



# Spring Boot et Angular

---

David THIBAU – 2021

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



# Agenda

---

- **Introduction**
  - Rappels Spring
  - Offre SpringBoot
  - Propriétés de configuration
- **API Rest avec SpringBoot**
  - Rappels Spring Core / Spring Boot
  - Surcharges des propriétés de configuration, Profils
  - Structure projet, API Rest
- **Persistance**
  - Alternatives pour la persistance
  - Repositories et SpringData
  - JPA, JDBC, OpenSession in View
- **API Rest**
  - Annotations contrôleurs
  - Sécurisation couche Web
- **SpringBoot Test**
  - Apports de Spring Boot
  - Tests auto-configurés
- **Démarrage Angular**
  - Rappels Javascript/ TypeScript/RxJx
  - Modèles de composants d'Angular, Modularité
  - Démarrage de projet avec Angular CLI
- **Les composants**
  - Binding
  - Syntaxe des templates
  - Pipes
- **Les services**
  - Injection de dépendances
  - Service
- **Routing et Formulaires**
- **Échanges HTTP**
- **Tests**
- **Déploiement**



# Introduction

---

## **Rappels Spring**

Offre Spring Boot

Propriétés de configuration, profils



# 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



# Conteneur léger

---

Spring est un conteneur léger qui applique le pattern IoC

- Le code applicatif ne dépend que des interfaces
- Les configurations fixent les implémentations

Au démarrage, Spring fabrique les objets, injecte les dépendances et les propriétés valeurs.

- La plupart des beans sont des singletons

Spring prend en charge les services techniques requis par les beans :

- Intégration à des systèmes tiers, transaction, sécurité, monitoring



# Types de configuration

---

- ❖ Différents choix sont possibles pour configurer le container :
  - **Fichier de configuration XML :**
    - Projet legacy, Configuration transverse
  - **Source Java :**
    - Classes de configuration
    - Annotations dans les beans
  - **Spring Boot :**
    - Mécanisme d'auto-configuration



# 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

---

**@Service**

```
public class MovieLister {  
    // MovieFinder est une interface  
    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

---

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

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

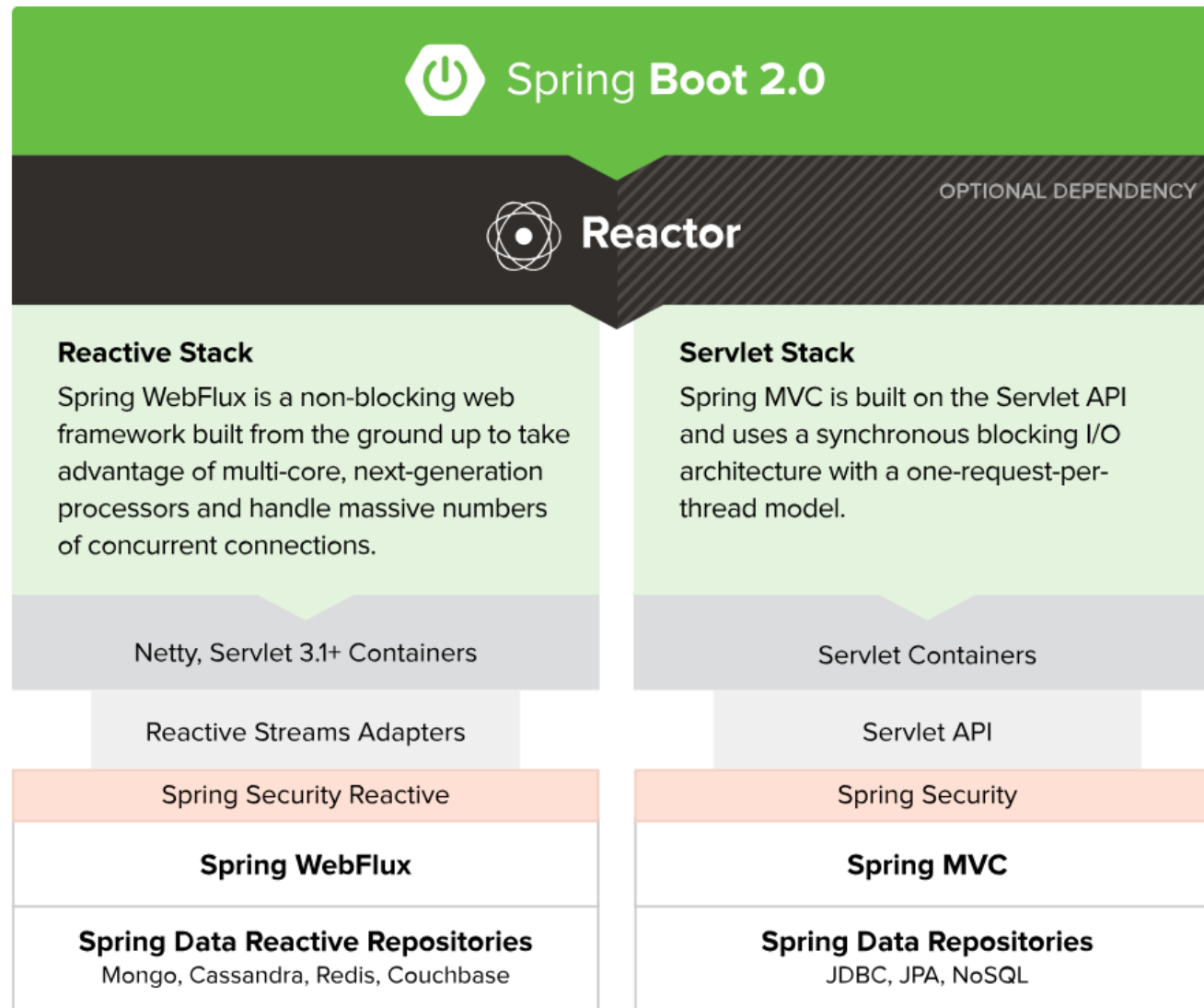
    public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = finder.findAll();
        List<Movie> ret = new ArrayList<Movie>() ;
        for (Movie movie : allMovies ) {
            if (!movie.getDirector().equals(arg))
                ret.add(movie);
        }
        return ret;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



# 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 5.x et SB 2.0





# 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

---

Pivotal fournit des plugins/extensions pour Eclipse, *VSCode* ou *Theia*

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

*Pour les autres IDEs, des fonctionnalités similaires à STS sont fournis par des plugins tierces*



# Introduction

---

Rappels Spring  
**Offre Spring Boot**  
Propriétés de configuration, profils



# 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 organisées en starter-modules, exploitées par un système de build et de gestion de dépendances ( ***Maven*** ou ***Gradle*** )
- Du code permettant d'appliquer automatiquement des configurations par défaut de beans en fonction des conditions d'exécution





# Auto-configuration

---

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

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

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

Dans un environnement Java, il repose principalement sur la présence ou l'absence de certaines classe dans le *classpath*



# Spring Initializr

---

**Spring Initializr** est un service web proposant un assistant de création de projet. (Il est intégré dans les IDEs)

Après avoir renseigné :

- Les coordonnées du projet, le package racine
- Les versions de SpringBoot et de Java. L'outil de build
- Les starters modules que l'on veut utiliser

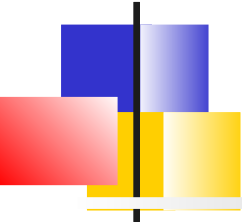
Il génère l'arborescence projet, le fichier de build (*pom.xml* ou *build.gradle*), la classe principale de l'application et une classe de test



# Arborescence projet

---

- my-project [boot]
  - src/main/java
    - org.formation
      - MyProjectApplication.java
  - src/main/resources
    - application.properties
  - src/test/java
    - org.formation
      - MyProjectApplicationTests.java
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml



# Exemple Maven

---

```
<!-- Héritage Spring Boot, fixe les librairies et leurs versions -->
<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>
<!-- Plugin SpringBoot : fatJar, buildInfo, buildImage, run -->
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



# Starter Modules

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

***spring-boot-starter-reactive-web***: librairies Spring WebFlux + configuration automatique d'un serveur embarqué (Netty).

***spring-boot-starter-data-\**** : Librairies d'accès aux données persistantes (JPA, NoSQL, SolR, ...). Facilite la configuration des sources de données et l'implémentation de la couche DAO

***spring-boot-starter-security*** : librairies de SpringSecurity + configuration simpliste de la sécurité

***spring-boot-starter-actuator*** : Permet l'exposition de points de surveillance via HTTP ou JMX (métriques de performances, sondes, audit sécurité, traces HTTP, ...).

***spring-boot-starter-devtools*** : Redémarrage automatique lors du dév.  
Configurations de développement

***spring-boot-configuration-processor*** : Traitement des annotations  
`@ConfigurationProperties`



# Classe principale SpringBoot

---

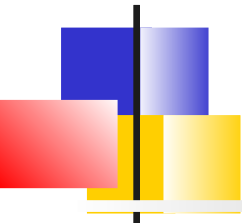
```
package com.infoq.springboot;

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

@RestController
@SpringBootApplication
public class Application {

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

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



# Annotation

## *@SpringBootApplication*

---

L'annotation **@SpringBootApplication** sur la classe principale englobe 3 annotations :

- **@EnableAutoConfiguration** : Action de la configuration automatique de SpringBoot
- **@Configuration** : Classe de configuration permettant de déclarer des méthodes annotées @Bean créant des beans Spring
- **@ComponentScan** : Permettant de scanner le package et les sous-packages à la recherche d'annotations Spring



# Autres annotations Spring

---

**@Component** : Annotation de classe permettant de déclarer un bean Spring

**Stéréotypes** : Annotations équivalentes à @Component mais qui précise le rôle du bean.

– Ex : **@Repository, @Service, @Controller, @RestController**

**@Autowired** : Demande d'injection d'un autre bean via son type

**@Resource** : Demande d'injection d'un autre bean via son nom

**@Value** : Injection d'une propriété de configuration

**@ConfigurationProperties** : Bean dont les attributs sont mis à jour avec les propriétés de configuration



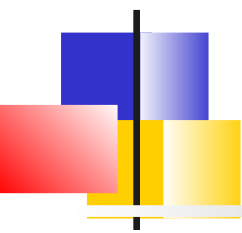


# Personnalisation de la configuration par défaut

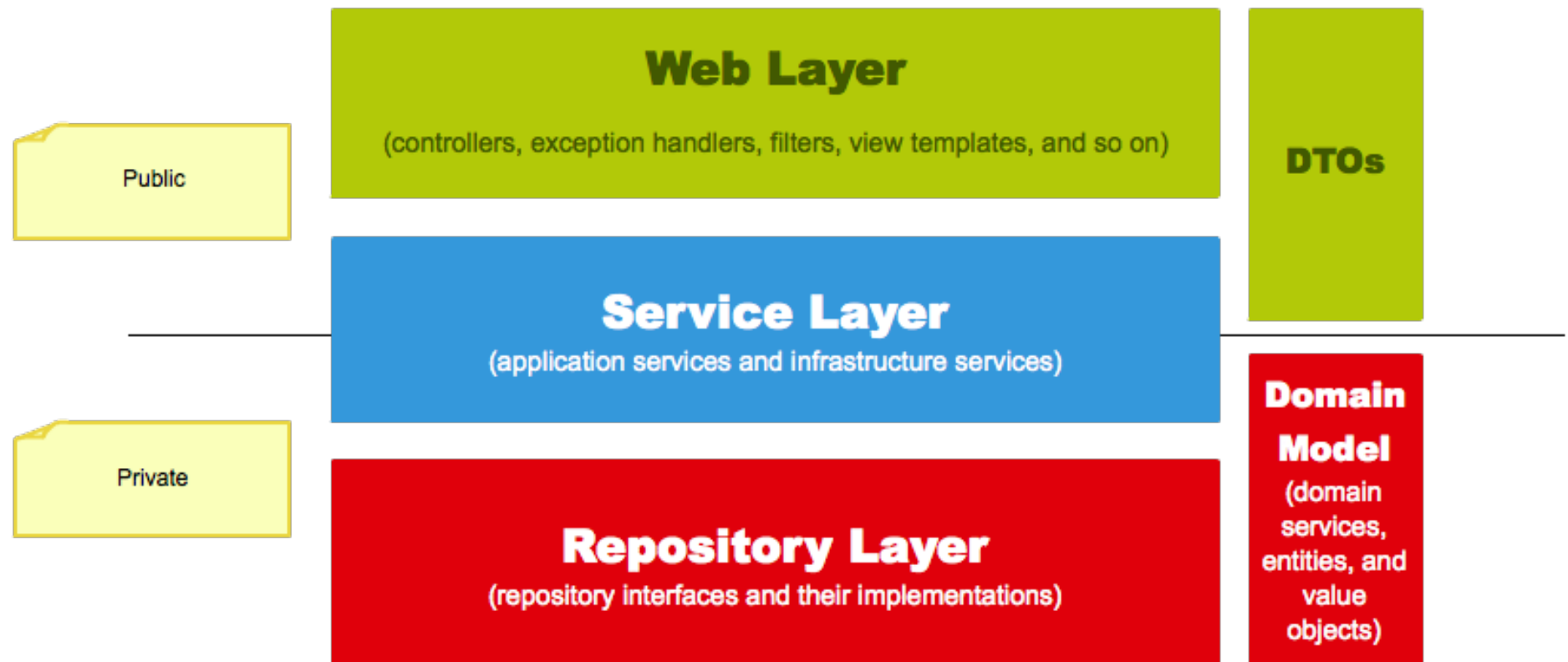
---

La configuration par défaut peut être surchargée par différents moyens

- Les propriétés de configuration via :
  - des **fichiers de configuration externe** (*.properties* ou *.yaml*)
  - Des variables d'environnement
  - Des arguments de commande en ligne
- Des classes utilitaires Spring **\*Configurer** ou **\*Customizer** permettant de surcharger la configuration par défaut via l'API
- Utiliser des **classes spécifiques** au bean que l'on veut surcharger (exemple *AuthenticationManagerBuilder*)
- La **définition de Beans** remplaçant les beans par défaut
- La **désactivation** de l'auto-configuration



# Architecture classique projet



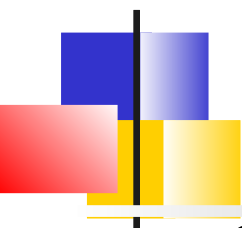


# Introduction

---

Rappels Spring  
Offre Spring Boot

**Propriétés de configuration, profils**



# Propriétés de configuration

---

Spring Boot permet d'externaliser la configuration des beans applicatifs et des beans d'intégration :

- Ex : Externaliser l'adresse de la BD, la configuration d'un client, ...

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

Les valeurs des propriétés sont ensuite injectées dans les beans :

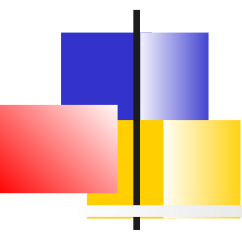
- Directement via l'annotation ***@Value***
- Ou associer à un objet structuré via l'annotation ***@ConfigurationProperties***



# 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

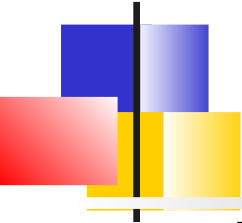
---

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
```



## Surcharge des propriétés via commande en ligne ou variable d'environnement

---

Les caractères -- permettent de positionner des propriétés lors du démarrage :

```
java --jar monAppli.jar --server.port=9000
java --jar monAppli.jar
  --spring.config.name=application,jdbc
  --spring.config.location=file:///Users/home/config
```

Les variables d'environnement peuvent également être utilisée :

```
export SERVER_PORT=9000
java --jar monAppli.jar
```



# 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

---

**@Component**

**@ConfigurationProperties("app")**

**@Validated**

```
public class MyAppProperties {
```

```
    @Pattern(regex = "\\d{3}-\\d{3}-\\d{4}")
```

```
    private String adminContactNumber;
```

```
    @Min(1)
```

```
    private int refreshRate;
```

```
        . . . . .
```

```
}
```



# Profils

---

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.

- Certains beans ne sont enregistrés que si leur profil est activé
- Les propriétés de configuration sont différentes en fonction des profils





# Beans conditionnés à des profils

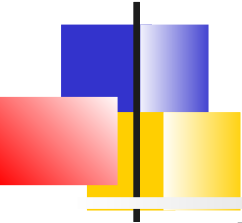
---

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

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

    // ...

}
```



# 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
```



# 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



# Persistence

---

## **Principes de SpringData** SpringBoot et JPA



# 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\*[OrderBy\*]*
- 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



# Paramètres et valeurs de retour

---

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);
```



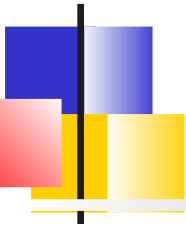
# Limite

---

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 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);  
}
```



# Persistence

---

Principes de SpringData  
**SpringBoot et 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***

# Rappels : Classes entités et associations

**@Entity**

```
public class Theme {  
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
    private String label;  
    @OneToMany(cascade = CascadeType.ALL)  
    private Set<MotClef> motclefs = new HashSet<MotClef>();  
}
```

**@Entity**

```
public class MotClef {  
    @Id  
    private Long id;  
    private String mot;  
  
    public MotClef(){}  
}
```



# 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 Hikari :

```
# Timeout en ms si pas de connexions dispo.  
spring.datasource.hikari.connection-timeout=10000
```

```
# Dimensionnement du pool  
spring.datasource.hikari.maximum-pool-size=50  
spring.datasource.hikari.minimum-idle= 10
```





# Propriétés

---

Les bases de données JPA embarquées sont créées automatiquement.

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

- 5 valeurs possibles : *none, validate, update, create, create-drop*

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
```



# Comportement transactionnel des Repository

---

Par défaut, les méthodes CRUD sont transactionnelles.

Pour les opérations de lecture, l'indicateur *readOnly* de configuration de transaction est positionné.

Toutes les autres méthodes sont configurées avec un *@Transactional* simple afin que la configuration de transaction par défaut s'applique



## *@Transactional* et *@Service*

---

Il est courant d'utiliser une façade (bean *@Service*) pour implémenter une fonctionnalité métier nécessitant plusieurs appels à différents Repositories

L'annotation *@Transactional* permet alors de délimiter une transaction pour des opérations non CRUD.



# Example

---

**@Service**

```
class UserManagementImpl implements UserManagement {  
  
    private final UserRepository userRepository;  
    private final RoleRepository roleRepository;  
  
    public UserManagementImpl(UserRepository userRepository,  
        RoleRepository roleRepository) {  
        this.userRepository = userRepository;  
        this.roleRepository = roleRepository;  
    }  
}
```

**@Transactional**

```
public void addRoleToAllUsers(String roleName) {  
  
    Role role = roleRepository.findByName(roleName);  
  
    for (User user : userRepository.findAll()) {  
        user.addRole(role);  
        userRepository.save(user);  
    }  
}
```



# 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`





# Applications Web

---

## **Rappels Spring MVC**

Support pour les APIs Rest  
Spring Boot pour les APIs Rest



# Introduction

---

*SpringBoot* est adapté pour le développement d'API REST

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

Spring MVC permet entre autre de déclarer des beans de type

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

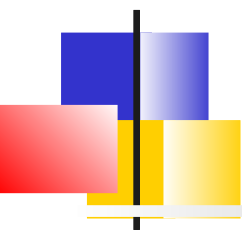


# Rappels Spring MVC

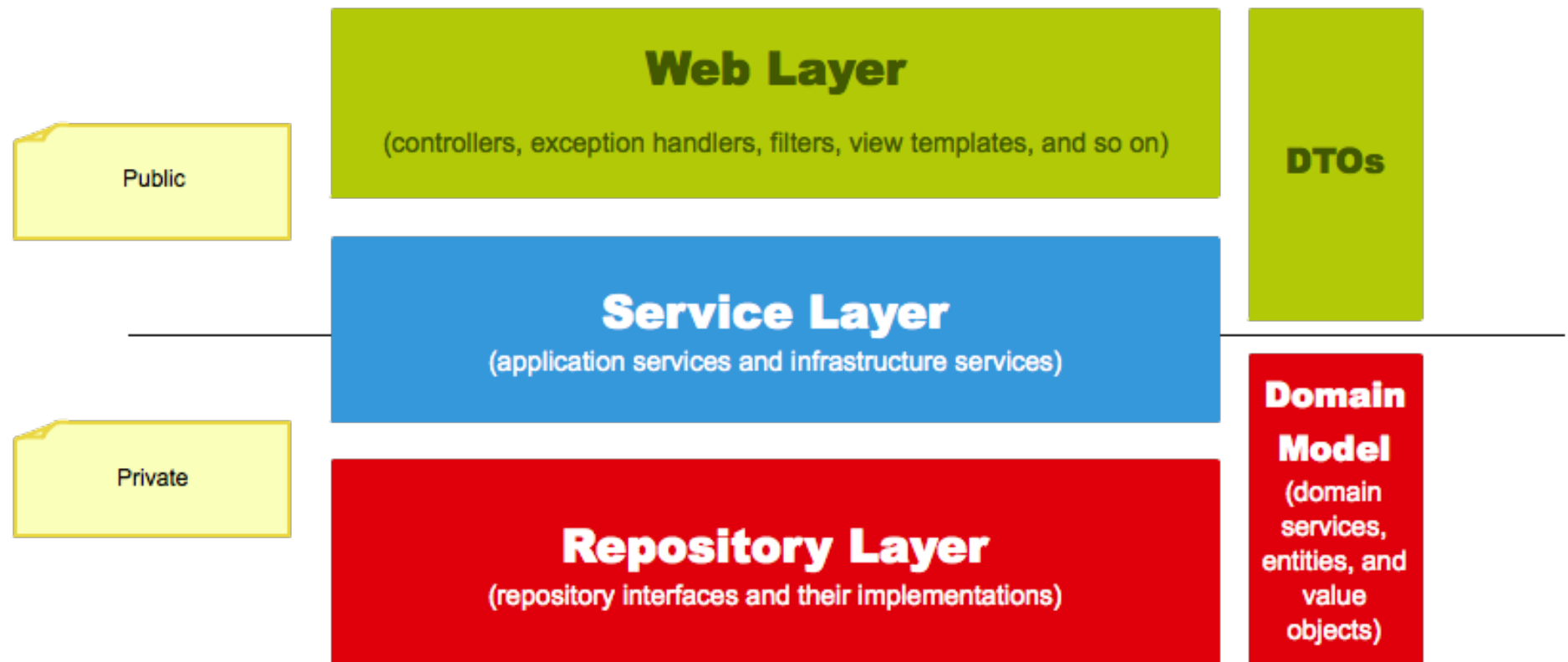
---

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





# *@RestController*

---

L'annotation **@RestController** se place sur de simples POJOs dont les méthodes publiques sont généralement accessible via HTTP

## **@RestController**

```
public class HelloWorldController {  
  
    @RequestMapping("/helloWorld")  
    public String helloWorld() {  
        return "Hello World";  
    }  
}
```



# @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}")*



# Types des arguments de méthode

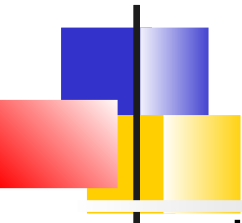
---

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

- La requête ou réponse HTTP (ServletRequest, HttpServletRequest, spring.WebRequest, ...)
- La locale, la time zone
- Les streams d'entrée/sortie
- La méthode HTTP
- L'utilisateur authentifié par HTTP (Principal)
- Dans le cadre d'une application MVC classique :
  - La session HTTP (HttpSession)
  - 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
- **@ModelAttribute** : Un attribut de requête
- **@ModelAttribute** : Un attribut du modèle (requête, session, etc.)
- **@Valid** : S'assure que les contraintes sur l'argument sont valides



# 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*

---

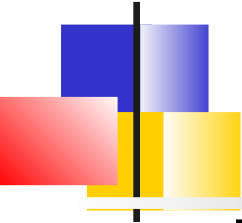
```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

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

    // ...

}
```



# Types des valeurs de retours des méthodes

---

Les types des valeurs de retour possibles sont :

- Pour le modèle MVC :
  - *ModelAndView*, *Model*, *Map*
  - Des Vues : *View*, *String*
- *void* : Si le contrôleur a lui-même généré la réponse
- Pour le modèle REST :
  - Une classe Modèle ou DTO converti via un *HttpConverter* (REST JSON) qui fournit le corps de la réponse HTTP
  - Une *ResponseEntity*<> permettant de positionner les codes retour et les entêtes HTTP



# 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
- ...



# Applications Web

---

Rappels Spring MVC  
**Support pour les APIs Rest**  
Spring Boot 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) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

    @PostMapping
    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 bidirectionnelles entre classes du modèle
- Adaptation aux besoins de l'interface de front-end
- Optimisation du volume de données échangées



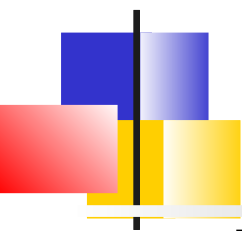
# API Jackson

---

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**



# Alternative à la sérialisation

---

Une API comme Jackson propose une sérialisation par défaut pour les classes modèles basée sur les getter/setter.

Pour adapter la sérialisation par défaut à ses besoins, 3 alternatives :

- Créer des classes DTO spécifiques
- Utiliser les annotations proposées par Jackson
- Implémenter ses propres *ObjectMapper*



# Annotations Jackson

---

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

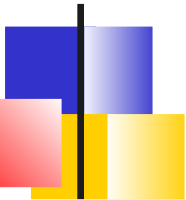
Permettant de définir les propriétés JSON

**@JsonRootName** : Arbre JSON

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

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

....



# 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> {  
        // ...  
    }  
}
```



# Applications Web

---

Rappels Spring MVC  
Support pour les APIs Rest  
**Spring Boot pour les 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
- Fourniture automatique de *RestTemplateBuilder* pour effectuer des appels REST



# 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 se configurer globalement en surchargeant la méthode *addCorsMapping* :

```
@Configuration
public class MyConfiguration implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**");
    }
}
```

A noter qu'il est également possible de configurer le cors individuellement sur les contrôleurs via l'annotation **@CrossOrigin**



# Exemple *Intercepteurs*

---

```
@SpringBootApplication
public class MyApplication implements WebMvcConfigurer {

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

    @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 :

- Modèle MVC
  - Implémenter **ErrorController** et l'enregistrer comme Bean
  - Ajouter un bean de type **ErrorAttributes** qui remplace le contenu de la page d'erreur
- Modèle REST
  - L'annotation **ResponseStatus** sur une exception métier lancée par un contrôleur
  - Utiliser la classe **ResponseStatusException** pour associer un code retour à une Exception
  - Ajouter une classe annotée par **@ControllerAdvice** pour centraliser la génération de réponse lors d'exception



# Exemple

---

**@ResponseStatus(value = HttpStatus.NOT\_FOUND)**

```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String message) {  
        super(message);  
    }  
    public MyResourceNotFoundException(Throwable cause) {  
        super(cause);  
    }  
}
```



# *ResponseStatusException*

---

```
@GetMapping(value =("/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
    }
    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```



# 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);
    }
}
```





# 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);
    }
}
```



# SpringDoc

---

**SpringDoc** est un outil qui simplifie la génération et la maintenance de la documentation des API REST

Il est basé sur la spécification OpenAPI 3 et s'intègre avec Swagger-UI

Il suffit de placer la dépendance dans le fichier de build :

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-ui</artifactId>
  <!-- OU : springdoc-openapi-webflux-ui -->
  <version>1.5.2</version>
</dependency>
```



# Fonctionnalités

---

Par défaut,

- La description OpenAPI est disponible à :  
<http://localhost:8080/v3/api-docs/>
- L'interface Swagger à :  
<http://localhost:8080/swagger-ui.html>

SpringDoc prend en compte

- les annotations javax.validation positionnées sur les DTOs
- Les Exceptions gérées par les @ControllerAdvice
- Les annotations de OpenAPI  
<https://javadoc.io/doc/io.swagger.core.v3/swagger-annotations/latest/index.html>

SpringDoc peut être désactivé via propriété :  
`springdoc.api-docs.enabled=false`



# Sécurité et Spring Boot

---

## **Rappels Spring Security**

Modèles stateful/stateless

Apports de Spring Boot

Exemple JWT



# 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 aux URLs et aux méthodes des services métier



# Principe et mécanisme

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

La configuration crée alors une chaîne de filtres via le bean **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 ...

Sans SpringBoot, Il est 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*)

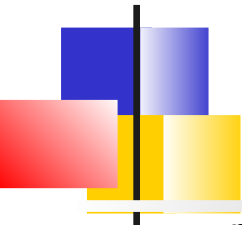




# Personnalisation

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 les filtres de *springSecurityFilterChain*
- *L'authentification : 2 alternatives*
  - **`void configure(AuthenticationManagerBuilder auth)`** : Permet de construire le gestionnaire d'authentification qui peut être fourni par Spring (*inMemory, jdbc, ldap, ...*) ou complètement personnalisé par l'implémentation d'un bean **UserDetailsService**
  - **`AuthenticationManager authenticationManagerBean()`** : Création d'un bean implémentant la vérification du mot de passe
- **`void configure(WebSecurity http)`** : Permet de définir le comportement du filtre lors des demandes de ressources statiques



# 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

---

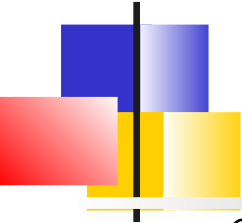
Pour debugger la configuration :

- 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* avec gestion mémoire

---

```
@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 une classe implémentant ***UserDetailsService***

L'interface contient une seule méthode :

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

- Elle 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 si le mot de passe saisi correspond.*

Le *UserDetailsService* est ensuite fourni à  
*l'authenticationManagerBuilder* :

```
@Configuration
public class UserDetailsServiceConfiguration extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsService usersDetailsService

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(usersDetailsService)
    }
}
```



# 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

---

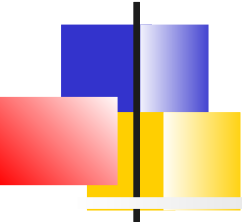
Spring Security 5 nécessite que les mots de passes soient encodés

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);  
}
```



# Sécurité et Spring Boot

---

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



# 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 crédeniels 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()`

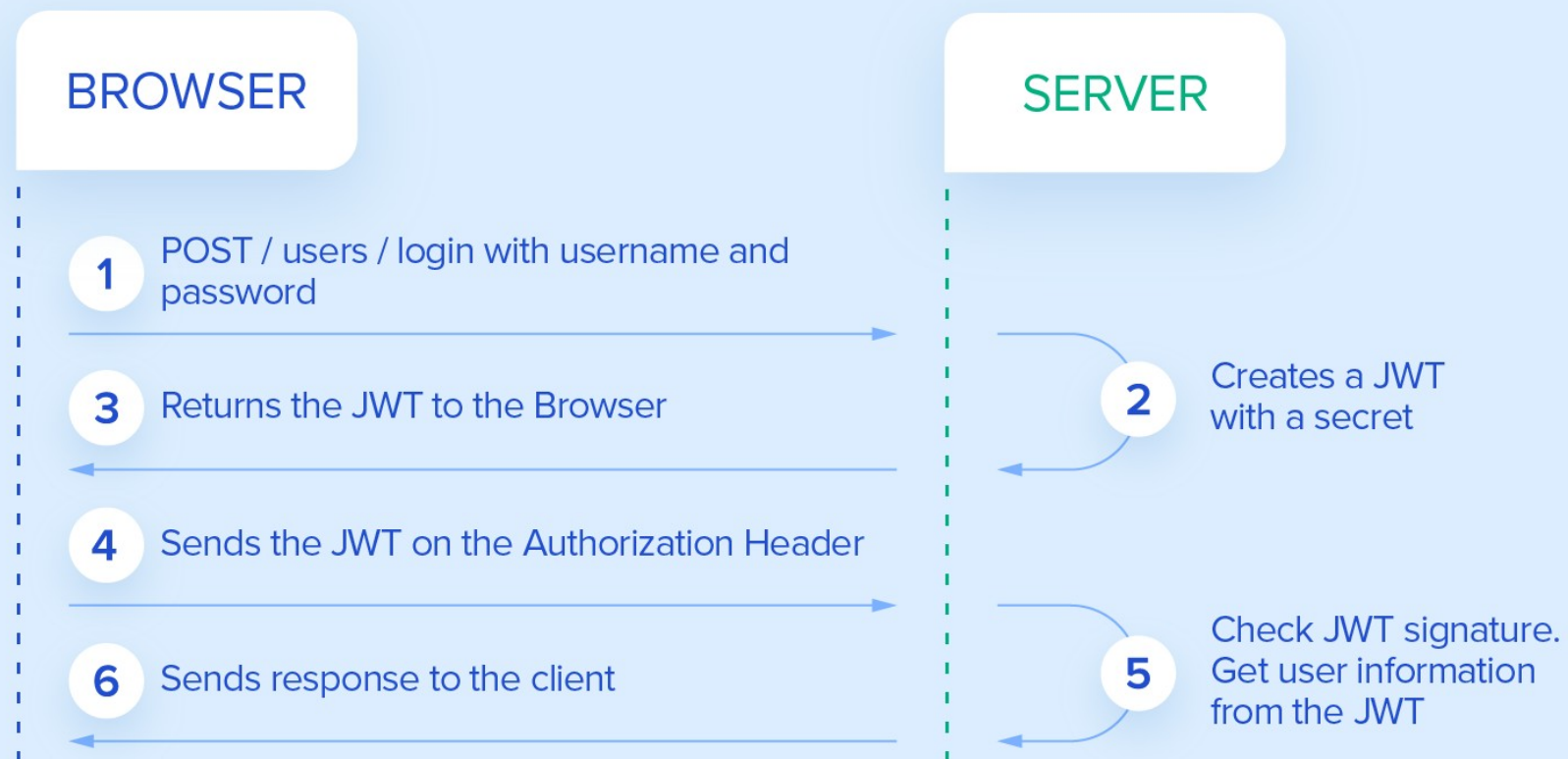


# Processus d'authentification appli REST

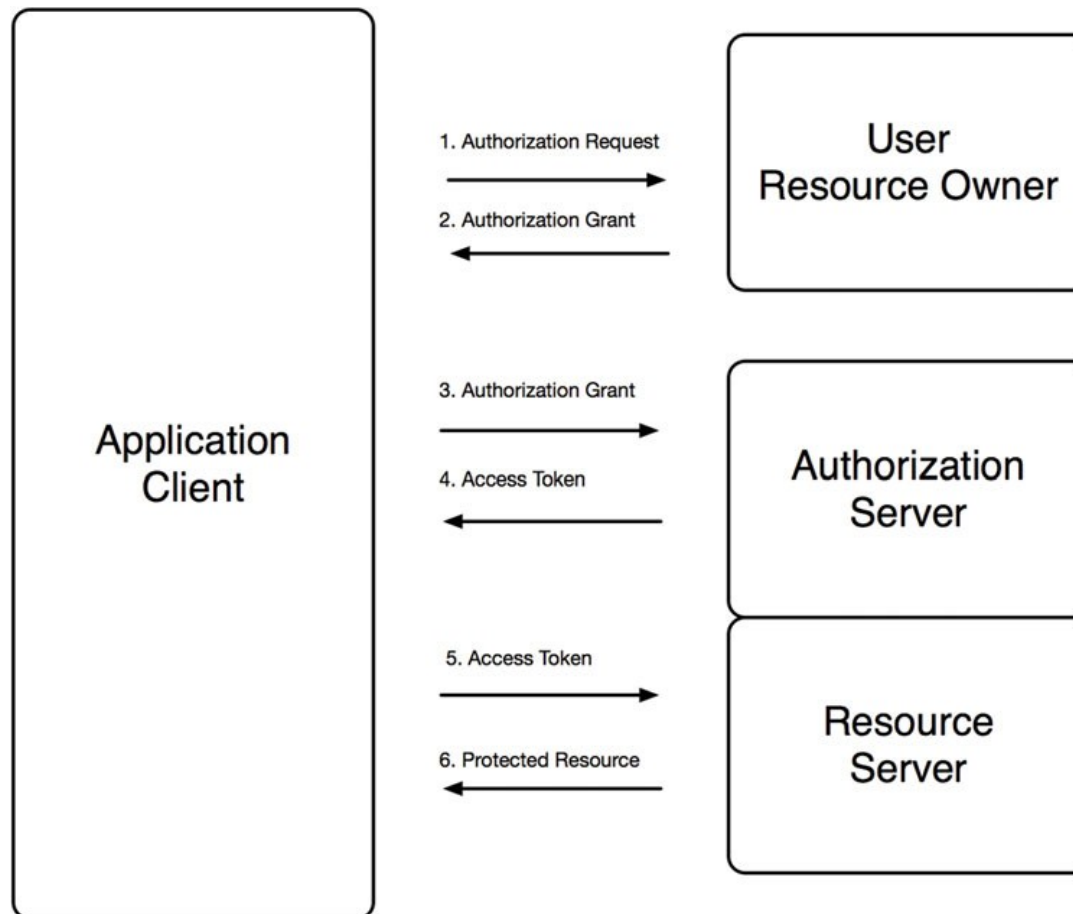
---

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



# Modèle *oAuth2*





# Sécurité et Spring Boot

---

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





# 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



# 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

---

La personnalisation consiste à définir une classe *@Configuration* de type *WebSecurityConfigurerAdapter* permettant de :

- D'adapter le filtre de sécurité afin de s'adapter au modèle stateful, stateless, oAuth, ...
- De redéfinir les ACLs sur les ressources :
- De définir son gestionnaire d'authentification :
  - Fournir un bean de type *AuthenticationManager*
  - Ou en configurer via *AuthenticationManagerBuilder*



# 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



# Sécurité et Spring Boot

---

Rappels Spring Security  
Modèles stateful/stateless  
Apports de Spring Boot  
**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

## 1. Configuration chaîne de filtres

---

```
@Autowired
JWTFilter jwtFilter ;

@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and() // Gestion des entêtes Cors avec l'entête d'Authorization
        .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() // Configuration Filtre JWT
        .addFilterBefore(jwtFilter,
                        UsernamePasswordAuthenticationFilter.class);
}
```



# *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*

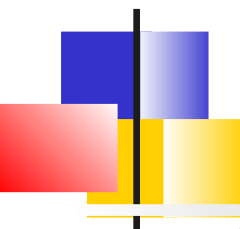
## *2. Implémentation du filtre JWT*

---

```
public class JWTFilter extends GenericFilterBean {
    private final 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

## 3. TokenProvider Création du jeton

```
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

## 4. TokenProvider 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);
}
```



# *Exemple JWT*

## *4. Fourniture d'un point d'accès*

---

```
@RequestMapping("/authenticate")
public ResponseEntity<JWTToken> authorize(@Valid @RequestParam String
login, @RequestParam String password) {

    UsernamePasswordAuthenticationToken authenticationToken =
        new UsernamePasswordAuthenticationToken(login, password);

    Authentication authentication =
this.authenticationManager.authenticate(authenticationToken);
    SecurityContextHolder.getContext().setAuthentication(authentication);

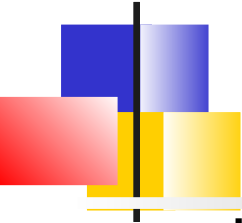
    String jwt = tokenProvider.createToken(authentication, true);
    HttpHeaders httpHeaders = new HttpHeaders();
    httpHeaders.add(JWTConfigurer.AUTHORIZATION_HEADER, "Bearer " + jwt);
    return new ResponseEntity<>(new JWTToken(jwt), httpHeaders,
HttpStatus.OK);
}
```



# SpringBoot et les tests

---

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



# *spring-boot-starter-test*

---

L'ajout de ***spring-boot-starter-test*** (dans le scope test), 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 Principalement utiliser pour tester le contexte applicatif en entier (Tests fonctionnels)
- Annotations permettant des tests auto-configurés. (Tests d'intégration par couche)  
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



# 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;

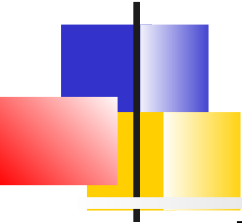
    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



# SpringBoot et les tests

---

Rappels Spring Test  
Apports de Spring Boot  
**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*



# Example

---

```
@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



# Annexes





# 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...  
    }  
}
```



# Particularités Spring Boot

---

**Monitoring avec actuator**

Déploiement

Support pour Docker



# 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





# Particularités Spring Boot

---

Monitoring avec actuator

**Déploiement**

Support pour Docker



# 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
```



# Particularités Spring Boot

---

Monitoring avec actuator  
Déploiement  
**Support pour Docker**



# Dockerfile basique

---

Il est facile de construire un jar exécutable d'une application Spring Boot via les plugins Maven/Gradle

Un Dockerfile basique est alors :

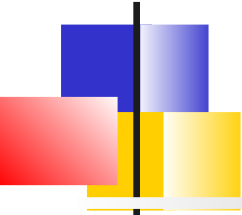
```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Pour le construire :

```
docker build --build-arg JAR_FILE=target/*.jar -t myorg/myapp .
```

Pour l'exécuter :

```
docker run -p 8080:8080 myorg/myapp
```



# Dockerfile + sophistiqué

---

Pour pouvoir passer des variables  
d'environnement et des propriétés  
SpringBoot au démarrage

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["sh", "-c", "java ${JAVA_OPTS} -jar /app.jar ${0} $@"]
```

On peut alors démarrer l'image via :

```
docker run -p 9000:9000 -e "JAVA_OPTS=-Ddebug -Xmx128m"
myorg/myapp --server.port=9000
```



# Taille des images

---

Les images alpine sont plus petites que les images standard openjdk de Dockerhub.

20 mB sont économisés en utilisant une jre plutôt qu'une jdk

On peut également essayer d'utiliser jlink (à partir de Java11) pour se créer une image sur mesure en ne sélectionnant que les modules Java utilisés

- Attention pas de cache, si tous les services Java utilisent des JRE personnalisés



# Couches Docker

---

Un jar Spring Boot a naturellement des "couches" Docker en raison de son packaging.

On peut isoler dans une couche Docker les dépendances externes ainsi tous les services utilisant les mêmes dépendances pourront se construire et se lancer plus vite.  
(Cache des container runtime)



# Dockerfile à couche

---

```
$ mkdir target/dependency
$ (cd target/dependency; jar -xf ../*.jar)
$ docker build -t myorg/myapp .
```

—

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java", "-cp", "app:app/lib/*", "hello.Application"]
```



# Accélérateur de démarrage

---

Quelques astuces pour accélérer le démarrage du service :

- Améliorer le scan du classpath avec *spring-context-indexer*<sup>1</sup>
- Limiter *actuator* à ce qui est vraiment nécessaire
- SB 2.1+ et S 5.1+
- Spécifier le chemin vers le fichier de config :  
*spring.config.location*
- Désactiver JMX `spring.jmx.enabled=false`
- Exécuter la JVM avec `-noverify`
- Pour Java8, également `-XX:+UnlockExperimentalVMOptions`  
`-XX:+UseCGroupMemoryLimitForHeap`

1. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-scanning-index>





# Utilisateur dédié

---

Les services doivent s'exécuter avec un utilisateur non root

```
FROM openjdk:8-jdk-alpine
```

```
RUN addgroup -S demo && adduser -S demo -G demo  
USER demo
```



# Spring-Boot Plugin

---

Depuis la 2.3, Spring boot fournit des plugins Maven/Gradle capables de construire les images

```
$ ./mvnw spring-boot:build-image
```

```
$ ./gradlew bootBuildImage
```

- Plus besoin de Dockerfile
- Nécessite que le démon Docker s'exécute
- Par défaut le nom de l'image est :  
    `docker.io/<group>/<artifact>:latest`
- Utilise les *Cloud Native Buildpacks*



# Autres plugins

---

***Spotify Maven Plugin*** nécessite un Dockerfile et ajoute des objectifs permettant d'automatiser la construction de l'image dans le build Maven

***Palantir Gradle*** est capable de générer un Dockerfile ou d'utiliser celui que l'on fournit

***Jib*** est également un projet Google qui permet de construire des images optimisées (Docker ou OCI) sans Docker installé



# BuildPacks

---

Cloud Foundry utilise les ***buildbacks*** qui transforment automatiquement le code source en conteneur<sup>1</sup>

- Les développeurs n'ont pas besoin de se soucier des détails sur la construction du conteneur.
- Les buildpacks ont de nombreuses fonctionnalités pour la mise en cache des résultats de construction et des dépendances  
=> Souvent un buildpack s'exécute beaucoup plus rapidement qu'un docker natif



# Sortie standard d'un buildpack

---

```
$ pack build myorg/myapp --builder=cloudfoundry/cnb:bionic --path=.
2018/11/07 09:54:48 Pulling builder image 'cloudfoundry/cnb:bionic' (use --no-pull flag to skip this step)
2018/11/07 09:54:49 Selected run image 'packs/run' from stack 'io.buildpacks.stacks.bionic'
2018/11/07 09:54:49 Pulling run image 'packs/run' (use --no-pull flag to skip this step)
*** DETECTING:
2018/11/07 09:54:52 Group: Cloud Foundry OpenJDK Buildpack: pass | Cloud Foundry Build System Buildpack: pass | Cloud Foundry JVM Application Buildpack: pass
*** ANALYZING: Reading information from previous image for possible re-use
*** BUILDING:
-----> Cloud Foundry OpenJDK Buildpack 1.0.0-BUILD-SNAPSHOT
-----> OpenJDK JDK 1.8.192: Reusing cached dependency
-----> OpenJDK JRE 1.8.192: Reusing cached launch layer

-----> Cloud Foundry Build System Buildpack 1.0.0-BUILD-SNAPSHOT
-----> Using Maven wrapper
      Linking Maven Cache to /home/pack/.m2
-----> Building application
      Running /workspace/app/mvnw -Dmaven.test.skip=true package
...
---> Running in e6c4a94240c2
---> 4f3a96a4f38c
---> 4f3a96a4f38c
Successfully built 4f3a96a4f38c
Successfully tagged myorg/myapp:latest
```



# *KNative*

---

***KNative*** est un projet initié par Google qui a pour but de fournir une infrastructure *ServerLess* au dessus d'un cluster Kubernetes

Comme les *buildpacks*, il est capable de construire les conteneurs à partir du code source.