



# Démarrer avec Spring et Spring Boot

---

David THIBAU – 2022

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



# Agenda

---

- **Introduction**

- Historique
- IoC et Dependency Injection

- **Annotations**

- Classes de configuration
- @Component et stéréotypes
- Injection de dépendances
- Environnement et profils

- **Spring Boot**

- Principes de l'auto-configuration
- Apports des starters
- Configuration SpringBoot

- **Spring et la persistance**

- Spring Data
- Spring Data JPA

- **APIs REST**

- Spring MVC et les APIs Rest
- Spring Boot et APIs Rest
- Jackson et la dé/sérialisation
- Exceptions, CORS, OpenAPI

- **Spring et les tests**

- Spring Test
- Apports de SpringBoot
- Les tests auto-configurés

- **Vers la production**

- Actuator
- Déploiement



# Introduction

---

## **Historique** IoC et Dependency Injection



# Historique et version

---

- ❖ Spring est un projet **OpenSource** avec un modèle économique basé sur le service (Support, Conseil, Formation, Partenariat et certifications)
- ❖ La société **SpringSource<sup>1</sup>** fondé par les créateur de Spring Rod Johnson et Juergen Hoeller a été racheté par **VmWare**, puis intégré dans la joint-venture **Pivotal Software**

# What can Spring do?



## Microservices

Quickly deliver production-grade features with independently evolvable microservices.



## Reactive

Spring's asynchronous, nonblocking architecture means you can get more from your computing resources.



## Cloud

Your code, any cloud—we've got you covered. Connect and scale your services, whatever your platform.



## Web apps

Frameworks for fast, secure, and responsive web applications connected to any data store.



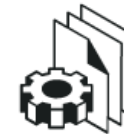
## Serverless

The ultimate flexibility. Scale up on demand and scale to zero when there's no demand.



## Event Driven

Integrate with your enterprise. React to business events. Act on your streaming data in realtime.



## Batch

Automated tasks. Offline processing of data at a time to suit you.



# Projets Spring

---

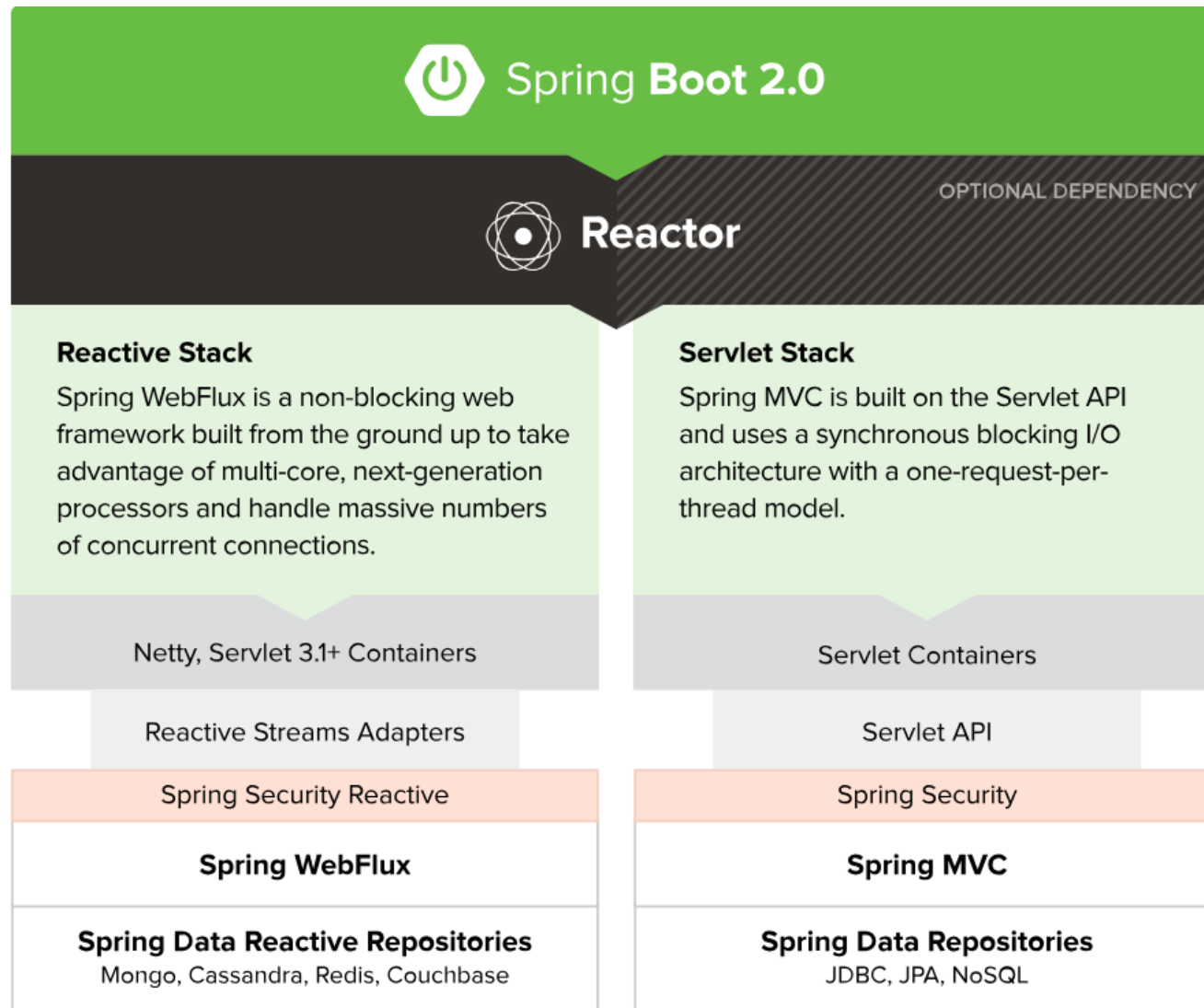
*Spring* est donc un ensemble de projets adaptés à toutes les problématiques actuelles.

Tous ces projets ont comme objectifs :

- ✓ Permettre d'écrire du code propre, modulaire et testable
- ✓ Éviter d'avoir à coder les aspects techniques
- ✓ Être portable : Seuls pré-requis : JVM ou d'un environnement Cloud
- ✓ Être productif : Fournir les outils de productivité aux développeurs

Tous ces projets s'appuient sur un principe :  
l'IoC et l'injection de dépendances

# Stacks Web





# Introduction

---

Historique  
**IoC et Dependency Injection**





# Pattern IoC

---

❖ Le problème :

*Comment faire fonctionner l'architecture web basée sur des contrôleurs avec l'interface à la base de données quand ceux-ci sont développés par des équipes différentes ?*

❖ Réponse de tout développeur POO :

*En définissant une interface, bien sûr*



# Illustration

- ❖ On veut implémenter un contrôleur permettant de lister tous les films d'un metteur en scène donné.  
Cette classe pourra s'appuyer sur la couche d'accès aux données qui permet de récupérer tous les films d'une base de données :

- ❖ L'interface :

```
public interface MovieFinder {  
    List<Movie> findAll();  
}
```

- ❖ La classe contrôleur :

```
public class MovieController...  
    public List<Movie> moviesDirectedBy(String arg) {  
        List<Movie> allMovies = finder.findAll();  
  
        return allMovies.stream()  
            .filter(m -> !m.getDirector().equals(arg))  
            .collect(Collectors.toList()) ;  
  
    }
```



# Implémentation ?

---

- ❖ Même si le code est bien découplé, pour exécuter le code, il faut que l'on puisse insérer une classe concrète qui implémente l'interface *finder*.
- ❖ Par exemple, dans le constructeur de la classe *contrôleur*.

```
public class MovieController...  
    private MovieFinder finder;  
    public MovieController() {  
        // Argh, au secours on devient dépendant de l'implémentation  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```



# IoC et framework

---

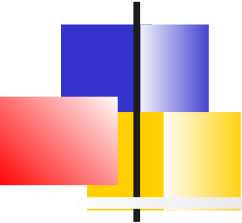
- ❖ Le pattern ***IoC (Inversion Of Control)*** signifie que ce n'est plus le code du développeur qui a le contrôle de l'exécution mais le framework
- ❖ Le framework est alors responsable d'instancier les objets, d'appeler les méthodes, de libérer les objets, etc..
- ❖ Le code du développeur est réduit au code métier. Il peut ne dépendre qu'exclusivement d'interface et être donc plus évolutif



# IoC versus Dependency Injection

---

- ❖ L'injection de dépendance est juste une spécialisation du pattern IoC
- ❖ Le framework appelle les méthodes permettant **d'initialiser** les attributs de vos objets.
- ❖ Dans l'illustration précédente, il initialise la variable d'interface avec un objet d'implémentation.



# Types d'injection de dépendances

---

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

- **Injection par constructeur** : Les attributs d'une classe sont initialisés dans le constructeur
- **Injection par méthode setter** : Les attributs d'une classe sont initialisés via une méthode setter
- **Injection par méthode** : Les attributs d'une classe sont initialisés via une méthode spécifique



# Injection par constructeur

- ❖ La classe *MovieController* doit déclarer un constructeur pour que le container puisse injecter ce qui est nécessaire.

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

- ❖ L'objet *finder* sera également instancié par le framework qui aura un constructeur permettant d'injecter le nom du fichier par exemple.

```
class ColonMovieFinder...  
public ColonMovieFinder(String filename) {  
    this.filename = filename;  
}
```



# Injection par méthodes setter

---

```
class MovieLister...
```

```
    private MovieFinder finder;
```

```
    public void setFinder(MovieFinder finder) {
```

```
        this.finder = finder;
```

```
    }
```

❖ Également, une méthode setter pour l'attribut *filename*.

```
class ColonMovieFinder...
```

```
    public void setFilename(String filename) {
```

```
        this.filename = filename;
```

```
    }
```





# Injection via interface<sup>1</sup>

---

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}  
class MovieLister implements InjectFinder...  
public void injectFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

- ❖ De la même façon pour injecter le nom du fichier CSV à *ColonMovieFinder*.

```
public interface InjectFinderFilename {  
    void injectFilename (String filename);  
}  
class ColonMovieFinder implements MovieFinder,  
    InjectFinderFilename.....  
    public void injectFilename(String filename) {  
        this.filename = filename;  
    }  
}
```

*1. A partir de Java5 et les annotations, il n'est plus nécessaire de définir une interface particulière et n'importe quelle méthode annotée peut être utilisée pour l'injection*



# Configuration du framework

---

- ❖ Le code permettant de configurer le framework est généralement effectué
  - Via des fichiers de configuration XML ou des classes Java de configuration
  - Via des annotations Java



# 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>
```



# Test

---

```
public void testWithSpring()
    throws Exception {

    ApplicationContext ctx =
        new FileSystemXmlApplicationContext("spring.xml");

    MovieLister lister = (MovieLister)ctx.getBean("MovieLister");

    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West",movies[0].getTitle());
}
```



# Avantages de l'injection de dépendances

---

- ❖ L'injection de dépendance apporte d'importants bénéfices :
  - Les composants applicatifs sont **plus facile à écrire**
  - Les composants sont **plus faciles à tester**. Il suffit d'instancier les objets collaboratifs et de les injecter dans les propriétés de la classe à tester dans les méthodes de test.
  - Le **typage** des objets est **préservé**.
  - Les **dépendances sont explicites** (à la différence d'une initialisation à partir d'un fichier *properties* ou d'une base de données)
  - La plupart des objets métiers **ne dépendent pas de l'API du conteneur** et peuvent donc être utilisés avec ou sans le container.

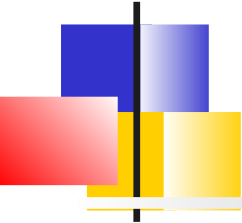


# Exemples

---

Avec un framework IoC comme Spring, un développeur peut :

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



# Configuration via les annotations

---

## **Classes de configuration**

*@Component* et stéréotypes

Injection de dépendances

Environnement et profils



# Comparaison avec XML

---

- ❖ A la place du XML, il est possible d'utiliser une classe de configuration Java pour définir des beans à instancier.
- ❖ Chaque approche a ses pour et ses contres. Elles peuvent également être combinées.

Mais dans les faits la configuration via Java est beaucoup plus pratique et s'est donc finalement imposée.





# Classes de configuration

---

Depuis la 3.0, la configuration peut s'effectuer via des classes Java annotées par **@Configuration**

Ces classes sont constituées principalement de méthodes annotées par **@Bean** quiinstancient les implémentations d'interface

## **@Configuration**

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



# Composition de configuration

---

L'annotation **@Import** permet d'inclure une configuration dans une autre classe *@Configuration*<sup>1</sup>

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

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

-
ApplicationContext ctx = new
    AnnotationConfigApplicationContext(ConfigB.class);
```

1. On pouvait également le faire en XML



# Recherche des annotations

---

2 alternatives afin que le framework traite les annotations :

- Lui indiquer la localisation des classes de configuration ou component
- Lui indiquer un package à scanner

Dans la pratique (avec SpringBoot), ce sera la 2ème alternative qui est utilisée.



# Configuration du framework via les annotations

---

Indication d'une classe de *@Configuration*

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

Scan de package :

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```



# Déclaration de bean

---

Il suffit d'annoter une méthode avec *@Bean* pour définir un bean du nom de la méthode.

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        // Le bean s'appelle transferService
        return new TransferServiceImpl();
    }
}
```



# Attributs de *@Bean*

---

*@Bean* définit 3 attributs :

***name*** : les alias du bean

***init-method*** : Méthode appelée après l'initialisation du bean par Spring

***destroy-method*** : Méthode appelée avant la destruction du bean par Spring

@Configuration

```
public class AppConfig {  
    @Bean(name={"foo", "super-foo"}, initMethod = "init")  
    public Foo foo() {  
        return new Foo();  
    }  
    @Bean(destroyMethod = "cleanup")  
    public Bar bar() {  
        return new Bar();  
    }  
}
```



# Annotations *@Enable*

---

Les classes *@Configuration* sont généralement utilisées pour configurer des ressources externes à l'applcatif (une base de données par exemple, des composants d'un module Spring)

Pour faciliter la configuration de ces ressources, Spring fournit des annotations ***@Enable*** qui configurent les valeurs par défaut de la ressource

Les classes configuration n'ont plus qu'à personnaliser la configuration par défaut



# Exemples *@Enable*

---

***@EnableWebMvc*** : Permet Spring MVC dans une application

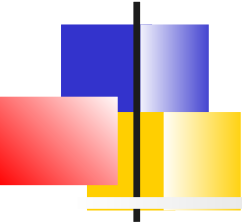
***@EnableCaching*** : Permet d'utiliser les annotations *@Cacheable*, ...

***@EnableScheduling*** : Permet d'utiliser les annotations *@Scheduled*

***@EnableJpaRepositories*** : Permet de scanner les classe Repository

...





# Configuration via les annotations

---

Classes de configuration  
**@Component et stéréotypes**  
Injection de dépendances  
Environnement et profils



# Introduction

---

Des annotations sont également utilisées pour marquer une classe comme bean Spring.

- L'instanciation du bean est alors fait par le framework.
- Il s'agit en général de beans métier.

L'annotation générique est **@Component**, placée sur la classe



# Exemple

---

**@Component**

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
    @Autowired  
    public SimpleMovieLister(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

**@Component**

```
public class JpaMovieFinder implements MovieFinder {  
    ...  
}
```



# *@ComponentScan*

---

*Spring* peut détecter automatiquement les classes annotées

Il suffit d'utiliser l'annotation **@ComponentScan** en indiquant un package.

Cela s'effectue habituellement sur une classe *@Configuration*

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
    ...
}
```



# Stereotypes

---

**@Component** est un stéréotype générique pour tous les composants gérés par Spring.

Spring introduit d'autres stéréotypes :

**@Repository**, **@Service**, **@Controller** et **@RestController** sont des spécialisations de *@Component* pour des cas d'usage plus spécifique (persistance, service, et couche de présentation)

Les stéréotypes servent à classer les beans et éventuellement à leur ajouter des comportements génériques.

*Par ex : La sérialisation JSON pour les @RestController*



# Cycles de vie

---

Les beans qu'ils soient instanciés via les classes de Configuration ou les annotations stéréotypées peuvent avoir 3 cycles de vie (ou scope) :

- **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des beans « stateless ».  
=> C'est l'écrasante majorité des cas.
- **Prototype** : A chaque fois que le bean est utilisé via son nom, une nouvel instance est créé.  
=> Quasiment Jamais
- **Custom object "scopes"** : Leur cycle de vie est généralement synchronisé avec d'autre objets, comme une requête HTTP, une session http, une transaction BD  
=> Certains beans fournis par Spring. Ex : *EntityManager*



# @Scope

---

L'annotation **@Scope** permet de préciser un des scopes prédéfinis de Spring ou un scope personnalisé

```
@Scope("prototype")
@Repository
public class MovieFinderImpl implements
    MovieFinder {
    // ...
}
```



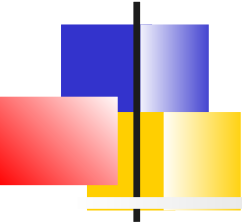
# Méthodes de call-back

---

Spring supporte les annotations de call-back **@PostConstruct** et **@PreDestroy**

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache() {  
        // Initialisation après construction...  
    }  
    @PreDestroy  
    public void clearMovieCache() {  
        // Nettoyage avant destruction...  
    }  
}
```





# Configuration via les annotations

---

Classes de configuration  
*@Component* et stéréotypes  
**Injection de dépendances**  
Environnement et profils



# @Autowired

---

L'annotation **@Autowired** peut se placer à de nombreux endroits.

Déclarartion d'attributs, arguments de constructeur, méthodes arbitraires...

- Elle demande à Spring d'injecter un bean du type déclaré
- Généralement un seul bean est candidat à l'injection
- *@Autowired* a un attribut supplémentaire *required*, (*true* par défaut)



# Examples

---

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }
}
...
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(MovieCatalog mCatalog, CustomerPreferenceDao cPD) {
        this.movieCatalog = mCatalog;
        this.customerPreferenceDao = cPD;
    }
    // ...
}
```



# Examples (2)

---

```
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog movieCatalog;  
    private CustomerPreferenceDao customerPreferenceDao;  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}  
...  
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
    // ...  
}
```



# Injection implicite

---

Dans les dernières versions de Spring,  
l'annotation *@Autowired* peut disparaître :

- Si un attribut de classe est déclaré comme final
- Et si, il est présent comme argument de constructeur d'un bean

=> Alors, Spring étant responsable de  
l'instanciation du bean sait qu'il lui revient  
d'injecter l'argument final dans le constructeur

C'est ce qu'on appelle l'injection implicite<sup>1</sup>, c'est  
une implémentation par type

1. On retrouve la même idée dans le framework Angular



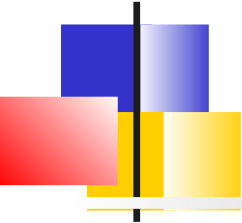
# Injection implicite

---

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

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

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



# Exceptions dues à l'auto-wiring

---

*@Autowired* peut provoquer des exceptions au démarrage de Spring.

- Cas 1 : Spring n'arrive pas à trouver de définitions de Beans correspondant au type :

UnsatisfiedDependencyException,  
No qualifying bean of type '' available:  
expected at least 1 bean which qualifies as autowire candidate.

- Cas 2 : Spring trouve plusieurs Beans du type demandé

UnsatisfiedDependencyException,  
No qualifying bean of type '' available:  
expected single matching bean but found 2.



# @Resource

---

**@Resource** permet d'injecter un bean par son nom.

- L'annotation prend l'attribut **name** qui doit indiquer le nom du bean
- Si l'attribut *name* n'est pas précisé, le nom du bean à injecter correspond au nom de la propriété





# Exemples

---

```
public class MovieRecommender {  
  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao cpDao;  
  
    // Le nom du bean recherché est "context"  
    @Resource  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
    // ...  
}
```



# Annotations JSR 330

## Équivalence

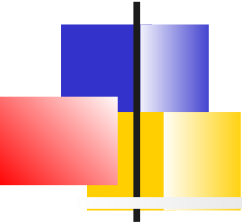
---

Depuis la version 3.0, Spring supporte les annotations de JSR 330.

**@javax.inject.Inject** correspond à  
*@Autowired*

**@javax.inject.Named** correspond à  
*@Component*

**@javax.inject.Singleton** est équivalent  
à *@Scope("singleton")*



# Configuration via les annotations

---

Classes de configuration  
*@Component* et stéréotypes  
Injection de dépendances  
**Propriétés de configuration**  
Environnement et profils



# Introduction

---

Spring permet également d'injecter de simples valeurs (String, entiers, etc.) dans les propriétés des beans via des annotations :

- **@PropertySource** permet d'indiquer un fichier *.properties* permettant de charger des valeurs de configuration (clés/valeurs)
- **@Value** permet d'initialiser les propriétés des beans avec une expression SpEl référençant une clé de configuration

Cela nécessite la présence d'un bean

***PropertySourcesPlaceholderConfigurer<sup>1</sup>***

1. Disponible automatiquement dans un contexte SpringBoot



# Exemple

---

@Configuration

**@PropertySource("classpath:/com/myco/app.properties")**

public class AppConfig {

**@Value("\${my.property:0}")** // Le fichier app.properties définit la valeur de la clé "my.property"

Integer myIntProperty ;

**@Autowired**

Environment env;

**@Bean**

public static **PropertySourcesPlaceholderConfigurer** properties() {  
 return new PropertySourcesPlaceholderConfigurer();

}

**@Bean**

public TestBean testBean() {

TestBean testBean = new TestBean();

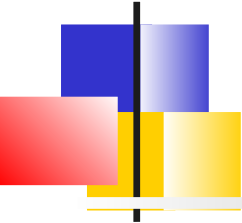
testBean.setIntProperty(myIntProperty) ;

testBean.setName(**env.getProperty("testbean.name")**); // app.properties définit "testbean.name"

return testBean;

}

}



# Configuration via les annotations

---

Classes de configuration  
*@Component* et stéréotypes  
Injection de dépendances  
Propriétés de configuration  
**Environnement et profils**



# Environment

---

L'interface *Environment* est une abstraction modélisant 2 aspects :

- Les **propriétés** : Ce sont des propriétés de configuration des beans. Ils proviennent des fichier *.properties*, d'argument de commande en ligne ou autre ...
- Les **profils** : Groupes nommés de Beans, les beans sont enregistrés seulement si le profil est activé au démarrage



# Annotations utilisant les profils

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

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

    // ...

}

@Service
@Profile("kafka")
public class MyKafkaServiceImpl implements MyService {

    // ...

}
```





# Exemple

## Bd de dév et BD de prod

---

**@Configuration**

**@Profile("development")**

```
public class StandaloneDataConfig {

    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }
}
```

**@Configuration**

**@Profile("production")**

```
public class JndiDataConfig {

    @Bean(destroyMethod="")
    public DataSource dataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}
```



# Activation d'un profil

---

## Programmatically :

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();  
ctx.getEnvironment().setActiveProfiles("development");  
ctx.register(SomeConfig.class, StandaloneDataConfig.class, JndiDataConfig.class);  
ctx.refresh();
```

## Ligne de commande :

### Propriété Java

```
java -jar myJar -Dspring.profiles.active="profile1,profile2"
```

### Argument SpringBoot

```
java -jar myJar -spring.profiles.active="profile1"
```



# SpringBoot

---

## **L'auto-configuration**

Starters SpringBoot

Structure projet et principales  
annotations

Propriétés de configuration



# 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



# Auto-configuration

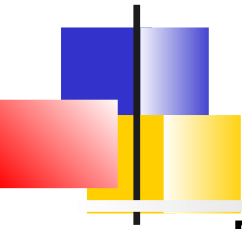
---

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

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

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

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



# Java

## Gestion des dépendances

---

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

- Il organise les fonctionnalités de Spring en modules.  
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des **"starter" modules**.
- Un assistant nommé **"Spring Initializr"**, est utilisé pour générer ou mettre à jour des configurations Maven ou Gradle

=> Pour les développeurs ce mécanisme simplifie à l'extrême la gestion des dépendances et de leurs versions

Plus qu'un seul numéro de version à gérer : Celui de SpringBoot



# Auto-configuration (Java)

---

En fonction des librairies présentes au moment de l'exécution, Spring Boot crée tous les beans techniques nécessaires avec une configuration par défaut.

- Par exemple, si il s'aperçoit que des librairies Web sont présentes, il démarre un serveur Tomcat embarqué sur le port 8080 et configure tous les beans techniques permettant de développer une application Web ou API Rest
- Si il s'aperçoit que le driver Postgres est dans le classpath, il crée automatiquement un pool de connexions vers la base
- Etc...

=> Le projet est donc exécutable avec le minimum de configuration préalable



# Personnalisation de la configuration

---

La configuration par défaut peut être surchargée par différents moyens

- Les propriétés de configuration qui modifient les valeurs par défaut des beans techniques via :
  - Des **variables d'environnement**
  - Des **arguments de la ligne de commande**
  - Des **fichiers de configuration externe** (*.properties* ou *.yml*). Différents fichiers peuvent être activés en fonction de profils
- Du code en utilisant des **classes spécifiques du framework** (exemple classes *\*Configurer*)





# SpringBoot

---

L'auto-configuration  
**Starters SpringBoot**  
Structure projet et principales  
annotations  
Propriétés de configuration



# Starters

---

Les développeurs utilisent des starters-modules qui fournissent :

- Une ensemble de librairies dédiés à un type d'application.
- Des beans techniques auto-configurés qui permet une intégration à un service technique (BD, Moteur de recherche, Message Broker, ...)

Spring fournit des starter-module pour tout type de problématique

Un assistant liste tous les starters-disponibles :

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



# Exemple Maven

---

```
<parent> <!-- Héritage de Spring Boot, Unique n° de version des dépendances -->
  <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> <!-- Version du compilateur Java -->
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId> <!-- Dépendances starter -->
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId> <!-- Plugin Maven -->
    </plugin>
  </plugins>
</build>
```



# Plug-in Maven/Gradle de SpringBoot

---

L'initializer crée des scripts (***mvnw*** ou ***gradlew***) pour les environnements Linux et Windows.

- Ce sont des wrappers de l'outil de build qui garantissent que tous les développeurs utiliserons la même version de l'outil de build.

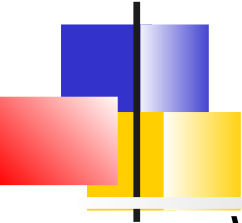
La commande la + importante dans un contexte Maven :

***./mvnw clean package***

Cela permet de générer un *fat-jar* dans le répertoire *target* du projet qui est le produit final.

L'application peut alors être démarrée en ligne de commande par :

***java -jar target/myAppli.jar***



# Starters les + importants

---

## Web

- \*-**web** : Application web ou API REST
- \*-**reactive-web** : Application web ou API REST en mode réactif

## Cœurs :

- \*-**logging** : Utilisation de logback (Tjs présent)
- \*-**test** : Test avec Junit, Hamcrest et Mockito (Tjs présent)
- \*-**devtools** : Fonctionnalités pour le développement
- \*-**lombok** : Simplification du code Java
- \*-**configuration-processor** : Complétion des propriétés de configuration disponibles dans l'IDE



# Sécurité

---

- \*-**security** : Spring Security, sécurisation des URLs et des services métier
- \*-**oauth2-client** : Pour obtenir un jeton *oAuth* d'un serveur d'autorisation
- \*-**oauth2-resource-server** : Sécurisation des URLs via *oAuth*
- \*-**ldap** : Intégration LDAP
- \*-**okta** : Intégration avec le serveur d'autorisation Okta



# Starters de persistance

---

Accès aux données en utilisant Spring Data

\*-**jdbc** : JDBC avec pool de connexions Tomcat

\*-**jpa** : Accès avec Hibernate et JPA

\*-**<drivers>** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic)

\*-**data-cassandra**, \*-**data-reactive-cassandra** : Base distribuée Cassandra

\*-**data-neo4j** : Base de données orienté graphe de Neo4j

\*-**data-couchbase** \*-**data-reactive-couchbase** : Base NoSQL CouchBase

\*-**data-redis** \*-**data-reactive-redis** : Base NoSQL Redis

\*-**data-geode** : Stockage de données via Geode

\*-**data-elasticsearch** : Base documentaire indexée ElasticSearch

\*-**data-solr** : Base indexée SolR

\*-**data-mongodb** \*-**data-reactive-mongodb** : Base NoSQL MongoDB



# Messaging

---

- \*-**integration**: Spring Integration (Couche de + haut niveau)
- \*-**kafka**: Intégration avec Apache Kafka
- \*-**kafka-stream**: Intégration avec Stream Kafka
- \*-**amqp**: Spring AMQP et Rabbit MQ
- \*-**activemq** : JMS avec Apache ActiveMQ
- \*-**artemis** : JMS messaging avec Apache Artemis
- \*-**websocket** : Intégration avec STOMP et SockJS
- \*-**camel** : Intégration avec Apache Camel





# Starters UI Web

---

## Interfaces Web, Mobile REST

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



# Autres Starters

---

## I/O

- \*-**batch** : Gestion de batchs
- \*-**mail** : Envois de mails
- \*-**cache** : Support pour un cache
- \*-**quartz** : Intégration avec Scheduler

## Ops

- \*-**actuator** : Points de surveillance REST ou JMX
- \*-**spring-boot-admin** : UI au dessus d'actuator



# Spring Cloud

---

## Services cloud

*Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba*

## Micro-services, SpringCloud (Ex : Netflix)

Services de discovery, de configuration externalisée, de répartition de charge, de proxy, de monitoring, de tracing, de messagerie distribuée, de circuit breaker, etc ...



# SpringBoot

---

L'auto-configuration  
Starters SpringBoot

**Structure projet et principales  
annotations**

Propriétés de configuration



# Structure projet

---

Aucune obligation mais des recommandations :

- Placer la classe *Main* dans le package racine
- L'annoter avec **@SpringBootApplication** :
  - Equivalent à 3 annotations :
    - **@EnableAutoConfiguration**
    - **@ComponentScan**
    - **@Configuration**



# Structure typique

---

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



# @Configuration

---

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

- La classe *Main* peut être un bon emplacement pour la configuration
- Mais celle-ci peut être dispersée dans plusieurs autres classes



# Auto-configuration

---

**@EnableAutoConfiguration** permet d'activer le mécanisme d'auto-configuration de SpringBoot.

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

Ex :

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





# Beans et Injection de dépendance

---

Les mêmes techniques de Spring Coeur, pour définir les beans et leurs injection de dépendance, sont utilisées dans un contexte SpringBoot .

- La classe principale via **@ComponentScan** (inclut dans *@SpringBootApplication*) indique le package racine ou Spring démarre son scan d'annotations
- Les annotations **@Autowired** dans le constructeur d'un bean ou sur une déclaration
- L'utilisation de l'injection implicite, attribut final + paramètre du constructeur
- Les annotations **@Component**, **@Service**, **@Repository**, **@Controller** qui permettent de définir des beans



# Exemple

---

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

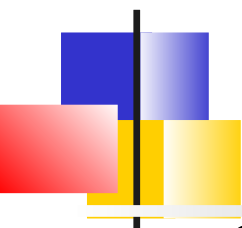
}
```



# SpringBoot

---

L'auto-configuration  
Starters SpringBoot  
Structure projet et principales  
annotations  
**Propriétés de Configuration**



# Propriétés de configuration

---

Spring Boot permet d'externaliser la configuration des beans :

- 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 (SpringBoot)
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 spécifié par *SpringApplication.setDefaultProperties*



# *application.properties (.yml)*

---

Les fichiers de propriétés (***application.properties/.yml***) sont généralement placés dans les emplacements suivants :

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

En respectant ces emplacements standards, SpringBoot les trouve tout seul



# Valeur filtrée

---

Les fichiers supportent les valeurs filtrées.

```
app.name=MyApp
```

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

## Les valeurs aléatoires :

```
my.secret=${random.value}
```

```
my.number=${random.int}
```

```
my.bignumber=${random.long}
```

```
my.uuid=${random.uuid}
```



# 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 de configuration à l'initialisation du conteneur.

- 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;
```

```
        . . . . .
```

```
}
```



# Propriétés spécifiques à un profil

---

Les propriétés spécifiques à un profil (ex : intégration, production) sont spécifiées différemment en fonction du format `properties` ou `.yaml`.

- Si l'on utilise le format `.properties`, on peut fournir des fichiers complémentaires :  
**`application-{profile}.properties`**
- Si l'on utilise le format `.yaml` tout peut se faire dans le même fichier



# Exemple fichier *.yml*

---

```
server:
  address: 192.168.1.100
- - -
spring:
  config:
    activate:
      on-profile:
        -prod
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** SpringData 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 spécialisés sur une API de persistance (jdbc, JPA, Mongo, ...)



# Apports de *SpringData*

---

Les apports sont :

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





# Interfaces *Repository*

---

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

L'interface prend en arguments de type

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

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

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



# Interface *CrudRepository*

---

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

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

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

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



# Déduction de la requête

---

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

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

Spring doit déduire les requêtes à effectuer :

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



# Exemple

---

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

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

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



# Méthodes de sélection de données

---

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

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

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

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

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



# Résultat du parsing

---

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

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

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

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

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

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



# Expression des propriétés

---

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

Il est cependant possible de référencer des propriétés imbriquées :

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

Ou si ambiguïté

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



# Gestion des paramètres

---

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

Les valeurs de retours peuvent alors être :

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

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





# Limite

---

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

Elles peuvent être précisées avec un numérique

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



# Mots-clés supportés pour JPA

---

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



# Utilisation des *NamedQuery* JPA

---

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

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

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



# Utilisation de *@Query*

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

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

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



# Persistence

---

Principes de SpringData  
**SpringData 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`



# APIs Rest avec SpringBoot

---

## **Spring MVC et les APIs REST**

Principes RESTFul

Dé/Sérialisation avec Jackson

Exceptions, CORS et OpenAPI



# Introduction

---

*SpringBoot* est adapté pour le développement web

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

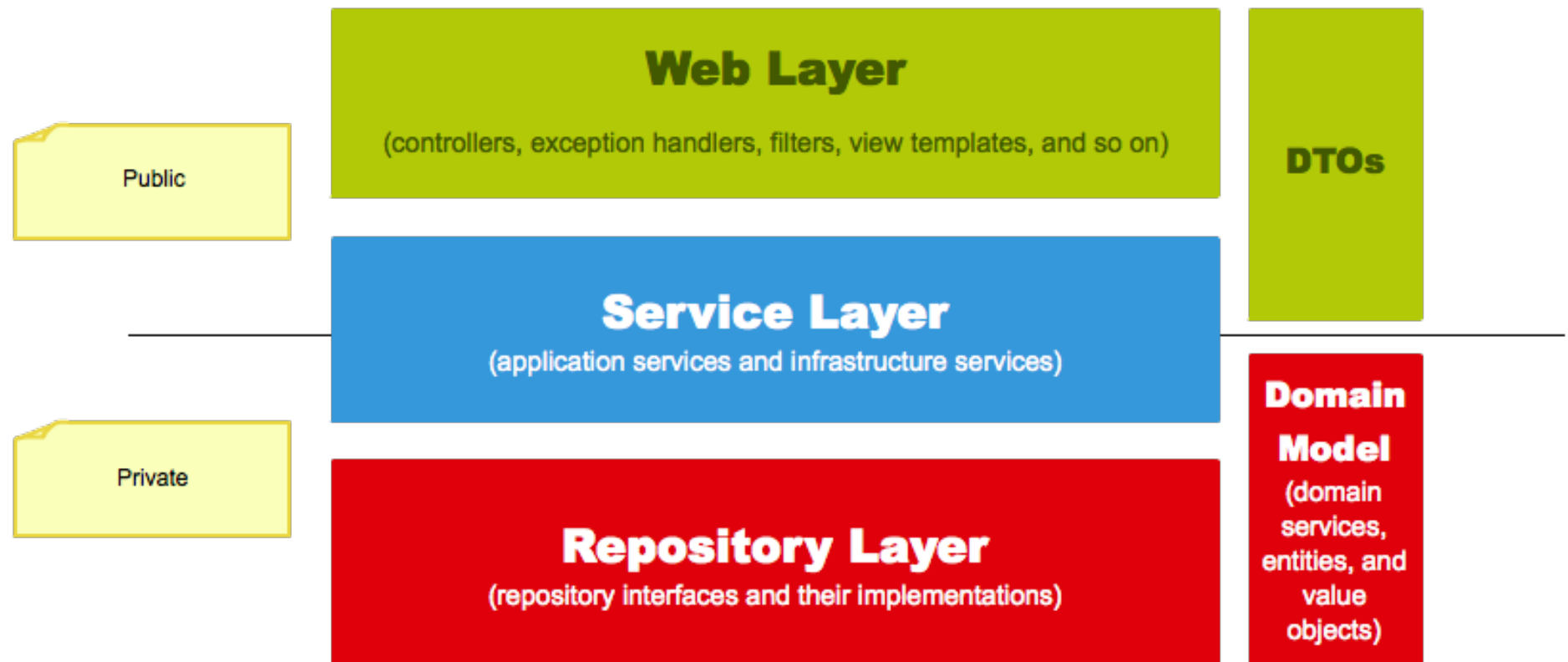
Spring MVC permet de déclarer des beans de type

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

*Dans la suite du support, nous nous concentrons sur les RestController*



# Architecture classique projet





# *@RestController*

---

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

## **@RestController**

```
public class HelloWorldController {  
  
    @GetMapping("/helloWorld")  
    public String helloWorld() {  
        return "helloWorld";  
    }  
}
```



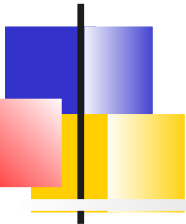
# Annotation *@RequestMapping*

---

## **@RequestMapping**

- Placée au niveau de la classe, elle indique que toutes les méthodes du contrôleur seront relatives à ce chemin
- Au niveau d'une méthode, l'annotation précise les attributs suivants :
  - **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  
Dans le cadre d'une API Rest il n'est pas nécessaire de préciser ces attributs car le format est toujours JSON





# Variantes *@RequestMapping*

---

Des variantes existent pour limiter à une seule méthode. Ce sont les annotation que l'on utilise en général dans une API Rest :

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



# Types des arguments de méthode

---

Une méthode annoté via *@\*Mapping* peut se faire injecter des arguments de type :

- La requête ou réponse HTTP (ServletRequest, HttpServletRequest, spring.WebRequest, ...)
- La session HTTP (HttpSession)
- La locale, la time zone
- La méthode HTTP
- L'utilisateur authentifié par HTTP (Principal)
- ..

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 de méthode

---

Ces annotations permettent d'associer un argument à une valeur de la requête HTTP. Les annotations principales utilisées dans le cadre d'une API Rest sont

- **@PathVariable** : Une partie de l'URI
- **@RequestParam** : Un paramètre HTTP (généralement passé par le caractère?)
- **@RequestHeader** : Une entête HTTP
- **@RequestBody** : Contenu de la requête au format Json qui sera converti en un objet Java
- **@RequestPart** : Une partie d'une requête multi-part



# Gabarits d'URI

---

Un gabarit d'URI permet de définir des variables. Ex : :

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

L'annotation **@PathVariable** associe la variable à un argument de méthode qui a le même nom

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



# Paramètres HTTP avec *@RequestParam*

---

```
@RestController
@RequestMapping("/pets")
public class PetController {

    // ...

    @GetMapping
    public Pet getPet(@RequestParam("petId") int petId) {
        Pet pet = this.clinic.loadPet(petId);

        return pet;
    }

    // ...

}
```

=> Exemple URL d'accès ***http://<server>/pets?petId=5***



# *@RequestBody* et convertisseur

---

L'annotation ***@RequestBody*** permet de convertir le corps JSON de la requête dans un objet métier.

- Elle est typiquement utilisée sur des méthodes annotées *@PostMapping*, *@PutMapping*, *@PatchMapping*
- La conversion, appelée dé-sérialisation, est effectuée par la librairie Jackson



# Exemple *@RequestBody*

---

```
@RestController
@RequestMapping("/pets")
public class PetController {

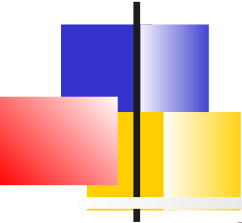
    // ...

    @PostMapping
    public Pet savePet(@RequestBody Pet pet) {
        Pet pet = this.clinic.savePet(pet);

        return pet;
    }

    // ...

}
```



# Types des valeurs de retours des méthodes

---

Les types des valeurs de retour possibles pour un contrôleur REST sont :

- Une classe **Modèle ou DTO** qui sera converti en JSON via la librairie Jackson.  
Le code retour est alors 200
- Un objet ***ResponseEntity<T>*** permettant de positionner les codes retour et les entêtes HTTP voulues





# Exemples

---

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

    @GetMapping(value="/{user}")
    public User getUser(@PathVariable Long user) {
        // ...
    }

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

    @DeleteMapping(value="/{user}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long user) {
        // ...
        return new ResponseEntity<>(HttpStatus.ACCEPTED);
    }

    @PostMapping
    public ResponseEntity<User> register(@RequestBody User user) {

        user = userRepository.save(user);

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



# APIs Rest avec SpringBoot

---

Spring MVC et les APIs REST

**Principes RESTFu**

Dé/Sérialisation avec Jackson

Exceptions, CORS et OpenAPI



# Auto-description

---

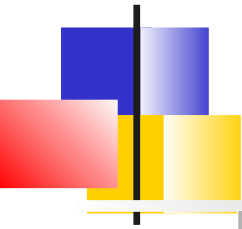
Une API bien conçue est compréhensible par un développeur sans qu'il ait à lire la documentation

- l'API est auto-descriptive.
- Une API se matérialise directement dans l'URL des requêtes HTTP envoyées au serveur exposant la ressource.

Exemple d'une requête HTTP sur une ressource de l'API de l'entreprise UBER :

*GET https://api.uber.com/<version>/partners/payments*

=> Un développeur sait intuitivement qu'il trouvera dans cette ressource les informations concernant les paiements reçues par le partenaire UBER (conducteur indépendant).



# Principe de Base

## *Une ressource $\leq \Rightarrow$ Une entité*

URI	GET	PUT	POST	DELETE
Collections : <i>http://api.example.com/produits</i>	Liste tous les produits	Remplace la liste de produits avec une autre liste	Crée une nouvelle entrée dans la collection.	Supprime la collection.
Element, <i>http://api.example.com/produits/5</i>	Récupère le produit d'ID 5	Remplace ou crée l'élément	Traite l'élément comme une collection et y ajoute une entrée	Supprime l'élément



# Règles de nommage

---

Pour les objets liés à l'entité principale. Il est recommandé d'utiliser une structure hiérarchique : /objets/{objet\_id}/sous\_objets

GET /calendar/meetings/{meeting\_id}/meeting\_room

L'opération sur la ressource peut être précisée si la méthode HTTP ne suffit pas :

GET /calendar/meetings/{meeting\_id}/attendees/search

# Paramètres de requête

I

Paramètre	Utilisation	Exemple
<b>Path Parameter</b>	Pour l'identification seulement : <ul style="list-style-type: none"><li>▪ Uniquement un ID, toujours suivant l'entité à laquelle il se réfère</li><li>▪ Paramètre obligatoire</li></ul>	{id} = 123 <a href="http://.../accounts/123/transactions">http://.../accounts/123/transactions</a>
<b>Query Parameter</b>	Pour la gestion de résultat - filtrer, trier, ordonner, grouper les résultats (paramètres courts) : <ul style="list-style-type: none"><li>▪ Paramètres techniques optionnels</li><li>▪ Valeurs sont définies et documentées dans la spécification de l'API</li></ul>	http://.../transactions ? from = NOW & sort = date:desc & limit = 50
<b>Header</b>	Pour la gestion du contexte d'application et de la sécurité <ul style="list-style-type: none"><li>▪ Utilisé par les navigateurs, les applications clientes et autres pour transmettre des informations sur le contexte de la demande</li><li>▪ Utilisé pour transmettre les paramètres d'authentification</li></ul> <p><u>NB</u> : Ne pas utiliser pour transmettre les paramètres fonctionnels</p>	Authorization : Bearer XXXXXXXX
<b>Body</b>	Pour les données fonctionnelles <ul style="list-style-type: none"><li>▪ Utilisé pour transmettre des informations fonctionnelles</li><li>▪ Doit être un objet JSON</li></ul>	{ "name": "phone", "category": "tech", "max_price": 45 }



# Codes retour

---

Les codes retours HTTP permettent de déterminer le résultat d'une requête ou d'indiquer une erreur au client.

Ils sont standards

- **1xx** : Information
- **2xx** : Succès
- **3xx** : Redirection
- **4xx** : Erreur client
- **5xx** : Erreur serveur



# Succès

---

Les codes retours « 2XX » sont les résultats des requêtes exécutées avec succès. Le code le plus courant est le code 200. Il en existe d'autres qui répondent à des cas plus précis.

- **200 - OK** : Toute requête réussie
- **201 - Created & Location** : Création d'un nouvel objet. Le lien ou l'identifiant de la nouvelle ressource est envoyé dans la réponse
- **204 - No content** : Mettre à jour ou supprimer un objet (avec une réponse vide)
- **206 - Partial Content** : Une liste paginée d'objets par exemple





# Erreurs client

---

Les codes retours « 4XX » indiquent que la requête envoyée par le client ne peut pas être exécutée par le serveur.

- **400 - Bad Request** : La requête est erronée. En général une mauvaise conversion
- **401 - Unauthorized** : La requête nécessite une authentification
- **403 - Forbidden** : Ressources non accessible pour l'utilisateur authentifié
- **404 - Not Found** : L'objet demandé n'existe pas
- **405 - Method Not Allowed** : L'URL est bonne mais la méthode HTTP
- **406 - Not Acceptable** : Les entêtes demandées ne peuvent pas être satisfaites. (Accept-Charset, Accept-Language)
- **409 - Conflict** : Par exemple : Tentative de création d'un nouveau utilisateur avec une adresse e-mail déjà existante
- **429 - Too Many Requests** : Le client a émis trop de requêtes dans un délai donné



# Erreurs serveur

---

Les codes retours « 5XX » indiquent que le serveur a rencontré une erreur. Les types d'erreurs serveur les plus fréquents sont :

- **501 - Not Implemented** : La méthode (GET, PUT, ...) n'est connue du serveur pour aucune ressource
- **502 - Bad Gateway ou Proxy Error** : La réponse du backend n'est pas comprise par l'API Gateway
- **503 - Service Unavailable** : API hors service, en maintenance, ...
- **504 - Gateway Time-out** : Timeout dépassé



# APIs Rest avec SpringBoot

---

Spring MVC et les APIs REST

Principes RESTFul

**Dé/Sérialisation avec Jackson**

Exceptions, CORS et OpenAPI



# Sérialisation JSON

---

Un des principales problématiques des back-end Spring et la conversion des objets du domaine au format JSON.

Des librairies spécialisés sont utilisées (Jackson, Gson), elles permettent de bénéficier de comportement par défaut

Mais, généralement le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bidirectionnelles entre classes du modèle
- Adaptation aux besoins de l'interface du front-end
- Optimisation du volume de données échangées
- Format des dates



# Comportement par défaut

---

```
public class Member {  
    private long id;  
    private String nom,prenom;  
    private int age;  
    private Date registeredDate;  
}
```

Devient :

```
{  
    "id": 5,  
    "nom": "Dupont",  
    "prenom": "Gaston",  
    "age": 71,  
    "registeredDate": 1645271583944 // Nombre de ms depuis le 1er Janvier 1970  
}
```



# Concepts Jackson

---

Avec Jackson, les sérialisations/désérialisations sont effectuées généralement par des ***ObjectMapper***

**// Sérialisation**

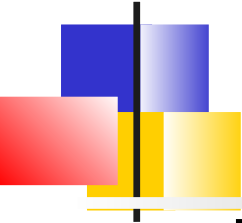
```
Member m = memberRepository.findById(4l) ;  
ObjectMapper objectMapper = new ObjectMapper() ;  
String jsonString = ObjectMapper.writeValueAsString(m) ;
```

...

**// Désérialisation**

```
String jsonString= "{\n\"id\" : 5,\n\" + ... + \"}\" ;  
Member m2 = ObjectMapper.readValue(jsonString) ;
```

Dans un contexte SpringBoot, on utilise rarement l'objet *ObjectMapper* directement ... mais on influence son comportement par des annotations.



# Solutions aux problématiques de sérialisation

---

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

- Créer des classes DTO spécifiques.  
La couche *Service* transforme les classes *Entité* provenant de la couche *Repository* en des classes Data Transfer Object encapsulant les données qui sont sérialisées par Jackson
- Utiliser les annotations proposées par Jackson  
Sur les classes DTO ou les classes *Entité*, utiliser les annotations Jackson pour s'adapter au besoin de la sérialisation
- Utiliser l'annotation *@JsonView*  
Le même objet *Entité* ou *Dto* peut alors être sérialisé différemment en fonction des cas d'usage
- Implémenter ses propres Sérialiseur/Désérialiserur.  
Spring propose l'annotation *@JsonComponent*



# Exemple DTO

---

```
@Service
public class UserService {
    @Autowired UserRepository userRepository;
    @Autowired RolesRepository rolesRepository;

    UserDto retrieveUser(String login) {
        User u = userRepository.findByLogin(login);
        List<Role> roles = rolesRepository.findByUser(u);

        return new UserDto(u,roles);
    }
}
```

---

```
public class UserDto {
    private String login, email, nom, prenom;
    List<Role> roles;

    public UserDto(User user, List<Role> roles) {
        login = user.getLogin(); email = user.getEmail();
        nom = user.getNom(); prenom = user.getPrenom();
        this.roles = roles;
    }
}
```





# Format de Dates

---

Pour avoir une représentation String des dates selon les bon vouloir du front-end, la solution est plus souple est d'utiliser @JsonFormat

```
public class Event {  
    public String name;  
    @JsonFormat(shape = JsonFormat.Shape.STRING,  
                pattern = "dd-MM-yyyy hh:mm:ss")  
    public Date eventDate;  
}
```



# Relations bidirectionnelles

## Le problème

---

```
public class User {  
    public int id;  
    public String name;  
    public List<Item> userItems;  
}  
  
public class Item {  
    public int id;  
    public String itemName;  
    public User owner;  
}
```

Lorsque *Jackson* sérialise l'une des 2 classes, il tombe dans une boucle infinie



# Relations bidirectionnelles

## Une solution

---

En annotant les 2 classes avec **@JsonManagedReference** et **@JsonBackReference**

```
public class User {  
    public int id;  
    public String name;  
  
    @JsonManagedReference  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonBackReference  
    public User owner;  
}
```

La propriété *userItems* est sérialisé mais pas *owner*



# Relations bidirectionnelles

## Une autre solution

En annotant les classes avec **@JsonIdentityInfo** qui demande à Jackson de sérialiser une classe juste avec son ID

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class User {...}
```

```
@JsonIdentityInfo(  
    generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")  
public class Item { ... }
```

Sérialisation d'un Item :

```
{  
  "id":2,  
  "itemName":"book",  
  "owner":  
    {  
      "id":1,  
      "name":"John",  
      "userItems":[2]  
    }  
}
```



# Relations bidirectionnelles

## Une autre solution

---

En annotant les classes avec **@JsonIgnore** on demande à Jackson de ne pas sérialiser une propriété

```
public class User {  
    public int id;  
    public String name;  
  
    public List<Item> userItems;  
}
```

```
public class Item {  
    public int id;  
    public String itemName;  
  
    @JsonIgnore  
    public User owner;  
}
```



# @JsonView

Des relations d'héritages peuvent être définies dans des classes statiques vides

```
public class CompanyViews {  
    public static class Normal{};  
    public static class Manager extends Normal{};  
    public static class HR extends Normal{};  
}
```

Les classes sont ensuite référencées via l'annotation *@JsonView* :

- Sur les classes du modèles :  
Quel attribut est sérialisé lorsque telle vue est activée ?
- Sur les méthodes des contrôleurs :  
Quelle vue doit être utilisée lors de la sérialisation de la valeur de retour de cette méthode ?



# @JsonView

## Annotations sur la classe du modèle

---

```
..  
public class Staff {  
  
    @JsonView(CompanyViews.Normal.class)  
    private String name;  
  
    @JsonView(CompanyViews.Normal.class)  
    private int age;  
  
    // 2 vues  
    @JsonView({CompanyViews.HR.class, CompanyViews.Manager.class})  
    private String[] position;  
  
    @JsonView(CompanyViews.Manager.class)  
    private List<String> skills;  
  
    @JsonView(CompanyViews.HR.class)  
    private Map<String, BigDecimal> salary;  
}
```



# Activation d'une vue

---

```
@RestController
public class StaffController {

    @GetMapping
    @JsonView(CompanyViews.Normal.class)
    public List<Staff> findAll() {
    }

    ...

    ObjectMapper mapper = new ObjectMapper();

    Staff staff = createStaff();

    try {
        String normalView =
            mapper.writerWithView(CompanyViews.Normal.class).writeValueAsString(staff);
```





# Autres 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

....



# Sérialiseur spécifique

---

L'annotation *Spring* **@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> {  
        // ...  
    }  
}
```



# APIs Rest avec SpringBoot

---

Spring MVC et les APIs REST  
Principes RESTFul  
Dé/Sérialisation avec Jackson  
**Exceptions, CORS et *OpenAPI***



# Personnalisation de la configuration Spring MVC

---

- Le personnalisation de la configuration par défaut de SpringBoot peut être effectuée en définissant un bean de type ***WebMvcConfigurer*** et en surchargeant les méthodes proposée.
- Dans le cadre d'une API Rest, une méthode permet de configurer le CORS<sup>1</sup>

1. CORS : *Cross-origin resource sharing*, une page web ne peut pas faire de requêtes vers d'autre serveurs que son serveur d'origine.



# Exemple Cross-origin

---

Le CORS peut se configurer globalement en surchargeant la méthode *addCorsMapping* de *WebMvcConfigurer* :

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

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



# 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, DocumentNotFound.class})
```

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

## **@Override**

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



# 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`

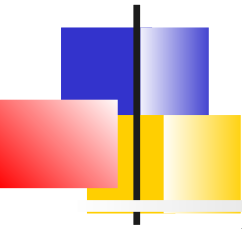


# Spring et les tests

---

## **Spring Test**

Apports de Spring Boot  
Tests auto-configurés



# Versions

## Spring/SpringBoot/JUnit

---

SpringBoot 1, Spring 4, JUnit4

Dernière version Septembre 2018

SpringBoot 2, Spring 5, JUnit5

Première version ~2018



# Rappels *spring-test*

---

Spring Test apporte peu pour le test unitaire

- **Mocking** de l'environnement en particulier l'API servlet ou Reactive
- Package **d'utilitaires** : *org.springframework.test.util*

Et beaucoup pour les tests d'intégration (impliquant un *ApplicationContext* Spring) :

- **Cache** du conteneur Spring pour accélérer les tests
- **Injection** des données de test
- Gestion de la **transaction** (roll-back)
- Des classes **utilitaires**
- **Intégration JUnit4 et JUnit5**



# Intégration JUnit

---

- Pour JUnit4 :

`@RunWith(SpringJUnit4ClassRunner.class)`

ou `@RunWith(SpringRunner.class)`

Permet de charger un contexte Spring, effectuer l'injection de dépendances, etc.

- Pour JUnit5 :

`@ExtendWith(SpringExtension.class)`

Permet aussi de charger un contexte Spring, effectuer l'injection de dépendances, etc.

Et en plus de l'injection de dépendance pour les méthodes de test, des conditions d'exécution en fonction de la configuration Spring, des annotations supplémentaires pour gérer les transactions



# Exemple JUnit5

---

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @Test
    void testMethod() {
        // test logic...
    }
}
```

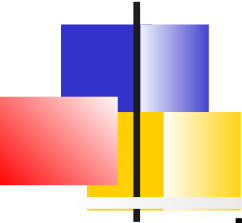




# SpringBoot et les tests

---

Rappels Spring Test  
**Apports de Spring Boot**  
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
- Annotations permettant des tests auto-configurés.  
Ex : Auto-configuration pour tester des *RestController* en isolation
- Annotation permettant de créer des beans Mockito



# @SpringBootTest

---

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

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



# Équivalence

---

```
// Annotations SpringBootTest
```

```
@RunWith(SpringRunner.class)
```

```
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
```

```
public class SpringBootTestApplicationTests {
```

```
// Annotations classiques
```

```
@RunWith(SpringRunner.class)
```

```
@SpringApplicationConfiguration(classes =  
    SprintBootTestApplication.class)
```

```
@WebAppConfiguration
```

```
public class SpringBootTestApplicationTests
```



# Attribut Class

---

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

Exemple :

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



# Attribut *WebEnvironment*

---

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

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



# Détection de la configuration

---

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

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

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

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





# 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;

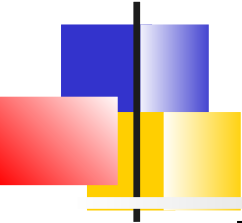
    @Test
    public void exampleTest() {
        // RemoteService has been injected into the reverser bean
        given(this.remoteService.someCall()).willReturn("mock");
        String reverse = reverser.reverseSomeCall();
        assertThat(reverse).isEqualTo("kcom");
    }
}
```



# 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

---

**@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

---

```
@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)

---

```
@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

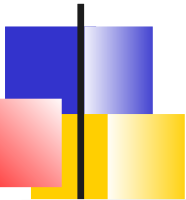
---

```
@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*.



# Example

---

```
@RestClientTest(RestService.class)
public class RestserviceTest {
    @Autowired
    private MockRestServiceServer server;
    @Autowired
    private ObjectMapper objectMapper;
    @Autowired
    private RestService restService;

    @BeforeEach
    public void setUp() throws Exception {
        Member aMember = ...
        String memberString = objectMapper.writeValueAsString(aMember);

        this.server.expect(requestTo("/members/1"))
            .andRespond(withSuccess(memberString, MediaType.APPLICATION_JSON));
    }

    @Test
    public void whenCallingGetMember_thenOk() throws Exception {
        assertThat(restService.getMember(1)).extracting("email").isEqualTo("d@gmail.com");
    }
}
```



# 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



# Vers la production

---

**Monitoring avec actuator**  
Déploiement



# 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





# Vers la production

---

Monitoring avec actuator  
**Déploiement**



# Introduction

---

Plusieurs alternatives pour déployer une application Spring-boot :

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



# Application stand-alone

---

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

```
mvn package
```

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

Pour l'exécuter :

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



# Fichier Manifest

---

Manifest-Version: 1.0

Implementation-Title: documentService

Implementation-Version: 0.0.1-SNAPSHOT

Archiver-Version: Plexus Archiver

Built-By: dthibau

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

Implementation-Vendor-Id: org.formation.microservice

Spring-Boot-Version: 1.3.5.RELEASE

Created-By: Apache Maven 3.3.9

Build-Jdk: 1.8.0\_121

Implementation-Vendor: Pivotal Software, Inc.

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



# Création de war

---

Pour créer un war, il est nécessaire de :

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

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



# Exemple

---

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



# Création de service Linux

---

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

=> target/artifactId.jar is executable !

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



# Cloud

---

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

La documentation de référence offre du support pour :

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





# Exemple CloudFoundry/Heroku

---

## Cloud Foundry

```
cf login
```

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

## Heroku

Mise à jour d'un fichier Procfile :

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

```
git push heroku master
```



# Annexe



# Mécanisme auto-configuration



# Auto-configuration

---

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

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

Ainsi au démarrage de Spring Boot, les conditions sont évaluées et si elles sont respectées, les beans d'intégration correspondant sont instanciés et configurés.



# Annotations conditionnelles

---

Les conditions peuvent se basées sur :

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



# Exemple Apache SolR

---

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

    private final SolrProperties properties;

    private SolrClient solrClient;

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

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

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



# Conséquences

---

Le starter *Solar* tire

- Les classes de Configuration conditionnelles
- Les librairies Sonar

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

```
@Component
public class MyBean {

    private SolrClient solrClient;

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



Format *.yml*





# 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
```

Fournit les clés suivantes :

```
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
```