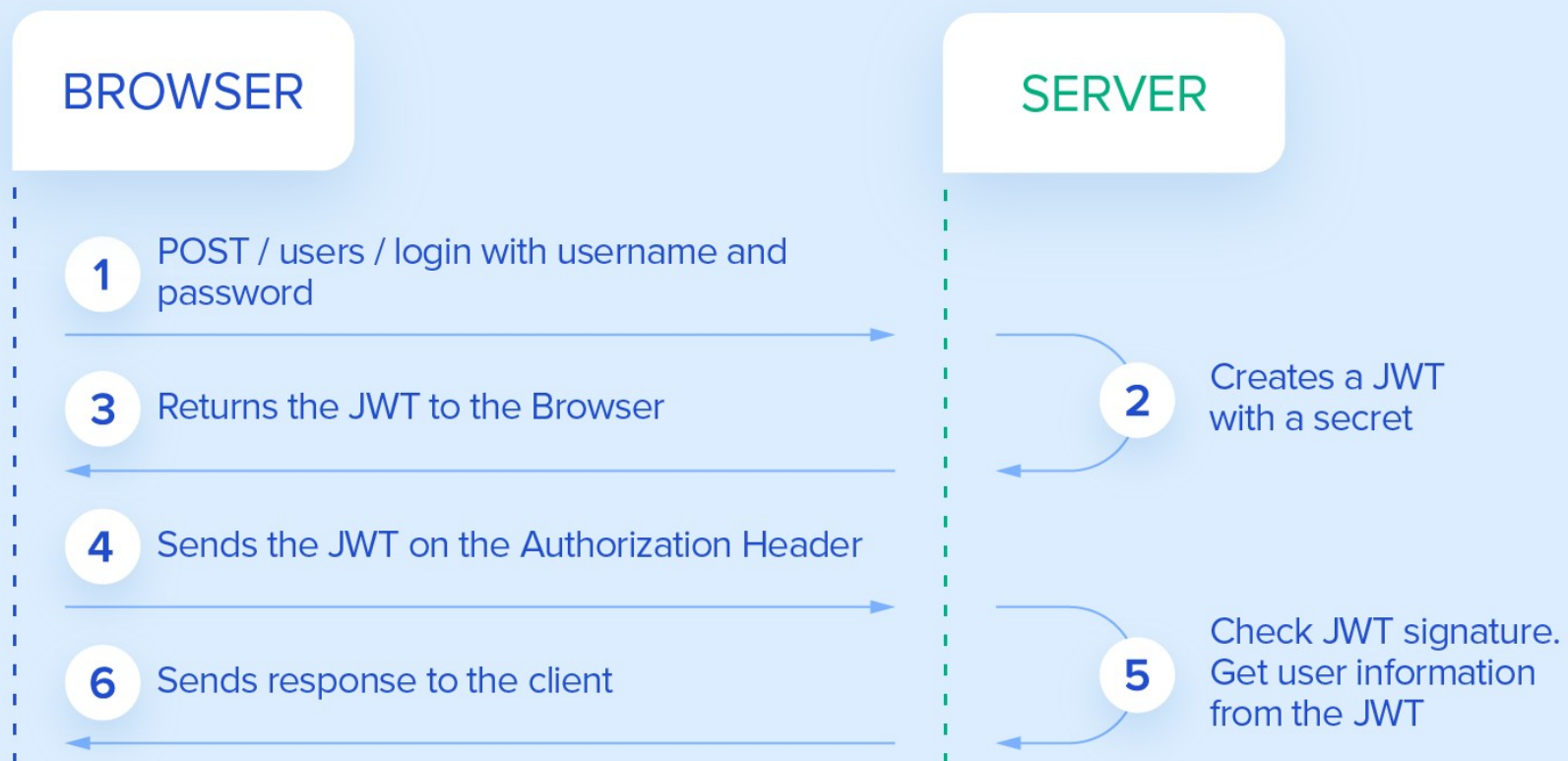


# Processus d'authentification appli REST

---

1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier en envoyant une réponse 403.
3. Le navigateur propose un formulaire de login puis envoie le formulaire sur un serveur d'authentification (peut être différent que le serveur d'API)
4. Le serveur d'authentification décide si les créden-tiels sont valides :
  1. si oui. Il génère un token avec un délai de validité
  2. Si non, le serveur redemande une authentification .
5. Le client récupère le jeton et l'associe à toutes les requêtes vers l'API
6. Le serveur de ressources décrypte le jeton et déduit les droits de l'utilisateur. Il autorise ou interdit l'accès à la ressource

# Authentication Rest







# Apports de Spring Boot



# Apports de SpringBoot

---

Si *Spring Security* est dans le classpath, la configuration par défaut :

- Sécurise toutes les URLs de l'application web par l'authentification formulaire
- Un gestionnaire d'authentification simpliste est configuré pour permettre l'identification d'un unique utilisateur



# Gestionnaire d'authentification par défaut

---

Le gestionnaire d'authentification par défaut définit un seul utilisateur *user* avec un mot de passe aléatoire qui s'affiche sur la console au démarrage.

Les propriétés peuvent être changées via *application.properties* et le préfixe *security*.

*security.user.name= myUser*

*security.user.password=secret*



# Autres fonctionnalités par défaut

---

D'autres fonctionnalités sont automatiquement obtenues :

- Les chemins pour les ressources statiques standard sont ignorées (*/css/\*\**, */js/\*\**, */images/\*\**, */webjars/\*\** et *\*\*/favicon.ico*).
- Les événements liés à la sécurité sont publiés vers *ApplicationEventPublisher* via *DefaultAuthenticationEventPublisher*
- Des fonctionnalités communes de bas niveau (HSTS, XSS, CSRF, caching)



# Personnalisation

---

Pour désactiver la configuration par défaut, on peut ajouter un bean avec l'annotation ***@EnableWebSecurity***.

Sinon, la personnalisation consiste à définir une classe *@Configuration* de type *WebSecurityConfigurerAdapter* permettant de :

- Définir des ACLs sur les ressources : Implémenter la méthode *protected void configure(HttpSecurity http)*
- De définir son gestionnaire d'authentification :
  - Utiliser un bean de type *AuthenticationManager*
  - Ou en configurer via *AuthenticationManagerBuilder*



# Example

---

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/", "/home").permitAll()
                .anyRequest().authenticated()
                .and()
            .formLogin()
                .loginPage("/login")
                .permitAll()
                .and()
            .logout()
                .permitAll();
    }

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        auth
            .inMemoryAuthentication()
                .withUser("user").password("password").roles("USER");
    }
}
```



# Gestionnaires d'authentification fournis

---

Spring Security fournit certains gestionnaires d'authentification qu'il suffit alors de configurer.

Citons :

*jdbcAuthentication(), ldapAuthentication()*

Exemple :

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {

    auth.jdbcAuthentication().
        dataSource(dataSource).
        usersByUsernameQuery("select login,password,true from users where login = ?").
        authoritiesByUsernameQuery("select username,authority from authorities where
        login =?");
}
```



# SSL

---

SSL peut être configuré via les propriétés préfixées par ***server.ssl.\****

Par exemple :

```
server.port=8443
```

```
server.ssl.key-store=classpath:keystore.jks
```

```
server.ssl.key-store-password=secret
```

```
server.ssl.key-password=another-secret
```

Par défaut si SSL est configuré, le port 8080 disparaît.

Si l'on désire les 2, il faut configurer explicitement le connecteur réseau





## Exemple JWT



# JWT

**JSON Web Token (JWT)** est un standard ouvert défini dans la RFC 75191.

Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties.

La sécurité consiste en la vérification de l'intégrité des données à l'aide d'une signature numérique. (HMAC ou RSA).

Dans le cadre d'une application REST SpringBoot, le jeton contient les informations d'authentification d'un user :  
Subject + Rôles

Différentes implémentations existent en Java,  
dont *io.jsonwebtoken*



# Exemple JWT de JHipster

## 1. Configuration chaîne de filtres

---

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .csrf().disable() // Jeton csrf n'est plus nécessaire
        .and() // Rien dans la session HTTP
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
        .and()
        .authorizeRequests() // ACLs
        .antMatchers("/api/authenticate").permitAll()
        .anyRequest().authenticated()
        .and()
        .apply(securityConfigurerAdapter()); // Configuration JWT
}

private JWTConfigurer securityConfigurerAdapter() {
    return new JWTConfigurer(tokenProvider);
}
```



# *Exemple JWT de JHipster*

## *2. Ajout du filtre JWT*

---

```
public class JWTConfigurer extends
    SecurityConfigurerAdapter<DefaultSecurityFilterChain, HttpSecurity> {

    public static final String AUTHORIZATION_HEADER = "Authorization";

    private TokenProvider tokenProvider;

    public JWTConfigurer(TokenProvider tokenProvider) {
        this.tokenProvider = tokenProvider;
    }

    @Override
    public void configure(HttpSecurity http) throws Exception {
        JWTFilter customFilter = new JWTFilter(tokenProvider);
        http.addFilterBefore(customFilter,
UsernamePasswordAuthenticationFilter.class);
    }
}
```



# Exemple JWT de JHipster

## 3. Implémentation du filtre JWT

---

```
public class JWTFilter extends GenericFilterBean {
    private TokenProvider tokenProvider; // Codage/Décodage du Token
    public JWTFilter(TokenProvider tokenProvider) {this.tokenProvider = tokenProvider;    }

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain
filterChain)
        throws IOException, ServletException {
        HttpServletRequest httpRequest = (HttpServletRequest) servletRequest;
        String jwt = resolveToken(httpRequest);
        if (StringUtils.hasText(jwt) && this.tokenProvider.validateToken(jwt)) {
            Authentication authentication = this.tokenProvider.getAuthentication(jwt);
            SecurityContextHolder.getContext().setAuthentication(authentication);
        }
        filterChain.doFilter(servletRequest, servletResponse);
    }

    private String resolveToken(HttpServletRequest request){
        String bearerToken = request.getHeader(JWTConfigurer.AUTHORIZATION_HEADER);
        if (StringUtils.hasText(bearerToken) && bearerToken.startsWith("Bearer ")) {
            return bearerToken.substring(7, bearerToken.length());
        }
        return null;
    }
}
```



# Exemple JWT de JHipster

## 4. Création du jeton TokenProvider

---

```
public String createToken(Authentication authentication, Boolean rememberMe) {  
    String authorities =  
        authentication.getAuthorities().stream().map(GrantedAuthority::getAuthority)  
            .collect(Collectors.joining(","));  
  
    long now = (new Date()).getTime();  
    Date validity = new Date(now + this.tokenValidityInMilliseconds);  
  
    return Jwts.builder()  
        .setSubject(authentication.getName())  
        .claim(AUTHORITIES_KEY, authorities)  
        .signWith(SignatureAlgorithm.HS512, secretKey)  
        .setExpiration(validity)  
        .compact();  
}
```



# Exemple JWT de JHipster

## 5. A partir du jeton retrouver l'authentification

```
public Authentication getAuthentication(String token) {  
    Claims claims = Jwts.parser()  
        .setSigningKey(secretKey)  
        .parseClaimsJws(token)  
        .getBody();  
  
    Collection<? extends GrantedAuthority> authorities =  
        Arrays.stream(claims.get(AUTHORITIES_KEY).toString().split(","))  
            .map(SimpleGrantedAuthority::new)  
            .collect(Collectors.toList());  
  
    User principal = new User(claims.getSubject(), "", authorities);  
  
    return new UsernamePasswordAuthenticationToken(principal, token, authorities);  
}
```



## Rappels *oAuth2*





# Rappels OAuth2 – rôles du protocole

---

Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

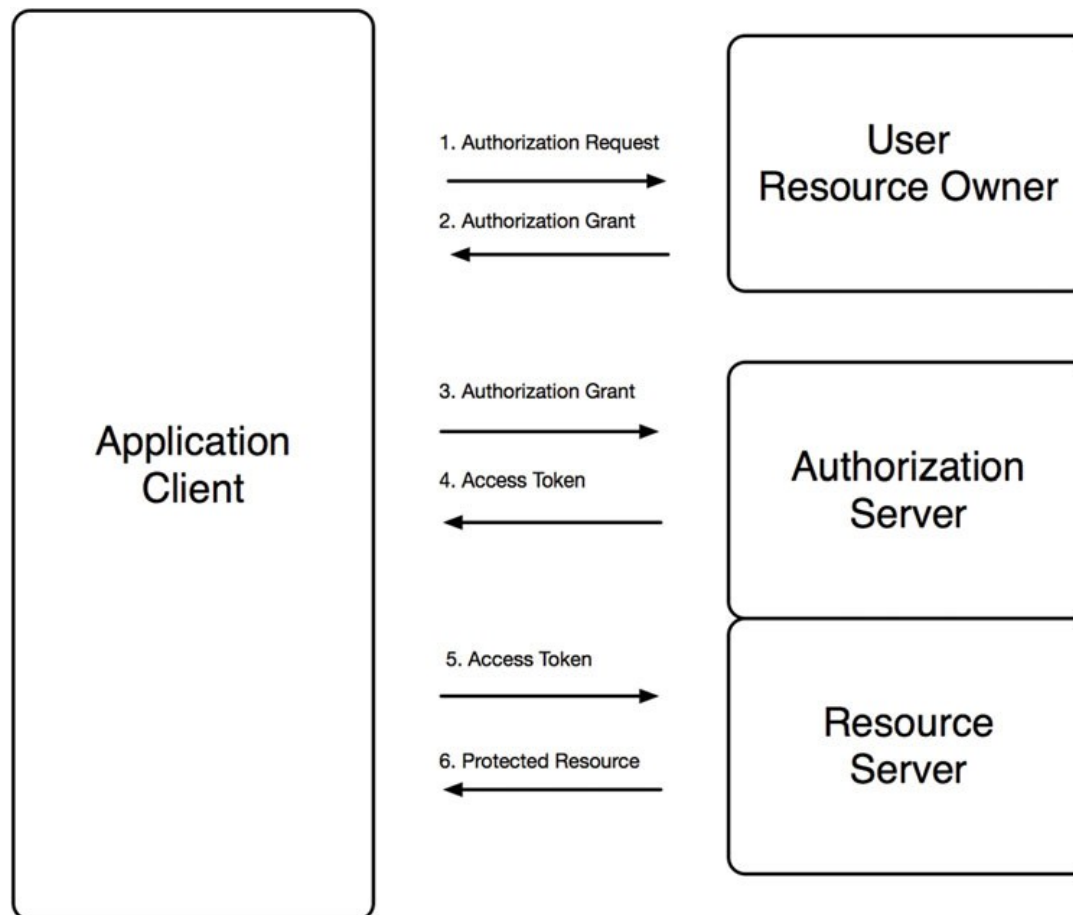
Le **serveur de ressources** est l'API utilisée pour accéder aux ressources protégées

Le **serveur d'autorisation** est le serveur qui autorise un client à accéder aux ressources en lui fournissant un jeton. Il peut demander l'approbation de l'utilisateur

**L'utilisateur** est la personne qui donne accès à certaines parties de son compte

Rq: Un participant du protocole peut jouer plusieurs rôles

# Protocole *oAuth2*





# Scénario

---

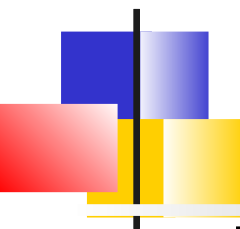
Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)

Obtenir l'autorisation de l'utilisateur.  
(4 types de grant)

Vérifier la réponse du service *oAuth* (state)

Obtention du token (date d'expiration)

Appel de l'API pour obtenir les informations voulues en utilisant le token



# Jetons

---

Les Tokens sont des chaînes de caractères aléatoire générées par le serveur d'autorisation

Les jetons sont ensuite présents dans les requêtes HTTP et contiennent des informations sensibles  
=> HTTPS

Il y a 2 types de token

Le **jeton d'accès**: Il a une durée de vie limité.

Le **Refresh Token**: Délivré avec le jeton d'accès. Il est renvoyer au serveur d'autorisation pour renouveler le jeton d'accès lorsqu'il a expiré



# Scope

---

Le ***scope*** est un paramètre utilisé pour limiter les droits d'accès d'un client

Le serveur d'autorisation définit les *scopes* disponibles

Le client peut préciser le *scope* qu'il veut utiliser lors de l'accès au serveur d'autorisation



# Enregistrement du client

---

Le protocole ne définit pas comment l'enregistrement du client doit se faire mais définit les paramètres d'échange.

Le client doit fournir :

***Application Name***: Le nom de l'application

***Redirect URLs***: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès

***Grant Types*** : Les types d'autorisations utilisables par le client

***Javascript Origin*** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*

Le serveur répond avec :

***Client Id***:

***Client Secret***: Clé devant rester confidentielle



# *OAuth2 Grant Type*

---

Différents moyens afin que l'utilisateur donne son accord : les **grant types**

***authorization code*** :

L'utilisateur est dirigé vers le serveur d'autorisation

L'utilisateur consent sur le serveur d'autorisation

Il est redirigé vers le client avec un code d'autorisation

Le client utilise le code pour obtenir le jeton

***implicit*** : Jeton fourni directement. Certains serveurs interdisent de mode

***password*** : Le client fournit les créidentiels de l'utilisateur

***client credentials*** : Le client est l'utilisateur

***device code*** :



# Usage du jeton

---

Le jeton est passé à travers 2 moyens :

Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)

## ***L'entête d'Authorization***

GET /profile HTTP/1.1

Host: api.example.com

**Authorization: Bearer MzJmNDc3M2VjMmQzN**





## *OAuth2 et Spring Boot*



# Apport de SpringBoot

---

Si ***spring-security-oauth2*** est dans le classpath, Spring Boot effectue des auto-configurations facilitant la mise en place

- de serveurs d'autorisation  
*@EnableAuthorizedServer*
- et de serveurs de ressources  
*@EnableResourceServer*

Attention, en cours de dépréciation, voir

*<https://github.com/spring-projects/spring-security/wiki/OAuth-2.0-Migration-Guide>*



# Serveur d'autorisation

---

Avec l'annotation

**@EnableAuthorizationServer** et les propriétés ***security.oauth2.client.client-id***, ***security.oauth2.client.client-secret***

on dispose d'un serveur d'autorisation enregistrant les clients en mémoire et qui expose un endpoint (sécurisé) permettant d'obtenir un token

```
curl client:secret@localhost:8080/oauth/token -d  
grant_type=password -d username=user -d password=pwd
```



# Serveur de ressources

---

Pour utiliser le jeton d'accès, il faut un serveur de ressource ; ajouter l'annotation ***@EnableResourceServer***

- Si le serveur de ressources et le serveur d'autorisation ne font qu'un, il n'y a rien d'autre à faire car le serveur de ressource sait décoder le jeton
- Sinon, il faut indiquer une URI :
  - Soit *security.oauth2.resource.user-info-uri*
  - Soit *security.oauth2.resource.token-info-uri*



# Exemple basique

---

```
@Configuration
```

```
@EnableAuthorizationServer
```

```
@EnableResourceServer
```

```
public class Resource0AuthSecurityConfiguration  
    extends ResourceServerConfigurerAdapter{
```

```
@Override
```

```
public void configure(HttpSecurity http) throws  
    Exception {
```

```
    http.authorizeRequests()
```

```
        .antMatchers("/").permitAll()
```

```
        .antMatchers("/api/**").authenticated();
```

```
}
```

```
}
```



# Usage de JWT

---

Une autre alternative pour décoder les droits d'accès des jetons est d'utiliser *JWT* (Json Web Token)

Dans ce cas,

- le serveur d'autorisation produit des Jetons JWT
- Le serveur de ressources dispose d'une clé permettant de les déchiffrer.  
`security.oauth2.resource.jwt.key-value`



# Apports Spring Boot 2.x

---

Spring Boot 2.x a revu son support pour OAuth2. Il offre 3 autres starters :

- **OAuth2 Client** : Intégration pour utiliser un login OAuth2 fournit par Google, Github, Facebook, ...
- **OAuth2 Resource server** : Application permettant de définir des ACLs par rapport aux scopes client et aux rôles contenu dans des jetons OAuth
- **Okta** : Pour travailler avec le fournisseur OAuth Okta



# SpringBoot et les tests

---

Rappels Spring Test  
Apports de Spring Boot  
Tests auto-configurés





# Rappels *spring-test*

---

Le pattern IoC facilite les tests unitaires et les tests d'intégration en permettant d'isoler les parties à tester

Spring Test apporte pour le test unitaire

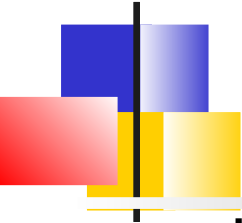
- **Mocking** de l'environnement, de JNDI et de l'API servlet
- Package **d'utilitaires** : *org.springframework.test.util*
- Spring MVC : **ModelAndViewAssert**
- **SpringRunner** : Permettant d'initialiser un contexte Spring avant l'exécution du test

Pour les tests d'intégration (impliquant un *ApplicationContext* Spring) :

- **Cache** du conteneur Spring pour accélérer les tests
- **Injection** des données de test
- Gestion de la **transaction** (roll-back)
- Des classes **utilitaires**



# Apports de Spring Boot



# *spring-boot-starter-test*

---

L'ajout de ***spring-boot-starter-test*** (dans le scope scope), ajoute les dépendances suivantes :

- *Spring Test : Utilitaires Spring pour le Test*
- ***Spring Boot Test*** : *Utilitaire liant Spring Test à Spring Boot*
- ***Spring Boot Test Autoconfigure*** : *Tests auto-configurés*
- *JUnit4, AssertJ, Hamcrest* (SB 1.x) ou *JUnit5* (SB 2.X):
- *Mockito* : Un framework pour générer des classes Mock
- *JSONassert* : Une librairie pour les assertions JSON
- *JsonPath* : XPath pour JSON.



# Annotations apportées

---

De nouvelles annotations sont disponibles via le starter :

- *@SpringBootTest* permettant de définir l'*ApplicationContext* Spring à utiliser pour un test grâce à un mécanisme de détection de configuration
- Annotations permettant des tests auto-configurés.  
Ex : Auto-configuration pour tester des *RestController* en isolation
- Annotation permettant de créer des beans Mockito



# @SpringBootTest

---

Il est possible d'utiliser l'annotation **@SpringBootTest** remplaçant la configuration standard de *spring-test* (*@ContextConfiguration*)

L'annotation crée le contexte applicatif (*ApplicationContext*) utilisé lors des tests en utilisant *SpringApplication* (classe principale)



# Attribut Class

---

L'annotation `@SpringBootTest` peut préciser les classes de configuration utilisé pour charger le contexte applicatif via l'attribut ***classes***

Exemple :

```
@SpringBootTest(classes = ForumApp.class)
```



# Attribut *WebEnvironment*

---

L'attribut *WebEnvironment* permet de préciser le type de contexte applicatif que l'on désire :

- **MOCK** : Fournit un environnement de serveur Mocké (le conteneur de servlet n'est pas démarré) : *WebApplicationContext*
- **RANDOM\_PORT** : Charge un *ServletWebServerApplicationContext*. Le conteneur est démarré sur un port aléatoire
- **DEFINED\_PORT** : Charge un *ServletWebServerApplicationContext*. Le conteneur est démarré sur un port spécifié
- **NONE** : Pas d'environnement servlet. *ApplicationContext* simple



# Détection de la configuration

---

Les annotations **@\*Test** servent comme point de départ pour la recherche de configuration.

Dans le cas de *SpringBootTest*, si l'attribut *class* n'est pas renseigné, l'algorithme cherche la première classe annotée

*@SpringBootApplication* ou *@SpringBootConfiguration* en **remontant de packages**

=> Il est donc recommandé d'utiliser la même hiérarchie de package que le code principal





# Personnalisation de la configuration

---

Si l'on veut personnaliser la configuration primaire, on peut utiliser une annotation ***@TestConfiguration*** imbriquée

Les annotations *@TestComponent* et *@TestConfiguration* permettent également de s'assurer que ces composants et configuration ne seront pas pris en compte lors du scan en conditions normales

L'utilisation de profil est une alternative plus générique



# Mocking des beans

---

L'annotation **@MockBean** définit un bean Mockito

Cela permet de remplacer ou de créer de nouveaux beans

L'annotation peut être utilisée :

- Sur les classes de test
- Sur les champs de la classe de test, dans ce cas le bean mockito est injecté

Les beans Mockito sont automatiquement réinitialisés après chaque test



# Exemple *MockBean*

---

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

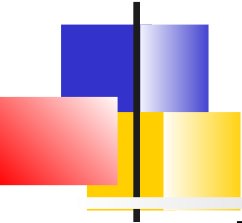
    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



# Tests auto-configurés



# Tests auto-configurés

---

Les capacités d'auto-configuration de Spring Boot peuvent ne pas être adaptées au test.

- Lorsque l'on teste la couche contrôleur, on n'a pas envie que SpringBoot nous démarre automatiquement une base de données

Le module *spring-boot-test-autoconfigure* incluent des annotations qui permettent de tester par couche les applications



# Tests JSON

---

Afin de tester si la sérialisation JSON fonctionne correctement, l'annotation **@JsonTest** peut être utilisée.

Elle configure automatiquement l'environnement *Jackson* ou *Gson*

Les classes utilitaires *JacksonTester*, *GsonTester* ou *BasicJsonTester* peuvent être injectées et utilisées, les assertions spécifiques à JSON peuvent être utilisées



# Example

---

```
@RunWith(SpringRunner.class)
@JsonTest
public class MyJsonTests {

    @Autowired
    private JacksonTester<VehicleDetails> json;

    @Test
    public void testSerialize() throws Exception {
        VehicleDetails details = new VehicleDetails("Honda", "Civic");
        // Assert against a `.json` file in the same package as the test
        assertThat(this.json.write(details)).isEqualToJson("expected.json");
        // Or use JSON path based assertions
        assertThat(this.json.write(details)).hasJsonPathStringValue("@.make");
        assertThat(this.json.write(details).extractingJsonPathStringValue("@.make")
            .isEqualTo("Honda");
    }

    @Test
    public void testDeserialize() throws Exception {
        String content = "{\"make\":\"Ford\",\"model\":\"Focus\"}";
        assertThat(this.json.parse(content))
            .isEqualTo(new VehicleDetails("Ford", "Focus"));
        assertThat(this.json.parseObject(content).getMake()).isEqualTo("Ford");
    }
}
```



# Tests de Spring MVC

---

L'annotation **@WebMvcTest** configure l'infrastructure Spring MVC et limite le scan aux annotations de Spring MVC

Elle configure également *MockMvc* qui permet de se passer d'un serveur Http complet

Pour les tests *Selenium* ou *HtmlUnit*, un client Web est également fourni





# Example

---

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyControllerTests {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        this.mvc.perform(get("/sboot/vehicle").accept(MediaType.TEXT_PLAIN))
            .andExpect(status().isOk()).andExpect(content().string("Honda
Civic"));
    }
}
```



# Example (2)

---

```
@RunWith(SpringRunner.class)
@WebMvcTest(UserVehicleController.class)
public class MyHtmlUnitTests {

    // WebClient is auto-configured thanks to HtmlUnit
    @Autowired
    private WebClient webClient;

    @MockBean
    private UserVehicleService userVehicleService;

    @Test
    public void testExample() throws Exception {
        given(this.userVehicleService.getVehicleDetails("sboot"))
            .willReturn(new VehicleDetails("Honda", "Civic"));
        HtmlPage page = this.webClient.getPage("/sboot/vehicle.html");
        assertThat(page.getBody().getTextContent()).isEqualTo("Honda Civic");
    }
}
```



# Tests JPA

---

**@DataJpaTest** configure une base de donnée mémoire, scanne les *@Entity* et configure les Repository JPA

Les tests sont transactionnels et un rollback est effectué à la fin du test

- Possibilité de changer ce comportement par *@Transactional*

Un *TestEntityManager* peut être injecté ainsi qu'un *JdbcTemplate*



# Exemple

---

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class ExampleRepositoryTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private UserRepository repository;

    @Test
    public void testExample() throws Exception {
        this.entityManager.persist(new User("sboot", "1234"));
        User user = this.repository.findByUsername("sboot");
        assertThat(user.getUsername()).isEqualTo("sboot");
        assertThat(user.getVin()).isEqualTo("1234");
    }
}
```



# Autres tests auto-configurés

---

**@WebFluxTest** : Test des contrôleurs Spring Webflux

**@JdbcTest** : Seulement la *datasource* et *jdbcTemplate*.

**@JooqTest** : Configure un *DSLContext*.

**@DataMongoTest** : Configure une base mémoire Mongo, *MongoTemplate*, scanne les classes *@Document* et configure les MongoDB repositories.

**@DataRedisTest** : Test des applications Redis applications.

**@DataLdapTest** : Serveur embarqué LDAP (if available), *LdapTemplate*, Classes *@Entry* et LDAP repositories

**@RestClientTest** : Test des clients REST. Jackson, GSON, ... + *RestTemplateBuilder*, et du support pour *MockRestServiceServer*.



# Test et sécurité

---

Spring propose plusieurs annotations pour exécuter les tests d'une application sécurisée par SpringSecurity.

```
<dependency>
<groupId>org.springframework.security</groupId>
<artifactId>spring-security-test</artifactId>
<scope>test</scope>
</dependency>
```

**@WithMockUser** : Le test est exécuté avec un utilisateur dont on peut préciser les détails (login, password, rôles)

**@WithAnonymousUser** : Annote une méthode

**@WithUserDetails("aLogin")** : Le test est exécuté avec l'utilisateur chargé par *UserDetailsService*

**@WithSecurityContext** : Qui permet de créer le SecurityContext que l'on veut



# Particularités Spring Boot

---

Monitoring des applications avec  
actuator

Déploiement

Créer une auto-configuration



# Spring Actuator





# Actuator

---

*Spring Boot Actuator* fournit un support pour la surveillance et la gestion des applications SpringBoot

Il peut s'appuyer

- Sur des points de terminaison HTTP (Si on a utilisé Spring MVC)
- Sur JMX

L'activation de Actuator nécessite  
***spring-boot-starter-actuator***



# Mise en production

---

Les fonctionnalités transverses offertes par *Actuator* concernent :

- Statut de santé de l'application
- Obtention de métriques
- Audit de sécurité
- Traces des requêtes HTTP
- Visualisation de la configuration
- ...

Elles sont accessibles via JMX ou REST



# Endpoints

---

Actuator fournit de nombreux endpoints :

- **beans** : Une liste des beans Spring
- **env / configprops** : Liste des propriétés configurables
- **health** : Etat de santé de l'appli
- **info** : Informations arbitraires. En général, Commit, version
- **metrics** : Mesures
- **mappings** : Liste des mappings configurés
- **trace** : Trace des dernières requête HTTP
- **docs** : Documentation, exemple de requêtes et réponses
- **logfile** : Contenu du fichier de traces

Si on développe un Bean de type **Endpoint**, il est automatiquement exposé via JMX ou HTTP



# Configuration

---

Les *endpoints* peuvent être configurés par des propriétés.

Chaque endpoint peut être

- Activé/désactivé
- Sécurisé par Spring Security
- Mappé sur une autre URL

Dans SB 2.x, seuls les endpoints */health* et */info* sont activés par défaut

Pour activer les autres :

- *management.endpoints.web.exposure.include=\**
- Ou les lister un par un



# Endpoint */health*

---

L'information fournie permet de déterminer le statut d'une application en production.

- Elle peut être utilisée par des outils de surveillance responsable d'alerter lorsque le système tombe (Kubernetes par exemple)

Par défaut, le endpoint affiche un statut global mais on peut configurer Spring pour que chaque sous-système (beans de type *HealthIndicator*) affiche son statut :

```
management.endpoint.health.show-details= always
```

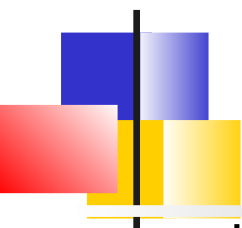


# Indicateurs fournis

---

Spring fournit les indicateurs de santé suivants lorsqu'ils sont appropriés :

- ***CassandraHealthIndicator*** : Base Cassandra est up.
- ***DiskSpaceHealthIndicator*** : Vérifie l'espace disque disponible .
- ***DataSourceHealthIndicator*** : Connexion à une source de données
- ***ElasticsearchHealthIndicator*** : Cluster Elasticsearch up.
- ***JmsHealthIndicator*** : JMS broker up.
- ***MailHealthIndicator*** : Serveur de mail up.
- ***MongoHealthIndicator*** : BD Mongo up.
- ***RabbitHealthIndicator*** : Serveur Rabbit up
- ***RedisHealthIndicator*** : Serveur Redis up.
- ***SolrHealthIndicator*** : Serveur Solr up
- ...



# Information sur l'application

---

Le *endpoint* **/info** par défaut n'affiche rien.

Si l'on veut les détails sur Git :

```
<dependency>
  <groupId>pl.project13.maven</groupId>
  <artifactId>git-commit-id-plugin</artifactId>
</dependency>
```

Si l'on veut les informations de build :

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>build-info</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



# Metriques

---

Le endpoint ***metrics*** donne accès à toute sorte de métriques. On retrouve :

- Système : Mémoire, Heap, Threads, GC
- Source de données : Connexions actives, état du pool
- Cache : Taille, Hit et Miss Ratios
- Tomcat Sessions





# Endpoints de SpringBoot 2

---

**/auditevents** : Liste les événements de sécurité (login/logout)

**/conditions** : Remplace */autoconfig*, rapport sur l'auto-configuration

**/configprops** – Les beans annotés par *@ConfigurationProperties*

**/flyway** ; Information sur les migrations de BD Flyway

**/liquibase** : Migration Liquibase

**/logfile** : Logs applicatifs

**/loggers** : Permet de visualiser et modifier le niveau de log

**/scheduledtasks** : Tâches programmées

**/sessions** : HTTP sessions

**/threaddump** : Thread dumps



# Déploiement



# Introduction

---

Plusieurs alternatives pour déployer une application Spring-boot :

- Application stand-alone
- Archive war à déployer sur serveur applicatif
- Service Linux ou Windows
- Image Docker
- Le cloud



# Application stand-alone

---

Le plugin Maven de Spring-boot permet de générer l'application stand-alone :

```
mvn package
```

Crée une archive exécutable contenant les classes applicatives et les dépendances dans le répertoire *target*

Pour l'exécuter :

```
java -jar target/artifactId-version.jar
```



# Fichier Manifest

---

Manifest-Version: 1.0

Implementation-Title: documentService

Implementation-Version: 0.0.1-SNAPSHOT

Archiver-Version: Plexus Archiver

Built-By: dthibau

**Start-Class: org.formation.microservice.documentService.DocumentsServer**

Implementation-Vendor-Id: org.formation.microservice

Spring-Boot-Version: 1.3.5.RELEASE

Created-By: Apache Maven 3.3.9

Build-Jdk: 1.8.0\_121

Implementation-Vendor: Pivotal Software, Inc.

**Main-Class: org.springframework.boot.loader.JarLauncher**



# Création de war

---

Pour créer un war, il est nécessaire de :

- Fournir une sous-classe de **SpringBootServletInitializer** et surcharger la méthode *configure()*. Cela permet de configurer l'application (Spring Beans) lorsque le war est installé par le servlet container.
- De changer l'élément packaging du *pom.xml* en war  
`<packaging>war</packaging>`
- Puis exclure les librairies de tomcat  
Par exemple en précisant que la dépendance sur le starter Tomcat est fournie

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-tomcat</artifactId>  
  <scope>provided</scope>  
</dependency>
```



# Exemple

---

```
@SpringBootApplication
public class Application extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder application) {
        return application.sources(Application.class);
    }
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```



# Création de service Linux

---

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <executable>true</executable>
      </configuration>
    </plugin>
  </plugins>
</build>
```

=> target/artifactId.jar is executable !

=> ln -s target/artifactId.jar /etc/init.d/artifact  
service artifact start





# Cloud

---

Les jars exécutable de Spring Boot sont prêts à être déployés sur la plupart des plate-formes PaaS

La documentation de référence offre du support pour :

- Cloud Foundry
- Heroku
- OpenShift
- Amazon Web Services
- Google App Engine



# Exemple CloudFoundry/Heroku

---

## Cloud Foundry

```
cf login
```

```
cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar
```

## Heroku

Mise à jour d'un fichier Procfile :

```
web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar
```

```
git push heroku master
```



# Annexes



## Spring Data Rest : Ressources URI



# Fonctionnalités de Spring Data Rest

---

- Permet de découvrir automatiquement l'API du domaine en utilisant HAL.
- Expose des ressources « collection », « détail » et association représentant le modèle.
- Supporte la pagination avec des liens de navigation
- Permet de filtrer dynamiquement les ressources collection
- Expose des ressources dédiées aux méthodes de requêtage définies dans les repositories.
- Permet d'ajouter des hooks sur le traitement des requêtes REST (Spring ApplicationEvents).
- Expose des méta-données sur le modèle
- Permet de définir des représentations spécifiques via les projections.
- Fournit un navigateur HAL
- Supporte JPA, MongoDB, Neo4j, Solr, Cassandra, Gemfire.



# Code statut par défaut

---

Code statut par défaut :

- **200 OK** pour les requêtes GET
- **201 Created** pour les requêtes POST et PUT qui crée de nouvelles ressources
- **204 No Content** pour PUT, PATCH, et DELETE si la configuration spécifie qu'aucun corps de requête n'est retourné en cas de mise à jour(*RepositoryRestConfiguration.returnBodyOnUpdate*).
- **200 OK** pour PUT provoquant une mise à jour



# Ressource collection

---

Le nom de la ressource et son chemin peuvent être personnalisée via l'annotation

**@RepositoryRestResource** sur l'interface Repository

- **GET** : Toutes les entités du repository grâce à sa méthode *findAll(...)*. Paramètres :
  - page, size : Si le repo supporte la pagination.
  - sort\* : Critères de tri
- **HEAD** : Indique si la collection est disponible
- **POST** : Crée une nouvelle entité avec le corps de requête => Appel de la méthode *save()*



# Ressource simple (item)

---

- **GET** : Retourne une entité => méthode *findOne()*
- **HEAD** : Indique si l'entité est disponible.
- **PUT** : Remplace l'item avec le corps de requête.
- **PATCH** : Équivalent à PUT mais pour les mises à jour partielle.
- **DELETE** : Suppression de la ressource





# Ressource « association »

---

Spring Data REST expose des sous-ressources pour chaque association qu'une entité a.

Le nom et le chemin reprennent par défaut le nom de la propriété d'association et peuvent être personnalisé par *@RestResource*.

- **GET** : Retourne le statut de la ressource association
- **PUT** : Associe la ressource référencée par l' URI(s)
- **POST** : Ajoute un nouvel élément (seulement pour les associations collection).
- **DELETE** : Supprime l'association.



# Ressource recherche

---

La ressource **recherche** (Par défaut /search) retourne de liens pour toutes les méthodes de requête du repository.

Le chemin et le nom de la méthode peuvent être modifiés via l'annotation *@RestResource*

- **GET** : Retourne une liste de liens pour accéder aux différentes méthodes
- **HEAD** : Indique si la ressource recherche est disponible.

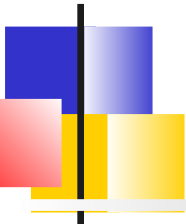


# Ressource méthode de requête

---

La ressource associée à une méthode de requête supporte 2 actions HTTP :

- **GET** : Retourne le résultat de la requête  
Paramètres :
  - page : (défaut 0).
  - size : (défaut 20).
  - sort : (\$propertyname,)+[asc|desc]?.
- **HEAD** : Indique si la ressource est disponible.



# Personnalisation du mapping

---

Il est quelquefois nécessaire de personnaliser le mapping effectué par Data REST

Dans ce cas, il suffit de configurer les *ObjectMapper* Jackson avec ses propres (dé)sérialiseurs.

- Soit programmatiquement
- Soit avec l'annotation *@JsonComponent* de SpringBoot



# Utilisation des entêtes

---

L'entête ***ETag*** permet de tagger les ressources avec le champ annoté par *@Version*

- Cela permet d'éviter des mises à jour concurrentes.

*If-Match* et *If-None-Match* permettent de faire des requêtes conditionnelles

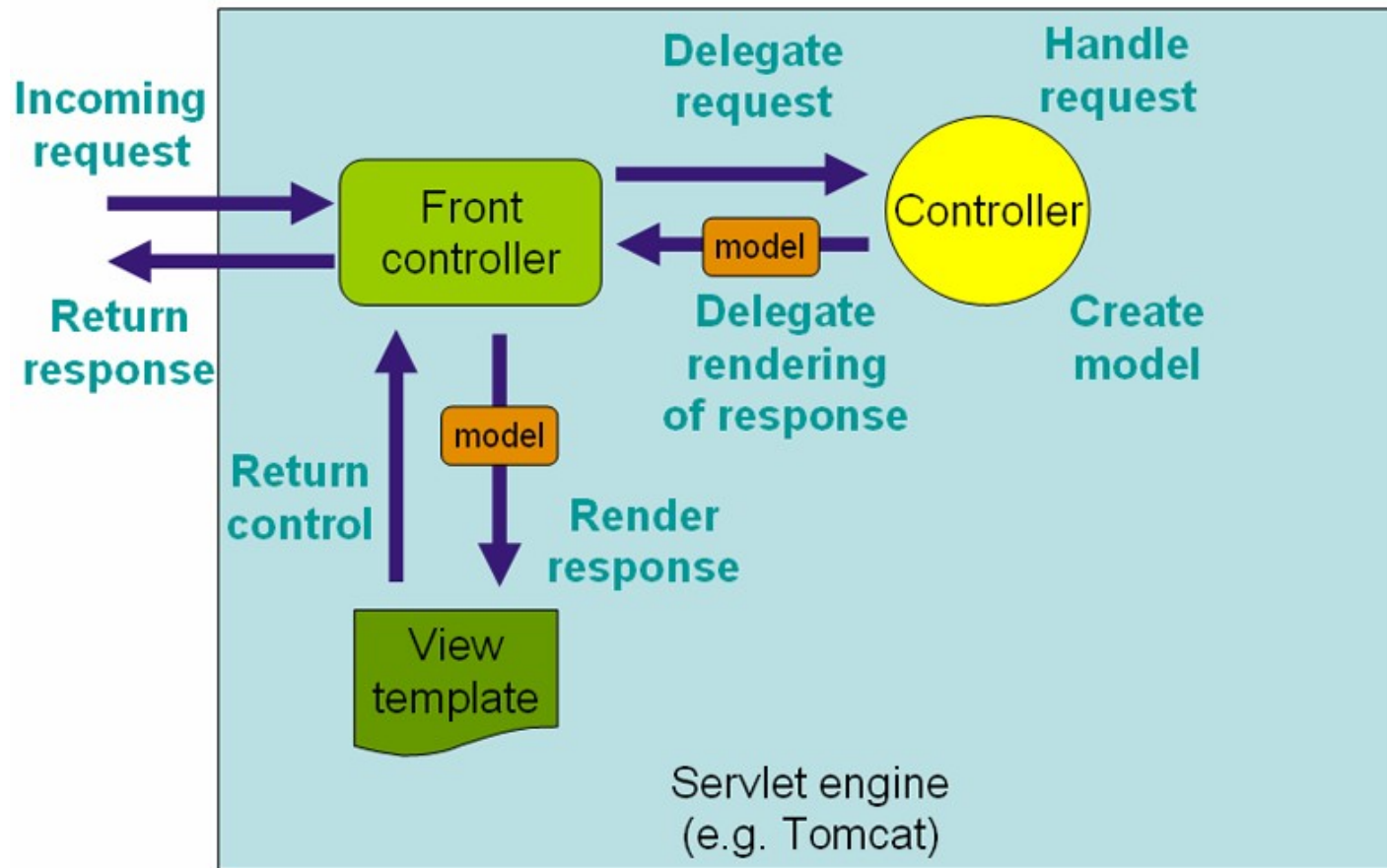
```
curl -v -X PATCH -H 'If-Match: <value of previous ETag>' ...
```

L'entête *If-Modified-Since* combinée à l'annotation *@LastModifiedDate* permet de vérifier si la ressource a changé depuis la dernière requête.



# Spring Boot et modèle MVC

# MVC





# Exemple Web Controller

---

**@Controller**

**@RequestMapping("/web")**

```
public class MembersController {  
    @Autowired  
    protected MemberRepository memberRepository;
```

**@GetMapping(path = "/register")**

```
public String registerForm(Model model) {  
    model.addAttribute("user", new User());  
    return "register";  
}
```

**@RequestMapping(path = "/register", method = RequestMethod.POST)**

```
public String register(@Valid @RequestBody Member member) {  
    member = memberRepository.save(member);  
    return "home" ;  
}  
}
```





# Autres fonctionnalités

---

D'autres fonctionnalités sont apportées par le framework :

- La localisation, le fuseau horaire
- La résolution des thèmes
- Support pour le téléchargement de fichiers
- ...



# Beans spécifiques à Spring MVC

---

***HandlerMapping*** : Associe les requêtes entrantes aux gestionnaires et définit une liste d'intercepteurs qui effectue des pré et post traitements sur la requête. L'implémentation la plus courante est *@Controller*

***HandlerExceptionResolver*** : Associe les exceptions à des gestionnaires d'exception.

***ViewResolver*** : Résout à partir d'un *outcome* la vue à rendre

***LocaleResolver*** & ***LocaleContextResolver*** : Détermine la locale du client

***ThemeResolver*** : Résout le thème à utiliser pour l'application (layout et style)

***MultipartResolver*** : Traite les requêtes multi-part pour le chargement de fichier par exemple.

***FlashMapManager*** : Permet de transférer des attributs d'une requête à une autre habituellement via une redirection d'URL



# Formats d'entrée/sorties

---

Il est également de spécifier une liste de type de média permettant de filtrer sur l'entête *Content-type* de la requête HTTP

En entrée, précise le format attendu

```
@PostMapping(path = "/pets", consumes = "application/json")  
public void addPet(@RequestBody Pet pet, Model model) {
```

Ou en sortie, précise le format généré :

```
@GetMapping(path = "/pets/{petId}",  
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE)  
@ResponseBody  
public Pet getPet(@PathVariable String petId, Model model) {
```



# @RequestBody et convertisseur

---

L'annotation **@RequestBody** utilise des *HTTPMessageConverter* qui se basent sur l'entête *content-type* de la requête

- *StringHttpMessageConverter*
- *FormHttpMessageConverter*  
(*MultiValueMap<String, String>*)
- *ByteArrayHttpMessageConverter*
- *MappingJackson2HttpMessageConverter* : JSON
- *MappingJackson2XmlHttpMessageConverter* : XML
- ...



# *@ResponseBody*

---

**@ResponseBody** est similaire à  
*@RequestBody*

Il utilise des convertisseurs qui alimentent la  
réponse HTTP

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



# Validation de formulaire

---

La validation de formulaire s'appuie sur les contraintes ***javax.validation***

Il suffit d'annoter un argument d'entrée avec ***@Valid*** afin que SpringMVC effectue la validation et positionne les erreurs dans un objet ***BindingResult***

C'est alors la responsabilité du contrôleur de tester la présence d'erreur de validation

Attention : l'argument *BindingResult* doit précéder l'argument du *Model*



# Exemple

---

```
@PostMapping(path = "/authenticate")
// Spring MVC ajoute la variable user dans le model !!
public String authenticate(@Valid User aUser,
    BindingResult result, Model model) {

    if ( result.hasErrors() ) {
        return "login";
    }
}
```



# Thymeleaf

---

**Thymeleaf** est la technologie de vue par défaut de SpringBoot

Elle permet de construire des pages HTML-5

Les vues sont constituées :

- De balises HTML-5
- D'attributs thymeleaf *data-th-* (ou en raccourci *th-*)
- D'expressions qui produisent les données dynamiques





# Attributs

---

***th:text*** : Permet l'externalisation (et la localisation des textes de la vue)

***th:object*** : Sélection d'une variable de contexte

***th:errors*** : Accès aux messages d'erreur

***th:href, th:attr, th:action*** : Permet de résoudre des URLs, des attributs de balise, des actions de formulaire

***th:insert, th:replace, th:with*** : Inclusion de fragments

***th:if, th-each*** : Test, boucles

***th:span, th:div, th:class, ...*** : Balises et CSS

***th:on\**** : Événements Javascript

....



# Expressions Thymeleaf

---

- `${...}`** : Variables contenues dans le contexte
- `*{...}`** : Sélection sur l'objet sélectionné  
th:object
- `#{...}`** : Messages localisés
- `@{...}`** : URL
- `~{...}`** : Fragment de balises `<=>`  
inclusion



# Exemple vue ThymeLeaf

---

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"
      xmlns:sec="http://www.thymeleaf.org/thymeleaf-extras-springsecurity3">
  <head><title>Registration Form </title></head>
  <body>
    <form th:action="@{/web/register}" th:object="${user}" method="post">
      <div><label> First name : <input type="text" th:field="*{firstName}"/> </label></div>
      <div><label> Last name: <input type="text" th:field="*{lastName}"/> </label></div>
      <div><input type="submit" value="Sign In"/></div>
      <div><label> Email : <input type="text" th:field="*{email}"/> </label></div>
      <div><label> Password: <input type="password" th:field="*{password}"/> </label></div>
      <div><input type="submit" value="Sign In"/></div>
    </form>
  </body>
</html>
```



# Spring Boot et Spring MVC



# Auto-configuration

---

*SpringBoot* effectue des configurations automatiques pour Spring MVC. Les principaux apports sont :

- Démarrage automatique des serveur embarqué
- Configuration par défaut pour servir des ressources statiques (index.html, favicon, Webjars)
- Détection et configuration automatique du langage de templating
- Configuration automatique des *HttpMessageConverters* permettant un comportement par défaut des sérialiseurs



# Personnalisation de la configuration

---

- Ajouter un bean de type ***WebMvcConfigurerAdapter*** (ou étendre ***WebMvcConfigurer***) et implémenter les méthodes voulues :
  - Configuration MVC (ViewResolver, ViewControllers), de gestionnaire d'exception ou d'intercepteurs, du Cors, ..
- Si l'on veut personnaliser des beans coeur de SpringMVC, déclarer un bean de type ***WebMvcRegistrationsAdapter*** (SB 1.x) ou ***WebMvcRegistrations*** (SB 2.x) et enregistrer ses propres gestionnaires de mappings ou solveurs d'exceptions



# Exemple : Définition de *ViewController*

---

```
@Configuration
```

```
public class MvcConfig extends WebMvcConfigurer {
```

```
    @Override
```

```
    public void addViewControllers(ViewControllerRegistry registry) {  
        registry.addViewController("/home").setViewName("home");  
        registry.addViewController("/").setViewName("home");  
        registry.addViewController("/hello").setViewName("hello");  
        registry.addViewController("/login").setViewName("login");  
    }
```

```
}
```



# Exemple *Intercepteurs*

---

```
@SpringBootApplication
public class MyApplication {

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

    @Bean
    public WebMvcConfigurer adapter() {
        return new WebMvcConfigurer() {
            @Override
            public void addInterceptors(InterceptorRegistry registry) {
                System.out.println("Adding interceptors");
                registry.addInterceptor(new MyInterceptor()).addPathPatterns("/**");

                super.addInterceptors(registry);
            }
        };
    }
}
```





# Convertisseur de messages HTTP

---

*SpringBoot* fournit les convertisseurs par défaut pour traiter les données JSON, XML, String en UTF-8

On peut ajouter des propres convertisseurs traitant un **type de média particulier**.

```
@Configuration
public class MyConfiguration {

    @Bean
    public HttpMessageConverters customConverters() {
        HttpMessageConverter<?> additional = ...
        HttpMessageConverter<?> another = ...
        return new HttpMessageConverters(additional, another);
    }
}
```



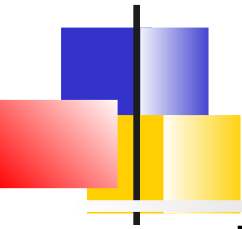
# Contenu statique

---

Par défaut, *SpringBoot* sert du contenu statique à partir du répertoire ***/static*** (ou */public* ou */resources* ou */META-INF/resources*) dans le classpath

- Il utilise alors *ResourceHttpRequestHandler* provenant de *SpringMVC*

Les emplacements peuvent être modifiés par la propriété :  
`spring.resources.staticLocations`



# Webjar

---

Les librairies clientes (ex : *jQuery*, *bootstrap*, ...) peuvent être packagées dans des jars : les ***webjars***

- Les webjars permettent une gestion des dépendances par Maven,

Spring est capable de servir des ressources Webjars présentes dans une archive située dans */webjars/\*\**



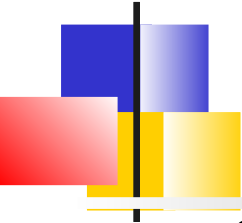
# Exemple

---

```
<dependency>  
<groupId>org.webjars</groupId>  
<artifactId>bootstrap</artifactId>  
<version>3.3.7-1</version></dependency>
```

...

```
href = /webjars/bootstrap/3.3.7-1/css/bootstrap.min.css
```



# Technologies de vue et templating

---

Spring MVC peut générer du html dynamique en utilisant une technologie basique de templating.

Spring Boot permet l'auto-configuration de

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

Les gabarits sont alors pris de l'emplacement  
***src/main/resources/templates***



# Gestion des erreurs

---

*Spring Boot* associe **/error** à la page d'erreur globale de l'application

- Un comportement par défaut en REST ou en Web permet de visualiser la cause de l'erreur

Pour remplacer le comportement par défaut :

- Implémenter **ErrorController** et l'enregistrer comme Bean
- Ajouter un bean de type **ErrorAttributes** qui remplace le contenu de la page d'erreur
- Ajouter une classe annotée par **@ControllerAdvice** pour personnaliser le contenu renvoyé



# Exemple *@ControllerAdvice*

## **@ControllerAdvice**

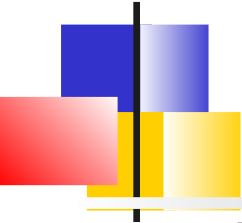
```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {
```

```
    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFound.class})
```

```
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }
```

## **@Override**

```
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```



# Page d'erreur

---

Pour remplacer la vue par défaut de la page `/error`, il suffit de construire une vue *error*.

- Typiquement, si on utilise Thymeleaf, écrire une vue *error.html* dans le répertoire *template*





# Pages d'erreur personnalisées

Si l'on veut afficher des pages d'erreur selon le code retour HTTP. Il suffit d'ajouter des pages statiques ou dynamiques dans le répertoire */error*

src/

+ - main/

+ - java/

| + <source code>

+ - resources/

+ - public/

+ - error/

| + - 404.html

| + - 5xx.ftl



# Configuration

---

```
@Configuration
@Profile("swagger")
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())

            .build();
    }
}
```



# Conteneur de servlets



# Choix du conteneur

---

Le choix du conteneur par Spring Boot dépend des dépendances configurées.

Pour utiliser un conteneur autre que Tomcat, il suffit de déclarer la dépendance et exclure celle de Tomcat

Par exemple, pour *undertow* :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions><exclusion>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
  </exclusion></exclusions>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```



# Servlets, Filtres et Listener

---

Spring Boot embarque un conteneur de servlet (Tomcat, Jetty ou Undertow) écoutant par défaut sur le port 8080

Il est donc possible d'enregistrer des servlets, des filtres ou des listener en utilisant

- des beans implémentant *Servlet*, *Filter* ou *\*Listener*
- ou en scannant des composants Servlets
  - *@ServletComponentScan*
  - + *@WebServlet*, *@WebFilter* *@WebListener*



# Mapping des servlets

---

Les beans de type *Servlet* ou autres peuvent utiliser les propriétés définies dans *application.properties*

Par défaut

- si l'application ne contient qu'un seul Servlet, il est mappé sur /
- Sinon, le nom du bean est utilisé comme préfixe
- Les filtres sont mappés sur /\*

Si cela n'est pas le comportement voulu, il faut utiliser les beans *ServletRegistrationBean*, *FilterRegistrationBean* ou *ServletListenerRegistrationBean*



# Initialisation de contexte

---

Les serveurs embarqués n'exécutent pas les interfaces

`javax.servlet.ServletContainerInitializer` OU  
`org.springframework.web.WebApplicationInitializer`

Pour effectuer des initialisations de contexte, il faut enregistrer un bean implémentant :

`org.springframework.boot.context.embedded.ServletContextInitializer`

- La méthode unique *onStartup* permet d'accéder au *ServletContext*



# Configuration du conteneur

---

Typiquement cela s'effectue dans  
*application.properties*

- Réseau : *server.port, server.address*
- Session : *server.session.persistence, server.session.timeout, server.session.store-dir, server.session.cookie.\**.
- Gestion des erreurs : *server.error.path*
- SSL
- Compression HTTP





# Spring WebFlux



# Programmation réactive

---

Spring se met à la programmation réactive avec **WebFlux** dans sa version 5.

Concrètement, il s'agit de pouvoir implémenter une application web (contrôleur REST) ou des clients HTTP de manière réactive.

Pour ce faire, Spring 5 intègre désormais **Reactor** et permet de manipuler les objets **Mono** (1 objet) et **Flux** (N objet(s)).



# Motivation *Webflux*

---

2 principales motivations pour Spring Webflux :

- Le besoin d'un stack non-bloquante permettant de gérer la concurrence avec peu de threads et de scaler avec moins de ressources CPU/mémoire
- La programmation fonctionnelle



# Introduction

---

Le module *spring-web* est la base pour Spring Webflux.

Il offre en plus 2 modèles de programmation :

- **Contrôleurs annotés** : Idem à Spring MVC avec les mêmes annotations.  
Les méthodes des contrôleurs peuvent retourner des types réactifs, des arguments réactifs sont associés à *@RequestBody*.
- **Endpoints fonctionnels** : Programmation fonctionnelle basée sur les lambdas.  
Idéal pour de petites libraires permettant de router et traiter des requêtes.  
Dans ce cas, l'application est en charge du traitement de la requête du début à la fin.



# Contrôleurs annotés

---

Les annotations **@Controller** de Spring MVC sont donc supportés par *WebFlux*.

Les différences sont :

- Les beans cœur comme *HandlerMapping* ou *HandlerAdapter* sont non bloquants et travaillent sur les classes réactives
- ***ServerHttpRequest*** et ***ServerHttpResponse*** plutôt que *HttpServletRequest* et *HttpServletResponse*.



# Exemple

---

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }
    @PostMapping("/person")
    Mono<Void> create(@RequestBody Publisher<Person> personStream) {
        return this.repository.save(personStream).then();
    }
    @GetMapping("/person")
    Flux<Person> list() {
        return this.repository.findAll();
    }
    @GetMapping("/person/{id}")
    Mono<Person> findById(@PathVariable String id) {
        return this.repository.findOne(id);
    }
}
```



# Méthodes des contrôleurs

---

Les méthodes des contrôleurs ressemblent à ceux de Spring MVC (Annotations, arguments et valeur de retour possibles), à quelques exception près

– Arguments :

- ***ServerWebExchange*** : Encapsule, requête, réponse, session, attributs
- ***ServerHttpRequest*** et ***ServerHttpResponse***

– Valeurs de retour :

- ***Observable<ServerSentEvent>*** : Données + Méta-données
- ***Observable<T>*** : Données seules

– Request Mapping (consume/produce) : *text/event-stream*



# *Endpoints* fonctionnels

---

Dans ce modèle de programmation fonctionnelle, les fonctions (lambda-expression) sont utilisées pour :

- Router : ***RouterFunction***
- et traiter les requêtes :  
***HandlerFunction***





# Traitement des requêtes via *HandlerFunction*

---

Les fonctions de type ***HandlerFunction*** prennent en entrée un *ServerRequest* et fournissent un *Mono<ServerResponse>*

Exemple :

```
HandlerFunction<ServerResponse> helloWorld =  
    request -> ServerResponse.ok().body(fromObject("Hello  
World"));
```

Généralement, les fonctions similaires sont regroupées dans une classe *contrôleur*.



# Example

---

```
public class PersonHandler {  
    private final PersonRepository repository;  
  
    public PersonHandler(PersonRepository repository) { this.repository = repository;}  
  
    public Mono<ServerResponse> listPeople(ServerRequest request) {  
        Flux<Person> people = repository.allPeople();  
        return ServerResponse.ok().contentType(APPLICATION_JSON).body(people, Person.class);  
    }  
  
    public Mono<ServerResponse> createPerson(ServerRequest request) {  
        Mono<Person> person = request.bodyToMono(Person.class);  
        return ServerResponse.ok().build(repository.savePerson(person));  
    }  
  
    public Mono<ServerResponse> getPerson(ServerRequest request) {  
        int personId = Integer.valueOf(request.pathVariable("id"));  
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();  
        Mono<Person> personMono = this.repository.getPerson(personId);  
        return personMono  
            .then(person -> ServerResponse.ok().contentType(APPLICATION_JSON).body(fromObject(person)))  
            .otherwiseIfEmpty(notFound);  
    }  
}
```



# Mapping via *RouterFunction*

Les requêtes sont routées vers les *HandlerFunction* avec une ***RouterFunction*** :

Prend en entrée un *ServerRequest* et retourne un *Mono<HandlerFunction>*

- Les fonctions ne sont en général pas écrites directement. On utilise :  
***RouterFunctions.route(RequestPredicate, HandlerFunction)***  
permettant de spécifier les règles de matching

Exemple :

```
RouterFunction<ServerResponse> helloWorldRoute =  
RouterFunctions.route(RequestPredicates.path("/hello-world"),  
request -> Response.ok().body(fromObject("Hello World")));
```



# Combinaison

---

2 fonctions de routage peuvent être composées en une nouvelle fonction via les méthodes

```
RouterFunction.and(RouterFunction)
```

```
RouterFunction.andRoute(RequestPredicate,  
    HandlerFunction)
```

Si la première règle ne matche pas, la seconde est évaluée ... et ainsi de suite



# Example

---

```
PersonRepository repository = ...
```

```
PersonHandler handler = new PersonHandler(repository);
```

```
RouterFunction<ServerResponse> personRoute = RouterFunctions.
```

```
  route(RequestPredicates.GET("/person/{id}"))
```

```
    .and(accept(APPLICATION_JSON)), handler::getPerson)
```

```
  .andRoute(RequestPredicates.GET("/person"))
```

```
    .and(accept(APPLICATION_JSON)), handler::listPeople)
```

```
  .andRoute(RequestPredicates.POST("/person"))
```

```
    .and(contentType(APPLICATION_JSON)), handler::createPerson);
```



# Configuration WebFlux

---

## @Configuration

```
public class WebConfig implements WebFluxConfigurer {  
    @Bean  
    public RouterFunction<?> routerFunctionA() { // ... }  
    @Bean  
    public RouterFunction<?> routerFunctionB() { // ... }  
    @Override  
    public void configureHttpMessageCodecs(ServerCodecConfigurer configurator) {  
        // configure message conversion...  
    }  
    @Override  
    public void addCorsMappings(CorsRegistry registry) {  
        // configure CORS...  
    }  
    @Override  
    public void configureViewResolvers(ViewResolverRegistry registry) {  
        // configure view resolution for HTML rendering...  
    }  
}
```



---

Créer une auto-configuration



# Introduction

---

Le mécanisme d'auto-configuration de Spring Boot est implémenté à base :

- De classe **@Configuration** classiques
- Et des annotations **@Conditional** qui précisent les conditions pour que la configuration s'active

Le packaging consiste à fournir :

- Le code d'auto-configuration
- Les dépendances vers les librairies utilisées

Il est donc assez simple de mettre en place ses propres configurations automatiques





# Liste des auto-configuration

---

Dans le module d'auto-configuration, le fichier ***META-INF/spring.factories*** permet de lister les configurations automatiques susceptibles d'être activées

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\ncom.mycom.autoconfigure.LibConfiguration
```



# Annotations conditionnelles

---

Des annotations conditionnelles permettent de préciser les conditions d'activation de la configuration.

Les conditions peuvent se basées sur :

- La présence ou l'absence d'une classe :  
**@ConditionalOnClass** et **@ConditionalOnMissingClass**
- La présence ou l'absence d'un bean :  
**@ConditionalOnBean** ou **@ConditionalOnMissingBean**
- Une propriété :  
**@ConditionalOnProperty**
- La présence d'une ressource :  
**@ConditionalOnResource**
- Le fait que l'application est une application Web ou pas :  
**@ConditionalOnWebApplication** ou  
**@ConditionalOnNotWebApplication**
- Une expression SpEL



# Exemple SolR

---

```
@Configuration
@ConditionalOnClass({ HttpSolrClient.class, CloudSolrClient.class })
@EnableConfigurationProperties(SolrProperties.class)
public class SolrAutoConfiguration {

    private final SolrProperties properties;

    private SolrClient solrClient;

    public SolrAutoConfiguration(SolrProperties properties) {
        this.properties = properties;
    }

    @Bean
    @ConditionalOnMissingBean
    public SolrClient solrClient() {
        this.solrClient = createSolrClient();
        return this.solrClient;
    }

    private SolrClient createSolrClient() {
        if (StringUtils.hasText(this.properties.getZkHost())) {
            return new CloudSolrClient(this.properties.getZkHost());
        }
        return new HttpSolrClient(this.properties.getHost());
    }
}
```



# Conséquences

---

Le starter *Solar* tire

- Les classes de Configuration conditionnelles
- Les librairies Sonar

Les beans d'intégration de Sonar sont créés et peuvent être injectés dans le code applicatif.

```
@Component
public class MyBean {

    private SolrClient solrClient;

    public MyBean(SolrClient solrClient) {
        this.solrClient = solrClient;
    }
}
```



# Séquence des auto-configuration

---

L'ordre d'activation des auto-configuration peut être contrôlé par les annotations : ***@AutoConfigureAfter***, ***@AutoConfigureBefore***



# starter

---

Un starter Spring pour une librairie contient :

- Le module ***autoconfigure*** contenant le code d'auto-configuration
- Le module ***starter*** qui fournit la dépendance vers le module *autoconfigure* et les autres dépendances