



Spring Boot

David THIBAU – 2021

david.thibau@gmail.com



Agenda

- **Introduction**

- Historique
- IoC et Dependency Injection
- Évolutions du Framework

- **Spring Core**

- Présentation du Conteneur IoC
- BeanFactory et ApplicationContext
- Les Beans
- Autres fonctionnalités cœur

- **Annotations**

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

- **Spring AOP**

- Concepts de l'AOP
- Caractéristiques de Spring AOP
- AOP via XML
- AOP via @AspectJ

- **Spring Boot**

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

- **Spring et la persistance**

- Spring Data
- JPA
- MongoDB

- **Applications Web**

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

- **Spring Security**

- Principes de Spring Security
- Modèle stateful
- Modèle stateless avec OAuth2



Agenda

- **Spring et les tests**
 - Spring Test
 - Apports de SpringBoot
 - Les tests auto-configurés
- **Messaging Spring**
 - Support Spring pour les message broker
- **Reactive Spring**
 - Programmation réactive
 - Le projet Reactor
 - Spring Data reactive (Mongodb, Cassandra)
 - Spring WebFlux
- **Vers la production**
 - Actuator
 - Déploiement



Introduction

Historique

IoC et Dependency Injection
Evolutions du framework



Historique et version

- ❖ Spring est un projet **OpenSource** avec un modèle économique basé sur le service (Support, Conseil, Formation, Partenariat et certifications)
- ❖ La société **SpringSource¹** fondé par les créateur de Spring Rod Johnson et Juergen Hoeller a été racheté par **VmWare**, puis intégré dans la joint-venture **Pivotal Software**



Spring et Java EE

- ❖ Spring a été initié en opposition à la spécification Java EE
- ❖ Les fondateurs de la solution ont toujours clamé que Spring apportait tous les services d'un serveur Java EE sans la lourdeur de la spécification.
D'où le nom de *conteneur léger*
- ❖ Après l'abandon de la spécification Java EE par Oracle, Spring est le framework Java le plus utilisé pour construire les applications back-end accessible via HTTP.



Les 7 commandements

- ❖ Les 7 commandements de Spring
 - **JavaEE est trop complexe**, il devrait être plus facile d'utilisation
 - Il est toujours plus avantageux d'utiliser des **interfaces** plutôt que des classes. Spring facilite l'utilisation d'interfaces.
 - La « **bonne** » **programmation objet** est bien plus importante que les technologies d'implémentation comme J2EE.
 - Les **exceptions contrôlées sont trop utilisées en Java**. Un framework ne doit pas forcer les développeurs de traiter les exceptions sur lesquelles on ne peut rien.
 - La **testabilité** d'un programme est essentielle. Spring doit permettre du code facile à tester.
 - Le code applicatif **ne doit pas être dépendant du framework**
 - Spring ne doit pas concurrencer les bonnes solutions existantes mais plutôt **favoriser leurs intégration** (AspectJ, JDO, TopLink, Hibernate)
- ❖ Livre de référence : *Expert One-on-One J2EE Design and Development* (Rod Johnson).



Bénéfices

- ❖ **Organisation** élégante du tiers métier (Spring effectuant le travail de « plomberie »).
- ❖ Un **format unifié** pour la configuration de tous les couches de l'application
- ❖ Adoption de **bonnes pratiques** de programmation en particulier avec les interfaces.
- ❖ **Dépendance** sur le moins grand nombre d'APIs possible.
- ❖ **Testabilité** : Spring facilite les tests unitaires et les tests d'intégration.
- ❖ Container Spring fournit un ensemble de **services techniques** (Gestion des transactions, sécurité par exemple).
- ❖ **Évolutivité** : les couche d'abstraction de Spring permettent de facilement switcher d'une technologie à une autre (exemple JDBC à JPA)
- ❖ Utilisation de **POJOs (Plain Old Java Objects)**. Le code de POJO ne contient que la logique métier.

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.



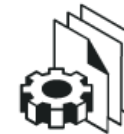
Serverless

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.



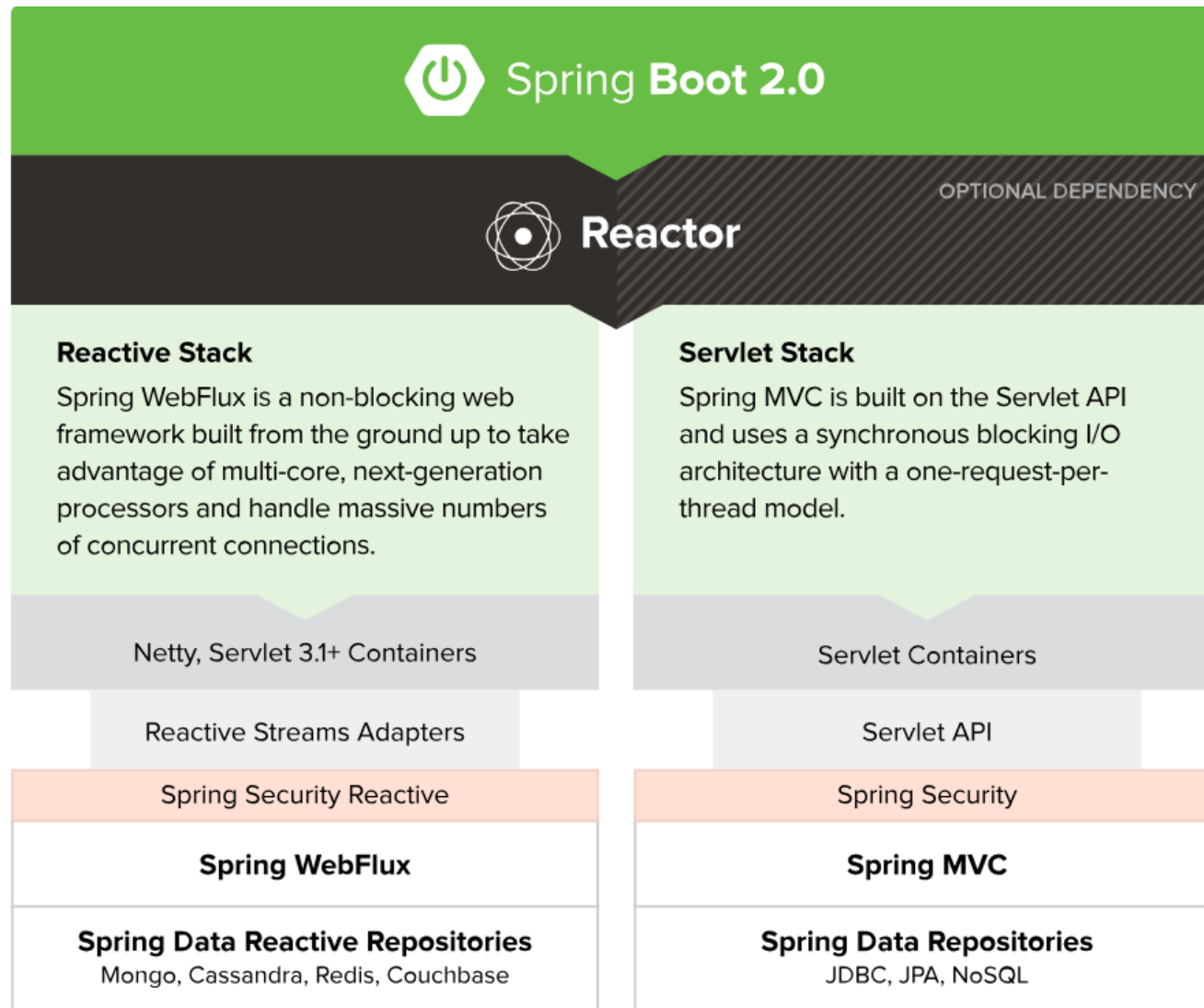
Projets Spring

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 (plomberie)
- ✓ Être portable : Les pré-requis étant disposer d'une JVM ou d'un environnement Cloud
- ✓ Être productif : Fournir les outils de productivité aux développeurs

Stacks Web





Modules cœur

Conteneur cœur :

- Cœur et Beans : IoC et Injection de dépendances
- Contexte : Accès aux beans, chargement de ressources, internationalisation, Validation, Data Binding
- Expression Language : Langage de requête et de manipulations des beans

AOP, Aspects et Instrumentation :

Intercepteurs, profilers

Test : Intégration avec junit et TestNG



Introduction

Historique
IoC et Dependency Injection
Évolutions du framework



Pattern loc

❖ 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 ?



Illustration

- ❖ On veut implémenter une fonction métier permettant de lister tous les films d'un metteur en scène donné. Cette classe métier *MovieLister* s'appuie sur un objet *finder* qui permet de récupérer tous les films d'une base de données :

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

- ❖ On veut également que la méthode *moviesDirectedBy()* soit complètement indépendante de la façon dont les films sont stockés. Ainsi la méthode s'appuie sur un objet *finder* implémentant une interface :

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



Implémentation ?

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

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

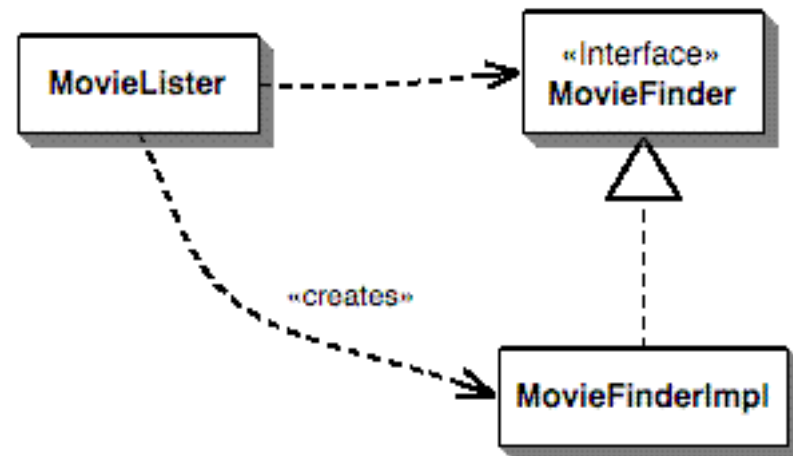

Dépendance

=> La classe *MovieLister* est dépendante de l'interface **et** de l'implémentation !!

Le but serait qu'elle ne soit dépendante que de l'interface.

Mais alors comment indiquer l'implémentation à exécuter.

=> Solution : **Pattern IoC**





IoC et framework

- ❖ *Framework* : Code qui appelle vos méthodes, agit comme un programme principal séquençant l'appel à des méthodes.
- ❖ Cette « **inversion de contrôle** » rend les frameworks comme des squelettes de programmes extensibles que l'utilisateur remplit avec ses propres méthodes.
- ❖ Le pattern *Inversion of Control* est ce qui différencie un framework d'une librairie :
 - Une librairie est essentiellement un ensemble de fonctions que vous pouvez appeler.
 - Un framework est un squelette de programme qui appelle vos fonctions.

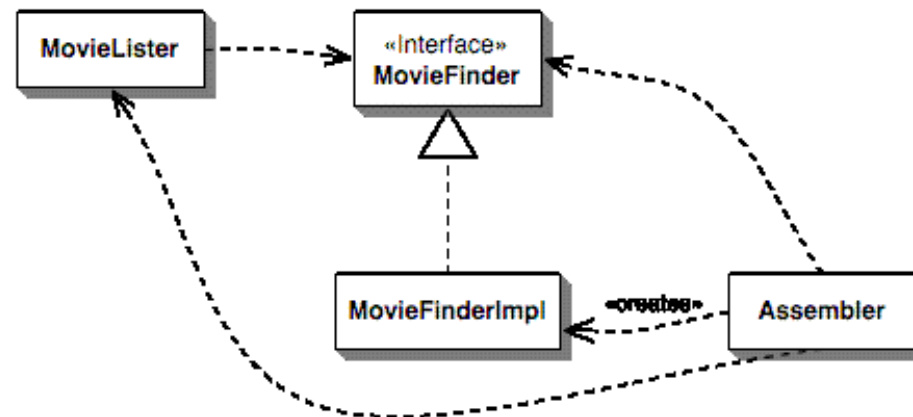


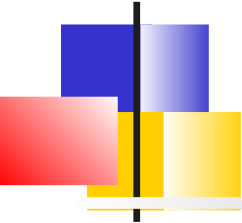
IoC versus Dependency Injection

- ❖ 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.

L'assembleur

- Dans le cas du pattern d'injection de dépendance, un autre objet : l'assembleur, est responsable d'initialiser les champs de la classe avec les implémentations appropriées
- Cet assemblage pourrait être fait soit au build, soit à l'exécution





Types d'injection de dépendances

- ❖ Il y a 3 principaux types d'injections de dépendances :
 - **Injection par constructeur** : type 3 IoC (constructor injection)
 - **Injection par méthode setter** : type 2 IoC (setter injection)
 - **Injection par méthode** : type 1 IoC (interface/méthode injection)



Injection par constructeur

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

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

- ❖ L'objet *finder* sera également géré par le container 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 interface¹

```
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 container

- ❖ Le code permettant de configurer le container est généralement effectué dans une classe différente.
 - Par exemple, toute personne qui voudrait utiliser la classe *MovieLister* devrait écrire un code approprié configurant le container en n'utilisant que ses propres implémentations.
- ❖ La bonne pratique est de centraliser ce type d'information dans des **fichiers de configuration** facilement lisible (XML ou classe Java)



Configuration

```
<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



Introduction

Historique
IoC et Dependency Injection
Évolutions du framework



Versions Spring Framework

- ❖ 1.2 Mars 2004 : IoC container, AOP, MVC, Configuration XML, DAO : TopLink, JDO 2.0, Hibernate 3.0.3 support
- ❖ 2.0 Octobre 2006 : XML namespaces, AOP avec AspectJ , Scopes spécifiques
- ❖ 2.5 Novembre 2007 : Support Java 6 et alignement sur Java EE 5, Configuration via Annotations, JUnit4 et TestNG
- ❖ 3.0 Décembre 2009 : Java 5 minimum, Spring Expression Language, javax.Validation, Alignement sur Java EE6, Intégration complète de JavaConfig, Support BD embarquées
- ❖ 4.0 Décembre 2013 : Java 6 minimum, Support de Java8, Alignement Java EE7, Groovy, Websocket
- ❖ 4.3 Juin 2016, Facilité de configuration annotations supplémentaires
- ❖ 5.0 2017 : Java 8+, Programmation réactive avec *Reactor*, Support avec Kotlin 1.1+, JUnit 5, Java EE 8 API level



Versions SpringBoot

Avril 2014, spring boot 1.x pour Spring4

Mars 2018, spring boot 2.x pour Spring5



Types de configuration

- ❖ Actuellement 2 choix sont possibles pour configurer le container :
 - **Fichier de configuration XML :**
Changement sans recompilation, Utilisation de *namespace* spécifique
 - **Source Java :**
 - Classes de configuration
 - Annotations dans les beans



Configuration XML

```
<beans>
  <bean id="MovieLister" class="spring.MovieLister">
    <property name="finder">
      <ref local="MovieFinder"/>
    </property>
  </bean>
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
    <property name="filename">
      <value>movies1.txt</value>
    </property>
  </bean>
</beans>
```



Configuration via annotations

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



Configuration et injection via annotations

@Service

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

@Autowired

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

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



SpringCore

Spring : le conteneur léger

BeanFactory et ApplicationContext

Beans

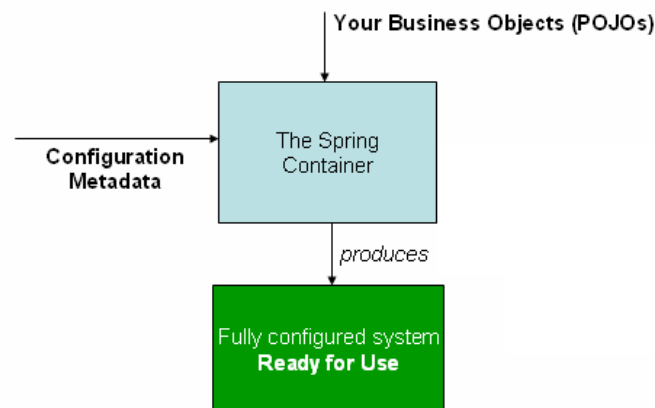
Autres fonctionnalités coeur

Processus de démarrage

Les objets définissent leurs dépendances (c'est-à-dire les autres objets avec lesquels ils travaillent) via des arguments de constructeur, des arguments d'une méthode d'usine ou des propriétés.

Le conteneur injecte ensuite ces dépendances lorsqu'il crée le bean.

Il peut en plus ajouter des services techniques aux beans qu'il crée. (Sécurité, transaction, etc..)



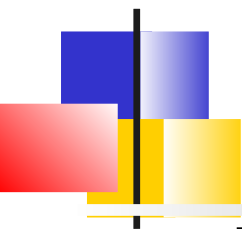


Méta-données de configuration

Les métadonnées de configuration sont traditionnellement fournies dans un format XML simple et intuitif.

Ce format est encore utilisé dans la documentation de référence même si dans la pratique, on utilise plutôt :

- Annotations (Spring 2.5)
- Classes de configuration Java (Spring 3.0)



Chargement de la config

Le chargement de la configuration peut s'effectuer de différentes façons :

- Lecture d'un ou plusieurs fichiers XML à partir du classpath ou d'un système de fichier
- Scan de packages à la recherche d'annotations @Configuration et @Bean
- Lecture d'un ou plusieurs fichiers properties ou yml pour spécifier les valeurs des beans.
- Fichiers Groovy
- ...



Utilisation du container

Le chargement de la configuration
retourne un objet de type
ApplicationContext : le conteneur.

La principale opération est alors :

```
T getBean(String name, Class<T> requiredType)
```



Exemples

// Configuration des beans

```
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml",  
    "daos.xml");
```

// Récupération d'une instance

```
PetStoreService service = context.getBean("petStore", PetStoreService.class);
```

// Utilisation

```
List<String> userList = service.getUsernameList();
```

-- Via Groovy

```
ApplicationContext context = new  
    GenericGroovyApplicationContext("services.groovy", "daos.groovy");
```

-- Avec Kotlin

```
val context = GenericApplicationContext()  
GroovyBeanDefinitionReader(context).loadBeanDefinitions("services.groovy",  
    "daos.groovy")  
context.refresh()
```



SpringCore

Spring : le conteneur léger
BeanFactory et ApplicationContext
Beans
Autres fonctionnalités coeur



API

Les packages ***org.springframework.beans*** et ***org.springframework.context*** constituent la base du conteneur IoC.

L'interface ***BeanFactory*** contient les mécanismes de configuration des beans.

ApplicationContext est une sous-interface de ***BeanFactory*** qui ajoute des fonctionnalités *d'entreprise*:

- Intégration plus facile avec les fonctionnalités AOP de Spring.
Interfaces *BeanPostProcessor* et *BeanFactoryPostProcessor*
- Gestion des ressources de message (i18n)
- Publication d'événement
- Contextes spécifiques à la couche d'application tels que *WebApplicationContext* à utiliser dans les applications Web.



BeanFactory vs ApplicationContext

Table 9. Feature Matrix

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Integrated lifecycle management	No	Yes
Automatic <code>BeanPostProcessor</code> registration	No	Yes
Automatic <code>BeanFactoryPostProcessor</code> registration	No	Yes
Convenient <code>MessageSource</code> access (for internationalization)	No	Yes
Built-in <code>ApplicationEvent</code> publication mechanism	No	Yes



Support pour l'internationalisation

- ❖ Un bean nommé ***messageSource*** implémentant l'interface ***MessageSource*** est déclaré dans la configuration. Il est automatiquement détecté lors du chargement de l'*ApplicationContext*
- ❖ Les méthodes disponibles de type *getMessage()* permettent de récupérer un message localisé.
- ❖ L'implémentation classique de *MessageSource* est ***org.springframework.context.support.ResourceBundleMessageSource***



Example

```
<beans>
  <bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="baseName" value="WEB-INF/test-messages"/>
  </bean>
  <bean id="example" class="com.foo.Example">
    <property name="messages" ref="messageSource"/>
  </bean>
</beans>

public class Example {
  private MessageSource messages;
  public void setMessages(MessageSource messages) {
    this.messages = messages;
  }
  public void execute() {
    String message = this.messages.getMessage("argument.required",
                                              new Object [] {"userDao"}, "Required", null);
    System.out.println(message);
  }
}
```



Modèle événementiel

- ❖ Les beans implémentant l'interface *ApplicationListener* reçoivent les événements de type *ApplicationEvent* :
 - ***ContextRefreshedEvent*** : L'*ApplicationContext* est initialisé ou rechargé
 - ***ContextClosedEvent*** : L'*ApplicationContext* est fermé et les beans détruits
 - ***RequestHandledEvent*** : Spécifique au *WebApplicationContext*, indique qu'une requête HTTP vient d'être servie.
- ❖ Il est également possible de définir ses propres événements et les projets Spring utilisent également ce modèle événementiel



SpringCore

Spring : le conteneur léger
BeanFactory et *ApplicationContext*
Beans
Autres fonctionnalités cœur



BeanDefinition

A l'intérieur du conteneur, les beans sont représentés par la classe `BeanDefinition` qui encapsule :

- Le nom qualifié de la classe associée
- La configuration comportementale du bean (scope, méthodes de call-back, ...).
- Les références aux autres beans (les collaborateurs ou dépendances)
- D'autres données de configuration (le dimensionnement d'un pool par exemple)



Nommage des beans

Un bean a un ou plusieurs identifiants (unique à l'intérieur du conteneur).

La convention est d'utiliser la convention Java standard pour les attributs d'une instance.

Les autres noms sont appelés des alias.



Instanciación

Les BeanDefinition sont utilisés pour instancier les beans

La propriété Class associée peut être utilisée de 2 façons :

- Directement pour instancier le constructeur réflexivement
- Pour spécifier la classe contenant la méthode factory



Cycles de vie

Les beans sont de simples POJOS qui peuvent avoir 3 cycles de vie (ou scope) :

- **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des services « stateless »
- **Prototype** : A chaque fois que l'objet est demandé via son nom, une instance est créée.
- **Custom object “scopes”** : Leur cycle de vie est généralement scopé sur un autre objets.
Ex : La requête HTTP, la session, une transaction BD



Interfaces *Aware*

- ❖ Les beans peuvent implémenter des interfaces de type *Aware* permettant de manipuler le conteneur :
 - Chargeur de ressources (***ResourceLoaderAware***)
 - Générateur d'évènement (***ApplicationEventPublisherAware***)
 - Gestionnaire de message (***MessageSourceAware***)
 - L'application context (***ApplicationContextAware***)
 - Le ServletContext dans le cas d'une application web (***ServletContextAware***)



Interfaces PostProcessor

Dans le cycle de construction du conteneur, Spring offre 2 hooks :

BeanFactoryPostProcessor permet des modifications custom des définitions de beans

Utile par exemple pour les fichiers de configuration personnalisés destinés aux administrateurs système qui remplacent les propriétés par défaut du bean

BeanPostProcessor permet de modifier directement l'instance du bean.

Typiquement utilisé pour appliquer des aspects via des annotations. (Voir AOP)



Points d'extensions du conteneur

- ❖ Sans implémenter des sous-classes de *BeanFactory* ou *ApplicationContext*, il est possible d'étendre les fonctionnalités du conteneur grâce à ses points d'extension.
 - Pour rajouter du code spécifique pendant l'initialisation des beans, il est possible de plugger une ou plusieurs implémentation de l'interface ***BeanPostProcessor***
 - Pour modifier la définition des beans, il est possible de plugger des implémentations de ***BeanFactoryPostProcessor***



BeanPostProcessor

- ❖ L'interface *BeanPostProcessor* comporte deux méthodes de call-back :
 - ***Object postProcessBeforeInitialization(Object bean, String beanName)***
appelée avant les méthodes d'initialisation du bean
 - ***Object postProcessAfterInitialization(Object bean, String beanName)***
appelée après les méthodes d'initialisation du bean
- ❖ Lors de la présence de plusieurs *BeanPostProcessor*, l'ordre d'appel aux méthodes de call-back peut être maîtrisé :
 - L'attribut ***order*** est précisé dans la configuration
 - Le *BeanPostProcessor* implémente également l'interface ***org.springframework.core.Ordered***



Enregistrement *BeanPostProcessor*

- ❖ Dans le cas d'une *ApplicationContext*, le conteneur détecte automatiquement les classes implémentant l'interface *BeanPostProcessor*.
Les *BeanPostProcessors* se déclarent alors comme un Bean classique :

```
<bean class="scripting.InstantiationTracingBeanPostProcessor"/>
```

- ❖ Lors de l'utilisation d'un *BeanFactory*, il est par contre nécessaire d'enregistrer les *BeanPostProcessor* :

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);
```

```
BeanPostProcessor postProcessor = new MyPostProcessor();
```

```
factory.addBeanPostProcessor(postProcessor);
```



BeanFactoryPostProcessor

- ❖ L'interface *BeanFactoryPostProcessor* permet de modifier les méta-données avant que Spring n'instancie les beans. Elle comporte une méthode :

`postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)`

- ❖ De la même façon que les *BeanPostProcessor*, les *BeanFactoryPostProcessor* peuvent être ordonnés.
- ❖ Spring offre plusieurs implémentations utiles de cette interface en particulier pour traiter les annotations Java5.



Exemple du *PropertyPlaceholderConfigurer*

```
<bean
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="locations">
    <value>classpath:com/foo/jdbc.properties</value>
  </property>
</bean>
<bean id="dataSource" destroy-method="close"
class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${jdbc.driverClassName}"/>
  <property name="url" value="${jdbc.url}"/>
  <property name="username" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
</bean>
```

❖ Fichier *jdbc.properties*

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```



Rappel séquence d'instanciation d'un bean

- Si le bean implémente *BeanNameAware*, appel de la méthode ***setBeanName()***
- Si le bean implémente *BeanClassLoaderAware* appel de ***setBeanClassLoader()***
- Si le bean implémente *BeanFactoryAware*, appel de ***setBeanFactory()***
- Si le bean implémente *ResourceLoaderAware*, appel de ***setResourceLoader()***
- Si le bean implémente *ApplicationEventPublisherAware*, appel de ***setApplicationEventPublisher()***
- Si le bean implémente *MessageSourceAware*, appel de ***setMessageSource()***
- Si le bean implémente *ApplicationContextAware* appel de ***setApplicationContext()***
- Si le bean implémente *ServletContextAware*, appel de ***setServletContext()***
- Appel des méthodes ***postProcessBeforeInitialization()*** des *BeanPostProcessor*
- Si le bean implémente *InitializingBean*, appel de ***afterPropertiesSet()***
- Si *init-method* défini, appel de la **méthode d'initialisation spécifique**
- Appel des méthodes ***postProcessAfterInitialization()*** des *BeanPostProcessor*



SpringCore

Spring : le conteneur léger
BeanFactory et *ApplicationContext*
Beans

Autres fonctionnalités cœur



Accès aux ressources

- ❖ Spring définit l'interface **Resource** qui offre des méthodes utiles pour le chargement de ressource URL (*getURL()*, *exists()*, *isOpen()*, *createRelative()*, ...)
- ❖ Plusieurs implémentations utiles sont également fournies :
 - *ClassPathResource* : Chargée à partir d'un classpath
 - *ServletContextResource* : Chargée à partir d'un chemin relatif à la racine d'une application web



ResourceLoader

❖ Chaque contexte Spring implémente l'interface *ResourceLoader* qui détermine la stratégie de chargement des ressources.

❖ Par exemple, le code

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

- retourne un *ClassPathResource* pour une *ClassPathXmlApplicationContext*
- retourne un *ServletContextResource* pour un contexte de type *WebApplicationContext*



Validation, Data Binding et Conversion de type

Tous les projets Spring offre du support pour :

- La **validation** des classes de données. Principalement par l'API Java Bean Validation
- Le **Data Binding** permet de facilement associé les entrées utilisateur à des classes du modèle
- La **conversion de type** qui permet de construire les objets du modèle facilement



SpEL

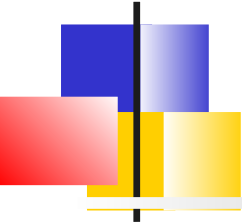
Spring Expression Language (SpEL) est une syntaxe permettant la recherche dans les graphes d'objet issus du conteneur.

Il permet l'accès aux propriétés mais également l'invocation de méthodes

```
@lombok.Data
public class FieldValueTestBean {

    // Injection de valeur
    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

}
```



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 des annotations dans la classe du bean.
- ❖ Chaque approche a ses avantages et ses inconvénients
 - Les annotations profitent de leur contexte de placement ce qui rend la configuration plus concise
 - XML permet d'effectuer le câblage sans nécessiter de recompilation du code source
 - Les classes annotées ne sont plus de simple POJOs
 - Avec les annotations, la configuration est décentralisée et devient plus difficile à contrôler
- ❖ Les annotations sont traitées avant la configuration XML ainsi la configuration XML peut surcharger la configuration par annotations



Concept

Dés 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** qui définissent l'instanciation et la configuration des objets gérés par Spring

@Configuration

```
public class AppConfig {
```

@Bean

```
public MyService myService() {  
    return new MyServiceImpl();  
}  
}
```



Composition de configuration

L'annotation **@Import** permet d'importer une autre classe
@Configuration

```
@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(); }  
}
```

```
-
```

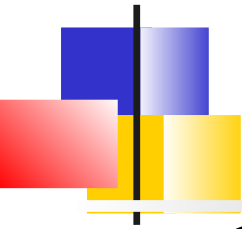
```
ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(ConfigB.class);
```



AnnotationConfigApplicationContext

AnnotationConfigApplicationContext traite les annotations *@Configuration* mais également *@Component*, *@Inject*, *@Qualifier*, *@Autowired*, etc..

- Les classes *@Configuration* et leurs méthodes annotées *@Bean* enregistrent des définitions de bean.
- Les classes *@Component* enregistrent également des définitions de beans
- Les annotations *@Autowired*, *@Inject*, *@Resource* sont utilisées pour l'injection de dépendances



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, c'est la 2ème alternative qui est utilisée.



Construction

Utilisation de classe *@Configuration*

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

Utilisation de classes *@Component*

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(MyServiceImpl.class,  
    Dependency1.class, Dependency2.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```



Méthode *scan()*

La méthode ***scan()*** permet de parcourir un package (et ses sous-packages) particulier :

```
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() {
        return new TransferServiceImpl();
    }
}
```



Attributs de *@Bean*

@Bean définit 3 attributs :

name : les alias du bean

init-method : méthodes de call-back d'initialisation

destroy-method : Call-back de destruction

@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();  
    }  
}
```



Injection de dépendances

L'expression de dépendances s'effectue tout simplement par des appels de méthodes

@Configuration

```
public class AppConfig {  
    @Bean  
    public Foo foo() {  
        return new Foo(bar());  
    }  
    @Bean  
    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*

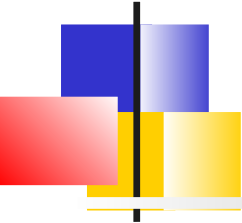
@EnableWebMvc : Permet Spring MVC dans une application

@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

Les annotations peuvent également être utilisées pour déclarer des définitions de beans généralement applicatifs

L'annotation principale pour la définition d'un bean applicatif 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 correspondant à des beans

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

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

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



Stereotypes

Spring introduit d'autres stéréotypes :

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

@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.

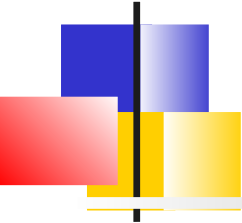
Par ex : Un comportement transactionnel à tous les composants @Service



@Scope

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

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



Configuration via les annotations

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



@Autowired

L'annotation **@Autowired** se place sur des méthodes *setter*, des méthodes arbitraires, des constructeurs

Il demande à Spring d'injecter un bean du type de l'argument

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 movieCatalog, CustomerPreferenceDao customerPreferenceDao)
    {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```




Examples

```
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;  
    // ...  
}
```



@Qualifier

@Qualifier permet de sélectionner un candidat à l'auto-wiring parmi plusieurs possibles

L'annotation prend comme attribut une String dont la valeur doit correspondre à un élément de configuration d'un bean



Exemple

```
public class MovieRecommender {  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
    // ...  
}  
---  
<beans>  
    <context:annotation-config/>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="main"/>  
        <!-- .... --></bean>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="action"/>  
        <!-- .... --></bean>  
    <bean id="movieRecommender" class="example.MovieRecommender"/>
```



@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



Examples

```
public class MovieRecommender {  
  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao cpDao;  
  
    @Resource  
    private ApplicationContext context;  
  
    public MovieRecommender() {  
    }  
    // ...  
}
```



Méthodes de call-back

Spring supporte les call-back

@PostConstruct et **@PreDestroy**

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



Annotations JSR 330

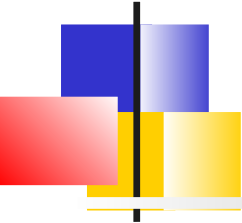
Équivalence

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 des valeurs dans les propriétés des beans via des annotations :

- **@PropertySource** permet d'indiquer un fichier *.properties* permettant de charger des valeurs de configuration
- **@Value** permet d'initialiser les propriétés des beans avec une expression SpEl
- Cela nécessite la présence d'un bean **PropertySourcesPlaceholderConfigurer**



Exemple

@Configuration

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

public class AppConfig {

@Value("\${my.property:0}")

Integer myIntProperty ;

@Autowired

Environment env;

@Bean

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

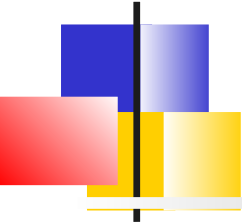
}

@Bean

public TestBean testBean() {
 TestBean testBean = new TestBean();
 TestBean.setIntProperty(myIntProperty) ;
 testBean.setName(env.getProperty("testbean.name"));
 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** : Groupe nommé de Beans, les beans sont enregistrés seulement si le profil est activé



Exemple

@Configuration

@Profile("development")

```
public class StandaloneDataConfig {

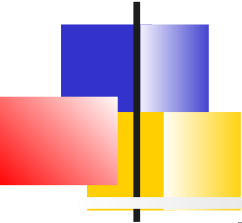
    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }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

@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");
    }
}
```



Propriétés spécifiques via **.properties*

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

application-{profile}.properties



Propriété spécifiques à un profil via fichier *.yml*

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

```
server:  
  address: 192.168.1.100
```

```
spring:  
  profiles: development
```

```
server:  
  address: 127.0.0.1
```

```
spring:  
  profiles: production
```

```
server:  
  address: 192.168.1.120
```



Inclusion de profils

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

Par exemple :

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




Activation d'un profil

Programmatically :

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

Propriété Java

-Dspring.profiles.active="profile1,profile2"



Spring AOP

Concepts de l'AOP

Caractéristiques de Spring AOP

AOP via XML

AOP via annotations *@AspectJ*

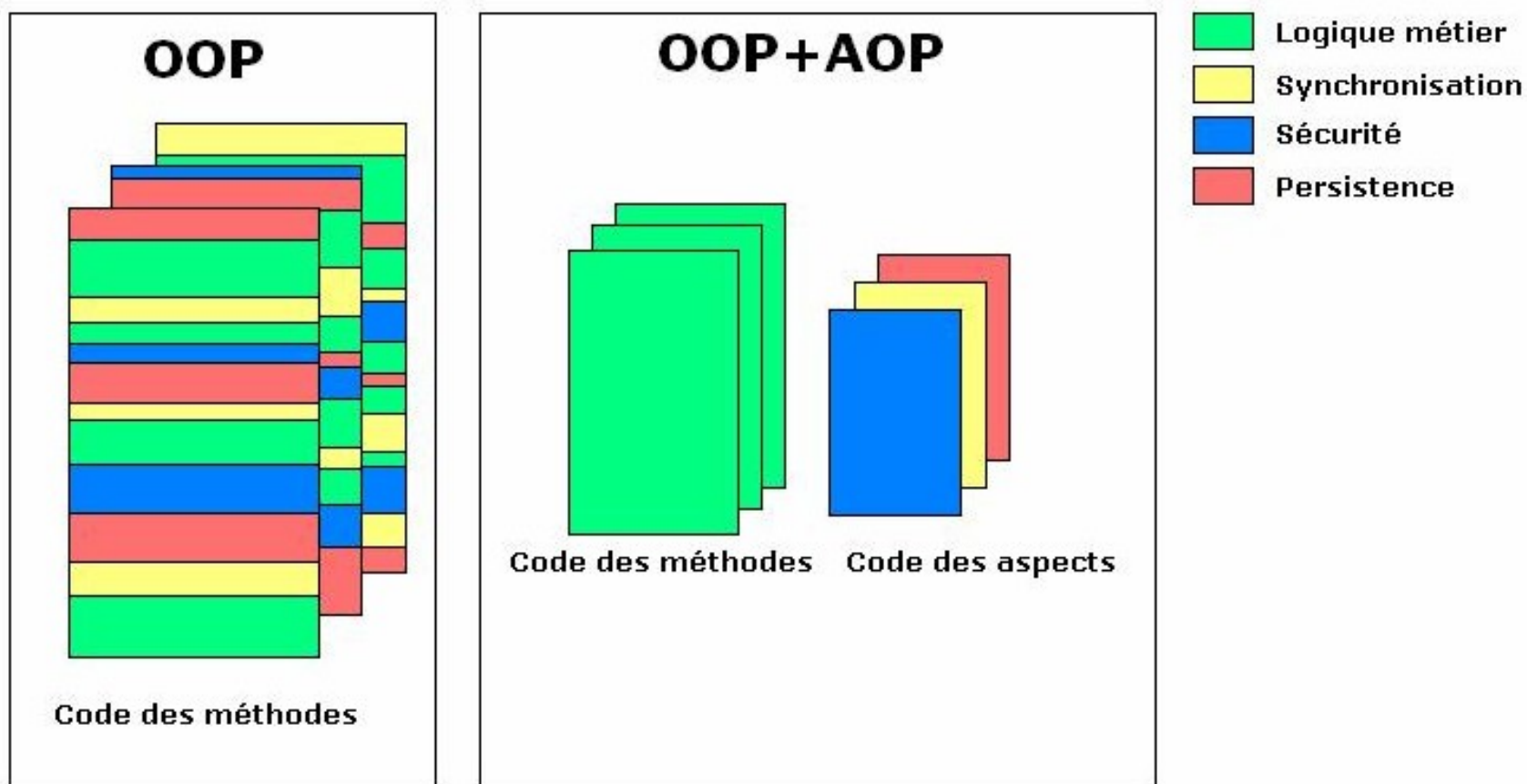


Introduction

- ❖ L'AOP (Aspect Oriented Programming) permet de contourner certaines limites de la programmation objets (OOP) :
 - Le code source des classes écrites dans un langage objet comporte beaucoup de lignes de code consacrées à des **crosscutting concerns** comme la production de trace, le traitement d'exception, la vérification de paramètre entrant, etc.
- ❖ Ces lignes de code ont tendance à être dispersées dans les méthodes des classes ne favorisant
 - ni la réutilisation : couplage avec les crosscutting concerns
 - ni l'évolutivité : Un changement de technologie pour un crosscutting concern impacte toutes les classes.
- ❖ L'AOP permet de mutualiser tout le code concernant le même **crosscutting concern ou aspect**



Cross-cutting concerns





Définitions

- ❖ L'**Aspect Oriented Programming** (AOP) fournit une solution très flexible pour implémenter des services techniques transversaux au code applicatif comme par exemple la gestion des transactions, le tracing, le profiling la sécurité.



Terminologie

- ❖ Un **aspect** (aspect) est une factorisation du code s'appliquant à plusieurs classes (Exemple : gestion des transactions).
- ❖ Un **point de jonction** (join point) : C'est un endroit spécifique dans le flot d'exécution, où il est valide d'insérer un greffon.
Par exemple, il n'est pas possible, d'insérer un greffon au milieu du code d'une fonction. Par contre on pourra le faire avant, autour de, à la place ou après l'appel de la fonction
- ❖ Un **greffon** (advice) : Un traitement qui sera exécuté à certains point de jonctions.
- ❖ Un **intercepteur** (interceptor) : C'est un type de greffon particulier qui s'exécute avant et après une méthode. Les intercepteurs sont organisés en chaîne d'interception.
- ❖ Un **point de rencontre** (pointcut) : C'est l'endroit du logiciel où est inséré un greffon par le tisseur d'aspect.
Un point de rencontre est exprimé en général par le langage d'expression. (Exemple : des critères sur le nom de la méthode) .



Terminologie

- ❖ **Introduction** (introduction) : Également appelée « inter-type declaration », un moyen pour ajouter des méthodes additionnelles à un objet. Spring permet d'introduire des interfaces
- ❖ **Objet cible** (Target object) : L'objet sur lequel on a placé des greffons. L'implémentation est basée sur le pattern proxy, ainsi tout objet cible est associé à un proxy.
- ❖ **AOP proxy** : L'objet créé par le framework AOP et qui implémente les méthodes des greffons.
- ❖ Le **Tissage** (*Weaving*) : C'est le traitement qui lie les aspects aux objets. Ce traitement peut être exécuté à la compilation ou dynamiquement à l'exécution. Spring exécute le tissage à l'exécution.



Cas d'utilisation

- ❖ L'application la plus classique de l'AOP concerne la **gestion déclarative des transactions**. Les greffons sont de type *around* et interceptent les appels aux méthodes pour :
 - A l'aller : Créer une transaction ou se rattacher à une transaction courante
 - Au retour : Commiter ou rollbacker la transaction
- ❖ Un autre cas d'utilisation peut être l'application de **contrôles de sécurité** spécifiques .
- ❖ On peut également insérer des greffons pour le **debugging** ou le **profiling** d'applications durant les phases de développement
- ❖ Des greffons qui mutualisent les **traitements des exceptions**, des intercepteurs qui envoient des emails lors d'erreurs fatales par exemple,



Spring AOP

Concepts de l'AOP

Caractéristiques de Spring AOP

AOP via XML

AOP via annotations *@AspectJ*



Introduction

- ❖ Spring AOP est implémenté intégralement en Java et aucune phase de compilation préalable n'est nécessaire.
- ❖ Il ne s'appuie pas sur un class loader hiérarchique et peut donc être utilisé dans un serveur Tomcat ou JavaEE
- ❖ L'objectif principal de Spring AOP est
 - de fournir des services techniques transverses à de simples POJOs. Exemple, service de transaction
 - De permettre l'implémentation d'aspects personnalisés
- ❖ 2 possibilités pour utiliser Spring AOP
 - Déclarer les aspects via XML
 - Utiliser les annotations de `@AspectJ`



Fonctionnalités AOP

❖ Spring AOP supporte

- **L'interception de méthode** : Des comportements spécifiques peuvent alors être ajoutés avant et après les invocations de méthodes applicatives.
- **L'introduction** : La spécification d'un greffon a comme conséquence l'implémentation d'une interface additionnelle.
- **Points de rencontre statique et dynamique** : Les points de rencontre expriment des conditions d'exécution, ces expressions peuvent être réutilisées par plusieurs greffons.
 - Les points de rencontre statiques concernent les signatures de méthodes.
 - Les points de rencontre dynamiques peuvent prendre en compte les arguments des méthodes.



Types de greffons

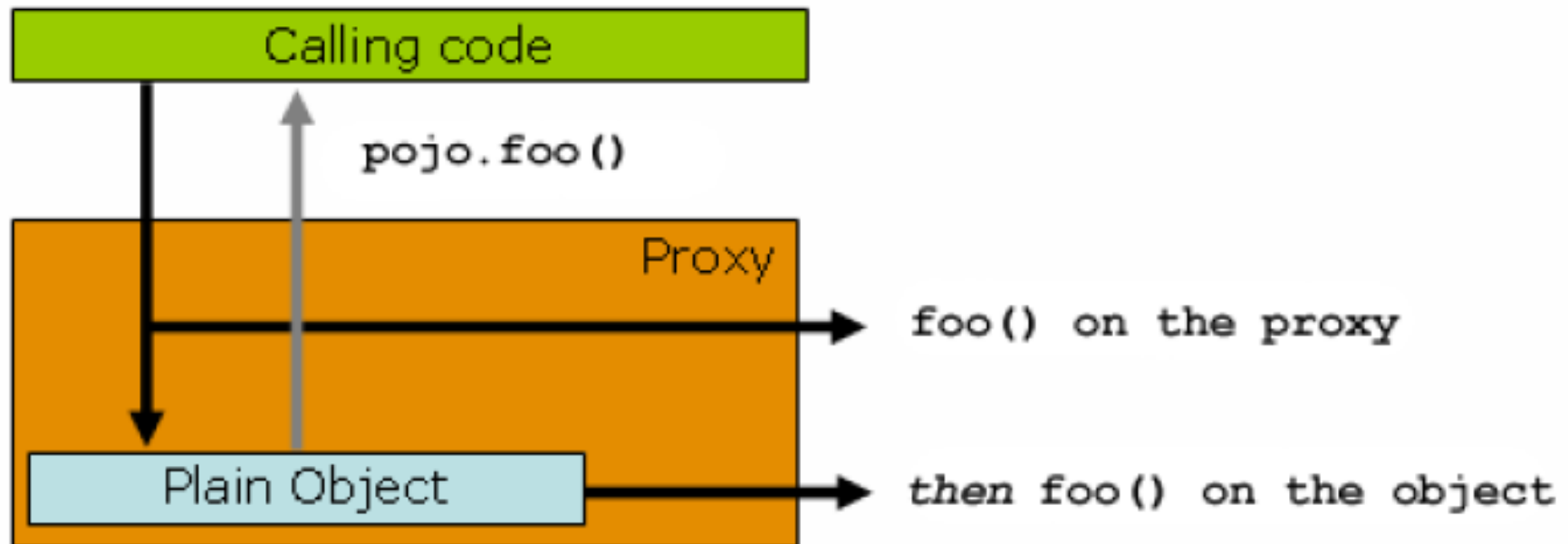
- ❖ Spring injecte les greffons sur les instances d'objets et non pas au niveau de la classe. Il est ainsi possible d'avoir différentes instances de la même classe sur lesquelles ne sont pas appliquées les mêmes greffons.
- ❖ 5 types de greffons sont disponibles :
 - **Before advice** : Exécution avant le point de rencontre. La méthode du point de rencontre est toujours exécutée sauf si le greffon lance une exception
 - **After returning advice** : Exécution après que la méthode du point de rencontre s'exécute normalement.
 - **After throwing advice** : Exécution après que la méthode du point de rencontre si celle-ci lance une exception
 - **After (finally) advice** : Exécution après que la méthode du point de rencontre de toute façon.
 - **Around advice** : Exécution avant et après le point de rencontre; c'est le greffon le plus générique. Il peut également décider de ne pas exécuter le point de rencontre si le greffon lance une exception.



Implémentation

- ❖ L'implémentation de l'AOP par Spring est basée sur les Proxy :
 - Soit en utilisant les **proxys dynamiques du JDK** (*java.lang.reflect.Proxy*). Cette méthode est utilisée lorsqu'une interface existe.
 - Soit en générant à l'exécution des classes proxys en utilisant la librairie **CGLIB**. Si les aspects sont appliqués sur des classes concrètes

Mécanisme de proxy





Mécanisme de proxy

- ❖ Attention aux appels directs de méthodes. Dans l'exemple suivant les greffons affectés à la méthodes *bar()* ne sont pas exécutés lorsque l'on appelle la méthode *foo()*

```
public class SimplePojo implements Pojo {  
    public void foo() {  
        this.bar();  
    }  
    public void bar() {  
        // some logic...  
    }  
}
```



Spring AOP

Concepts de l'AOP
Caractéristiques de Spring AOP
AOP via XML
AOP via annotations *@AspectJ*



Espace de nom

- ❖ Afin d'utiliser l'espace de nom *aop*, il faut au préalable importer le schéma ***spring-aop*** :

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



Éléments *<aop:*

- ❖ Ensuite, toutes les configurations relatives à AOP sont incluses dans un ou plusieurs éléments ***<aop:config>*** qui contient les sous-éléments suivants (dans l'ordre) :
 - ***<aop:pointcut/>*** : Permet d'exprimer des points de rencontre pouvant être utilisés par différents aspects.
 - ***<aop:advisor/>*** : Spécifique à Spring, c'est une simplification de langage pour exprimer un aspect contenant un unique greffon.
 - ***<aop:aspect/>*** : Permet de spécifier des aspects, il contient également des sous éléments *<aop:pointcut/>*.



Déclaration d'aspect

- ❖ Un aspect est un simple bean Spring auquel est ajouté la définition des points de rencontre et des greffons.
- ❖ Un aspect est donc déclaré grâce à l'élément `<aop:aspect>` qui indique la référence vers un bean Spring

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>
<bean id="aBean" class="...">
  ...
</bean>
```



Point de rencontre

❖ La syntaxe *AspectJ* est utilisée :

```
<aop:config>
```

```
<aop:pointcut id="businessService"
```

```
expression="execution(*com.xyz.myapp.service.*(..))"/>
```

```
</aop:config>
```



Greffons

❖ Les types de greffons disponibles :

- *<aop:before/>*
- *<aop:after-returning/>*
- *<aop:after-throwing/>*
- *<aop:after/>*
- *<aop:around/>*



Examples

```
<aop:aspect id="afterThrowingExample" ref="aBean">
  <aop:before pointcut-ref="dataAccessOperation"
    method="doAccessCheck"/>
  <aop:after-returning pointcut-ref="dataAccessOperation"
    returning="retVal" method="doAccessCheck"/>
  <aop:after-throwing pointcut-ref="dataAccessOperation"
    throwing="dataAccessEx" method="doRecoveryActions"/>
  <aop:after pointcut-ref="dataAccessOperation"
    method="doReleaseLock"/>
  <aop:around pointcut-ref="businessService"
    method="doBasicProfiling"/>

</aop:aspect>
```



Exemple - Performance

- ❖ Monitoring de performance sur les méthodes métier :

```
<bean id="performanceMonitor"
      class="com.example.PerformanceMonitor"/>
<aop:config>
  <aop:aspect ref="performanceMonitor">
    <aop:around pointcut="execution(public *
                          com.example.Service+.*(..))" method="monitor" />
  </aop:aspect>
</aop:config>
```

- ❖ La méthode *monitor()* de *com.example.PerformanceMonitor* est appelée dès qu'une méthode de la classe *com.example.Service* ou de ses sous-classes est invoquée.



Exemple - Performance

- ❖ La méthode *monitor()* démarre un timer, autorise l'exécution normale de la méthode cible (via *proceed()* de *org.aspectj.lang.ProceedingJoinPoint*), puis arrête le timer au retour de la méthode cible, enregistre la mesure puis retourne la valeur de la méthode cible:

```
public Object monitor(ProceedingJoinPoint pjp) throws Throwable {  
    long start = System.nanoTime();  
    try {  
        return pjp.proceed();  
    } finally {  
        long time = System.nanoTime() - start;  
        // do something with time...  
    }  
}
```




Spring AOP

Concepts de l'AOP
Caractéristiques de Spring AOP
AOP via XML

AOP via annotations @AspectJ



Activation de *@AspectJ*

- ❖ S'assurer de la présence de ***aspectjweaver.jar***
org.aspectj :aspectjweaver
- ❖ Puis autoriser le support d'*@AspectJ* :
 - Soit par la configuration XML

<aop:aspectj-autoproxy/>

- Soit par la configuration Java

@Configuration

@EnableAspectJAutoProxy

public class AppConfig



Utilisation de *@AspectJ*

- ❖ Une fois *@AspectJ* activé, il faut :
 - Définir un aspect, via *@Aspect*
 - Éventuellement, déclarer les points de rencontre
 - Déclarer les greffons par les annotations



Déclaration d'un aspect

- ❖ Classe Java, *org.xyz.NotVeryUsefulAspect*

```
package org.xyz;  
import org.aspectj.lang.annotation.Aspect;  
@Aspect  
public class NotVeryUsefulAspect {  
}
```
- ❖ Les aspects contiennent des méthodes et des attributs comme toute classe Java et de plus, ils déclarent des points de rencontre, des greffons et des introductions



Déclaration d'un point de rencontre

- ❖ La déclaration d'un point de rencontre est composée de deux parties :
 - Sa signature : nom de méthode et paramètres. Le type de retour est toujours *void*
 - Une expression : Permet de déterminer les méthodes sur lesquelles seront appliquées l'aspect

```
@Pointcut("execution(* transfer(..))")// the pointcut expression  
private void anyOldTransfer() {}// the pointcut signature
```



Expression des points de rencontre

- Spring AOP ne supporte qu'un sous-ensemble de la syntaxe *AspectJ*. Le caractère * est supporté :

execution : Certaines signatures de méthodes

within : Certains types

this : Un type donné du proxy

target : Un type donné de l'objet cible

args : Un type donné des arguments de la méthode

@target : Une annotation donnée de l'objet cible

@args : Une annotation donnée des arguments de la méthode

@within : Une annotation donnée d'un type de l'objet cible

@annotation : Une annotation donnée des méthodes

bean : Un nom de bean Spring (Spécifique Spring)



Exemples

execution(public * *(..)) : Toutes les méthodes publiques

execution(* com.xyz.service.*.*(..)) : Toutes les méthodes définies dans le package *service*

within(com.xyz.service..*) : Tous les points de jonction (méthodes pour Spring) définis dans le package *service* et ses sous-packages

this(com.xyz.service.AccountService) : Tous les points de jonction (méthodes pour Spring) ou le proxy implémente l'interface *AccountService*

target(com.xyz.service.AccountService) : Tous les points de jonction (méthodes pour Spring) ou l'objet cible implémente l'interface *AccountService*

args(java.io.Serializable) : Toutes les méthodes prenant un seul argument *Serializable*



Exemple *SystemArchitecture*

- ❖ Une approche recommandée consiste à déclarer un aspect "*SystemArchitecture*" qui capture les expressions communes des points de rencontre :

```
@Aspect
public class SystemArchitecture {
    @Pointcut("within(com.xyz.someapp.web..*)")
    public void inWebLayer() {}

    @Pointcut("within(com.xyz.someapp.service..*)")
    public void inServiceLayer() {}

    @Pointcut("within(com.xyz.someapp.dao..*)")
    public void inDataAccessLayer() {}

    @Pointcut("execution(* com.xyz.someapp.service.*(..))")
    public void businessService() {}

    @Pointcut("execution(* com.xyz.someapp.dao.*(..))")
    public void dataAccessOperation() {}
}
```




Déclaration des greffons

- ❖ Un greffon est associé à une expression de point de rencontre et s'exécute avant et/ou après les méthodes correspondant à l'expression
- ❖ Un greffon peut donc être associé :
 - À une référence vers l'expression du point de rencontre

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")  
public void doAccessCheck() { ... }
```

- A l'expression directement.

```
@Before("execution(* com.xyz.myapp.dao.*(..))")  
public void doAccessCheck() { ...}
```



Annotations des greffons

❖ Les annotations disponibles sont donc :

- ***@Before***
- ***@AfterReturning***
- ***@AfterThrowing***
- ***@After (finally)***
- ***@Around***



Particularités des greffons *after*

- ❖ *@AfterReturning* peut utiliser la valeur retournée par la méthode :

```
@AfterReturning(  
pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
returning="retVal")  
public void doAccessCheck(Object retVal) {  
    // ...  
}
```

- ❖ Il est possible de restreindre l'exécution de *@AfterThrowing* à un certain type d'exception :

```
@AfterThrowing(  
pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",  
throwing="ex")  
public void doRecoveryActions(DataAccessException ex) {  
    // ...  
}
```



Annotation *@Around*

- ❖ Le premier argument d'une méthode *Around* doit être de type *ProceedingJoinPoint*
- ❖ L'appel à la méthode *proceed()* sur cet argument démarre l'exécution de la méthode de l'objet cible
- ❖ On peut passer en argument un tableau d'objet qui serviront d'arguments à la méthode
- ❖ La valeur retournée par la méthode *@Around* sera la valeur reçue par l'appelant



SpringBoot

L'auto-configuration

Starters SpringBoot

Structure projet et principales
annotations

Configuration SpringBoot



Introduction

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

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

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



Essence

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

Les bibliothèques sont groupées en des *starter modules*

Toutes les librairies sont ensuite packagées avec le code applicatif dans un **fat-jar** : l'exécutable



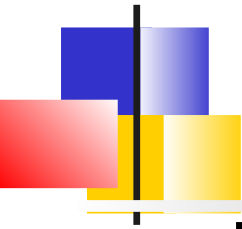
Auto-configuration

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

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

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

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

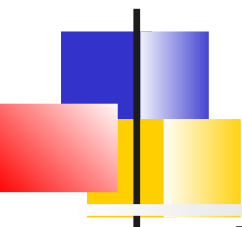


Java

Gestion des dépendances

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

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



Auto-configuration (Java)

En fonction des librairies présentes dans l'exécutable Java, 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



Personnalisation de la configuration

La configuration par défaut peut être surchargée par différents moyens

- Des **variables d'environnement**
- Des **arguments de la ligne de commande**
- Des **fichiers de configuration externe** (*.properties* ou *.yml*) présent dans l'archive ou accessible via http.
On peut fournir différents fichiers qui seront activés en fonction de profils
- Du code en utilisant des **classes spécifiques du framework** (exemple *AuthenticationManagerBuilder* pour gérer l'authentification)

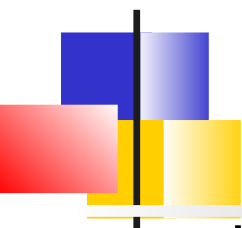


Auto-configuration

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;
    }
}
```



SpringBoot

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



Starters

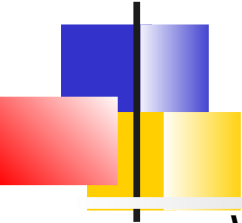
Les développeurs utilisent des starters-modules qui fournissent :

- Une ensemble de librairies dédiés à un type d'application.
- Des beans techniques qui fournissent des services d'intégration configurables

Spring fournit des starter-module pour tout type de problématique

Un assistant liste tous les starters-disponibles :

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



Starters les + importants

Web

- *-**web** : Application web ou API REST
- *-**reactive-web** : Application web ou API REST en mode réactif
- *-**web-services** : Services Web SOAP

Cœurs :

- *-**logging** : Utilisation de logback (Starter par défaut)
- *-**test** : Test avec Junit, Hamcrest et Mockito
- *-**devtools** : Fonctionnalités pour le développement
- *-**lombok** : Simplification du code Java
- *-**configuration-processor** : Complétion des propriétés de configuration 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
- *-**data-rest** : Utilisation de Spring Data Rest



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

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**

Configuration SpringBoot



Structure projet

Aucune obligation mais des recommandations :

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



Structure typique

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



@Configuration

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

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



Exemple

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



Auto-configuration

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

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

Ex :

```
@EnableAutoConfiguration(exclude={DataSourceA  
utoConfiguration.class})
```



Beans et Injection de dépendance

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

La technique la plus simple est généralement la combinaison de :

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



Exemple

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

    // ...

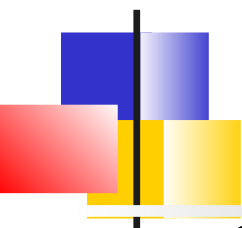
}
```




SpringBoot

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

Configuration SpringBoot



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 :

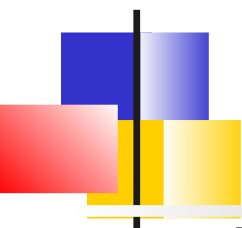
- Directement via l'annotation **@Value**
- Ou associer à un objet structuré via l'annotation **@ConfigurationProperties**



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 placé 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(regexp = "\\d{3}-\\d{3}-\\d{4}")
```

```
    private String adminContactNumber;
```

```
    @Min(1)
```

```
    private int refreshRate;
```

```
        .....

```

```
}
```




Persistence

Principes de SpringData

SpringData JPA

SpringData MongoDB



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);
```

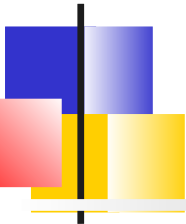


Limite

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
SpringData MongoDB



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`



Persistence

Principes de SpringData
SpringData et JPA
SpringData MongoDB



Introduction

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

Exemple MongoDB

`spring-boot-starter-data-mongodb`



Connexion à une base MongoDB

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

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

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



Entité

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

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

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

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



Mongo Repository

SpringData propose également des implémentations de Repository pour les base NoSQL

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

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



Usage

```
@Controller
public class MyController {

    @Autowired
    private CustomerRepository repository;

    @Override
    public void doIt(throws Exception {

        repository.deleteAll();

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



MongoTemplate

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

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



Mongo Embarqué

Il est possible d'utiliser un Mongo embarqué

Il suffit d'avoir des dépendances vers :

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

Le port utilisé est soit déterminé

aléatoirement soit fixé par la propriété :

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

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



Applications Web

Présentation Spring MVC

Modèle MVC classique

APIs Rest

Personnalisation configuration



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***



Beans spécifiques à Spring MVC

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

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

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

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

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

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

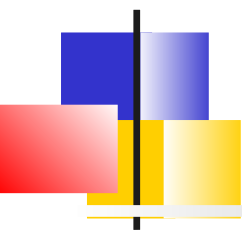
FlashMapManager : Permet de transférer des attributs d'une requête à une autre habituellement via une redirection d'URL



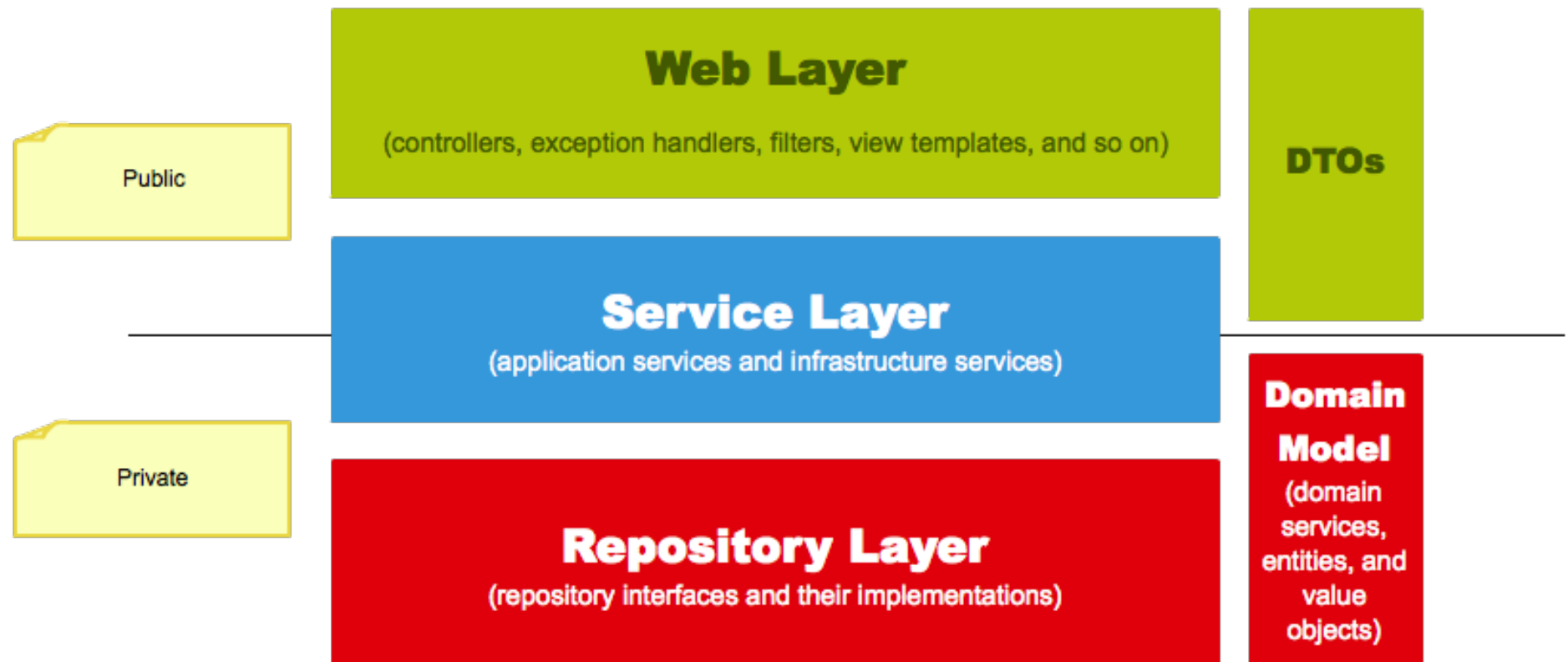
Rappels Spring MVC

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

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



Architecture classique projet





@Controller, @RestController

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

@Controller

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



@RequestMapping

@RequestMapping

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



Compléments *@RequestMapping*

Des variantes pour limiter à une méthode :

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

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

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

Utiliser des expressions régulières

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

Utiliser des propriétés de configuration

@RequestMapping("\${context}")



Types des arguments de méthode

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

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

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



Annotations sur les arguments

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

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



Gabarits d'URI

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

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

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

```
@GetMapping("/owners/{ownerId}")
```

```
public String findOwner(@PathVariable String  
    ownerId, Model model) {
```



Compléments

- Un argument **@PathVariable** peut être de type simple, Spring fait la conversion automatiquement
- Si *@PathVariable* est utilisée sur un argument *Map <String, String>*, l'argument est renseigné avec toutes les variables du gabarit
- Un gabarit peut être construit à partir de la combinaison des annotations de type et de méthode



Paramètres avec *@RequestParam*

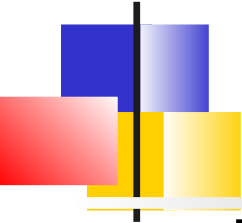
```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

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

    // ...

}
```



Types des valeurs de retours des méthodes

Les types des valeurs de retour possibles sont :

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



Formats d'entrée/sorties

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

En entrée, précise le format attendu

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

Ou en sortie, précise le format généré :

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



@RequestBody et convertisseur

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

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



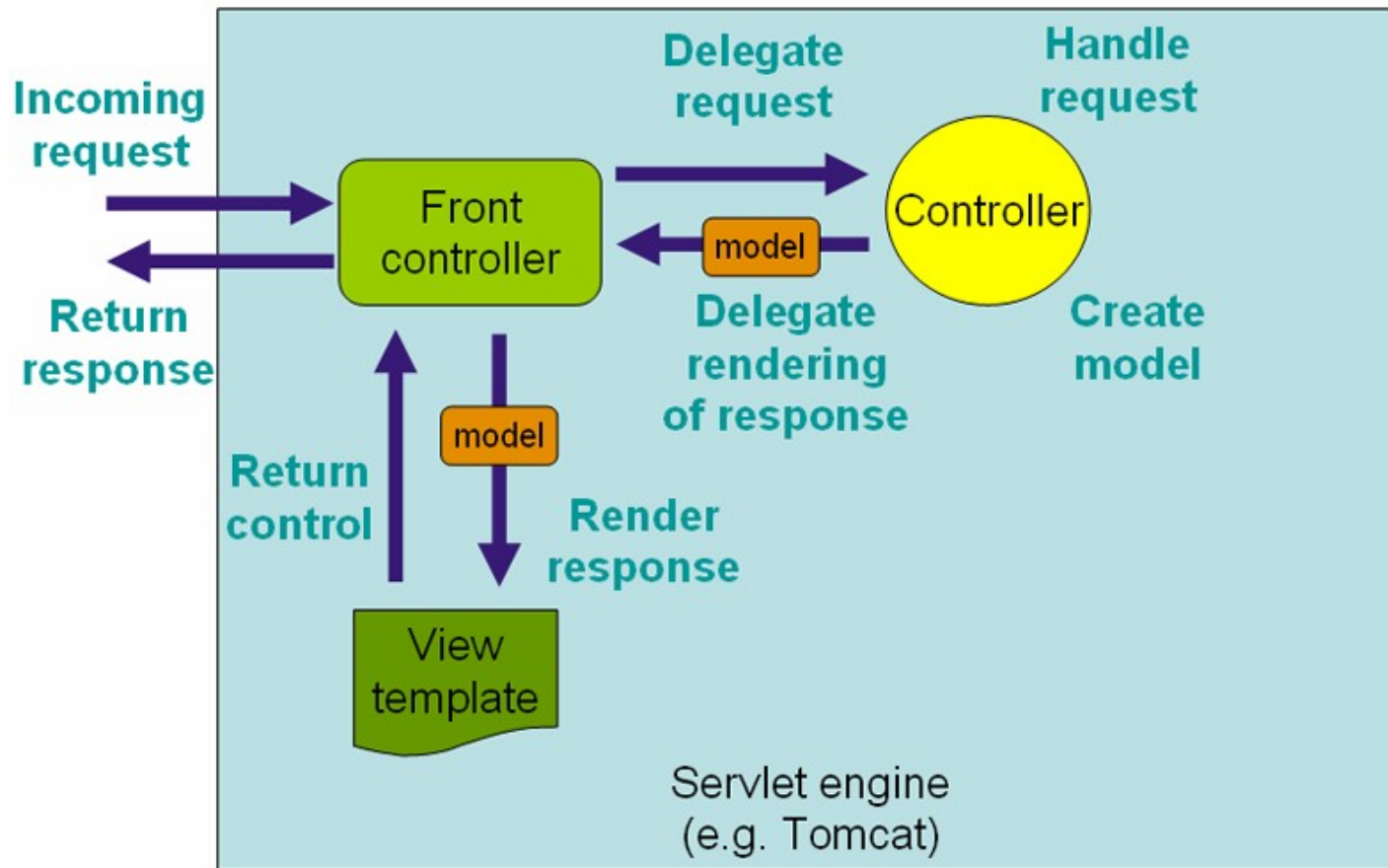
Applications Web

Présentation Spring MVC
Modèle MVC classique

APIs Rest

Personnalisation configuration

MVC





Exemple Web Controller

@Controller

@RequestMapping("/web")

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

@GetMapping(path = "/register")

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

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

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



Autres fonctionnalités

D'autres fonctionnalités sont apportées par le framework :

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



Validation de formulaire

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

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

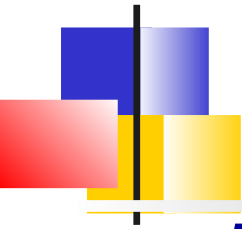
C'est alors la responsabilité du contrôleur de tester la présence d'erreur de validation

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



Exemple

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



Thymeleaf

Thymeleaf est la technologie de vue par défaut de SpringBoot

Elle permet de construire des pages HTML-5

Les vues sont constituées :

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



Attributs

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

th:object : Sélection d'une variable de contexte

th:errors : Accès aux messages d'erreur

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

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

th:if, th-each : Test, boucles

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

th:on* : Événements Javascript

....



Expressions Thymeleaf

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



Exemple vue ThymeLeaf

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




Auto-configuration

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

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



Applications Web

Présentation Spring MVC
Modèle MVC classique

APIs Rest

Personnalisation configuration



@RestController

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

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



Exemple pour des données JSON

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

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

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

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

    @PostMapping
    public ResponseEntity<Member> register(@Valid @RequestBody Member member) {

        member = memberRepository.save(member);

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



Sérialisation JSON

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

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

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



API Jackson

Jackson propose 3 principales API :

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

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



Alternative à la sérialisation

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

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

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



Annotations Jackson

@JsonProperty, @JsonGetter, @JsonSetter, @JsonAnyGetter, @JsonAnySetter, @JsonIgnore, @JsonIgnoreProperty, @JsonIgnoreType : Permettant de définir les propriétés JSON

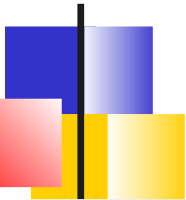
@JsonRootName : Arbre JSON

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

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

@JsonView : Permet de définir différentes sérialisations pour les mêmes objets

....



Sérialisation JSON spécifique

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

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

@JsonComponent

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



Auto-configuration

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

- Fourniture automatique de *RestTemplateBuilder* pour effectuer des appels REST



Exemple

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

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

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



SpringDoc

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

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

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

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



Fonctionnalités

Par défaut,

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

SpringDoc prend en compte

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

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



Applications Web

Présentation Spring MVC

Modèle MVC classique

APIs Rest

Personnalisation configuration



Personnalisation de la configuration

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



Exemple : Définition de *ViewController*

```
@Configuration
```

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

```
    @Override
```

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

```
}
```




Exemple Cross-origin

Le *crosss-origin resource sharing*, i.e pouvoir faire des requêtes vers des serveurs différents que son serveur d'origine peut facilement se configurer globalement en surchargeant la méthode *addCorsMapping* :

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

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



Exemple *Intercepteurs*

```
@SpringBootApplication
public class MyApplication implements WebMvcConfigurer {

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

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        System.out.println("Adding interceptors");
        registry.addInterceptor(new
MyInterceptor()).addPathPatterns("/**");
        super.addInterceptors(registry);
    }
}
```



Gestion des erreurs

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

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

Pour remplacer le comportement par défaut :

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



Exemple

@ResponseStatus(value = HttpStatus.NOT_FOUND)

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



ResponseStatusException

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

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



Exemple *@ControllerAdvice*

@ControllerAdvice

```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {

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



Spring Security

Principes

Modèles stateful/stateless
Autoconfiguration Spring Boot
oAuth2



Spring Security

Spring Security gère principalement 2 domaines de la sécurité :

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

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

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



Principe et mécanisme

La configuration par défaut de la sécurité web peut être provoquée par l'annotation **@EnableWebSecurity** ou par SpringBoot

La configuration crée alors le bean **springSecurityFilterChain** qui encapsule une chaîne de filtres responsable de tous les aspects de la sécurité. Le filtre est hautement configurable et s'adapte à toutes les approches

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



Quelques Filtres communs de *springSecurityFilterChain*

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

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

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

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



Personnalisation de la sécurité

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

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



Exemple : Configuration de la chaîne de filtres

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



Définition des filtres

L'ordre des filtres a une grosse importance.

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

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



Debug de la sécurité

Pour debugger la configuration :

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

Pour debugger l'exécution :

- Activer les traces de DEBUG :

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



Exemple : Configuration du gestionnaire d'authentification

```
@Configuration
```

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

```
@Override
```

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

```
    // Ici on utilise un realm mémoire, AuthenticationManagerBuilder permet  
    // également de facilement de se connecter à un annuaire LDAP ou une bd
```

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

```
    }  
}
```



Personnalisation via *UserDetailsService*

Une alternative à la personnalisation de l'authentification est de fournir un bean implémentant ***UserDetailsService***

L'interface contient une seule méthode :

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

- Elle est responsable de retourner, à partir d'un login, un objet de type *UserDetails* encapsulant le mot de passe et les rôles
C'est le framework qui vérifie si le mot de passe saisi correspond.
- La présence d'un bean de type *UserDetailsService* suffit à sa configuration



Exemple

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

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



Password Encoder

Spring Security 5 nécessite que les mots de passes soient encodés

Il faut alors définir un bean de type
PasswordEncoder

L'implémentation recommandée est
BcryptPasswordEncoder

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



{noop}

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

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



Spring Security

Principes

Modèles *stateful*/*stateless*

Autoconfiguration Spring Boot

oAuth2



Application Web et API Rest

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

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



Processus d'authentification appli web back-end

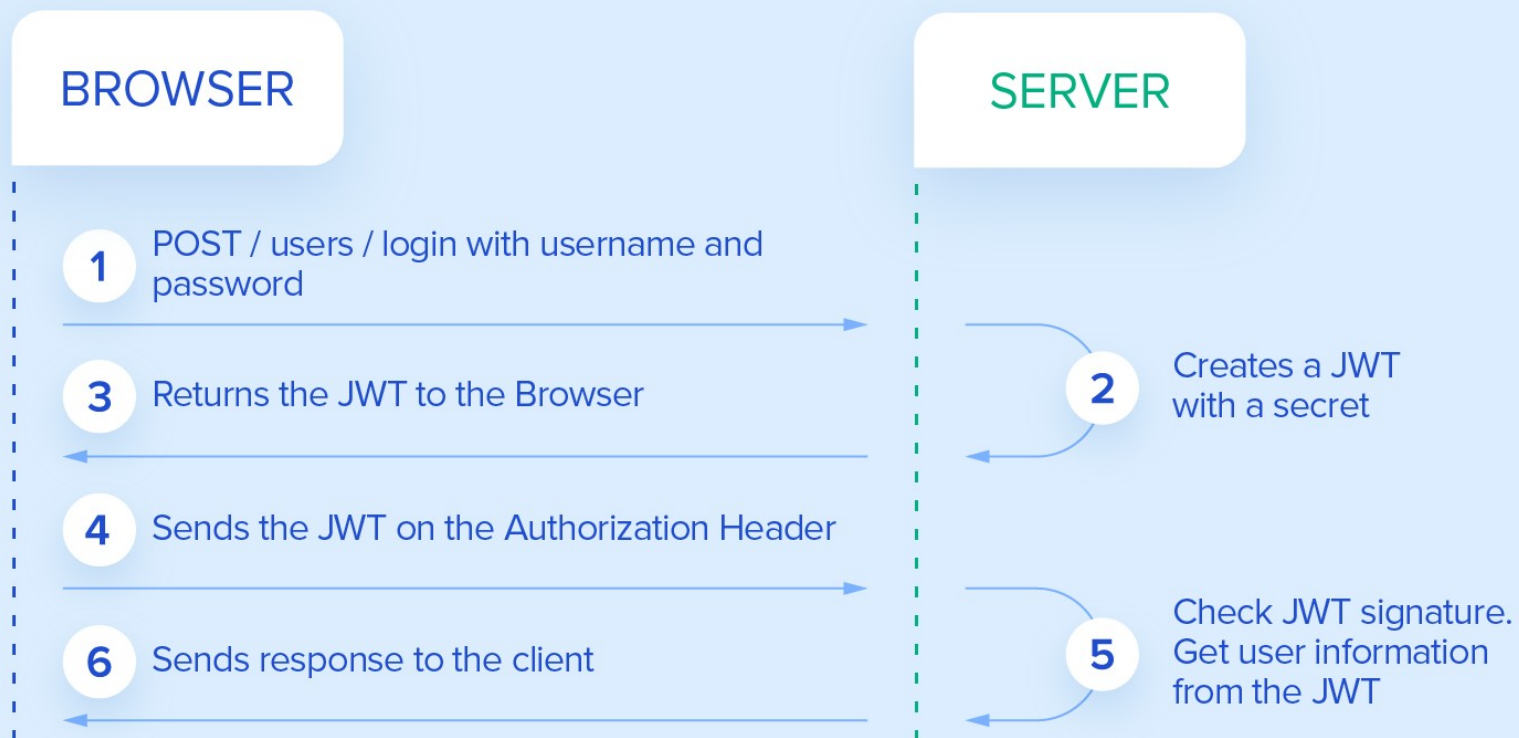
1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier :
 1. En redirigeant vers une page de login
 2. En fournissant les entêtes pour une authentification basique du navigateur .
3. Le navigateur renvoie une réponse au serveur :
 1. Soit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



Processus d'authentification appli REST

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

Authentication Rest





Spring Security

Principes

Modèles stateful/stateless

Autoconfiguration Spring Boot

oAuth2



Apports de SpringBoot

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

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



Gestionnaire d'authentification par défaut

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

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

security.user.name= myUser

security.user.password=secret



Autres fonctionnalités par défaut

D'autres fonctionnalités sont automatiquement obtenues :

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



SSL

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

Par exemple :

```
server.port=8443
```

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

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

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

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

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



Spring Security

Principes

Modèles stateful/stateless

Autoconfiguration Spring Boot

OAuth2



Introduction

oAuth2 est un protocole entre différents acteurs qui permet de spécifier des contraintes d'autorisation sur des ressources protégées

Il est également utilisé via OpenID Connect pour l'authentification

Spring Security offre des supports pour :

- S'authentifier sur un serveur d'autorisation externe OpenID (Github, Google, Facebook, ...)
- Protéger des ressources URLs via oAuth2
- S'intégrer à la solution Okta



Rôles du protocole

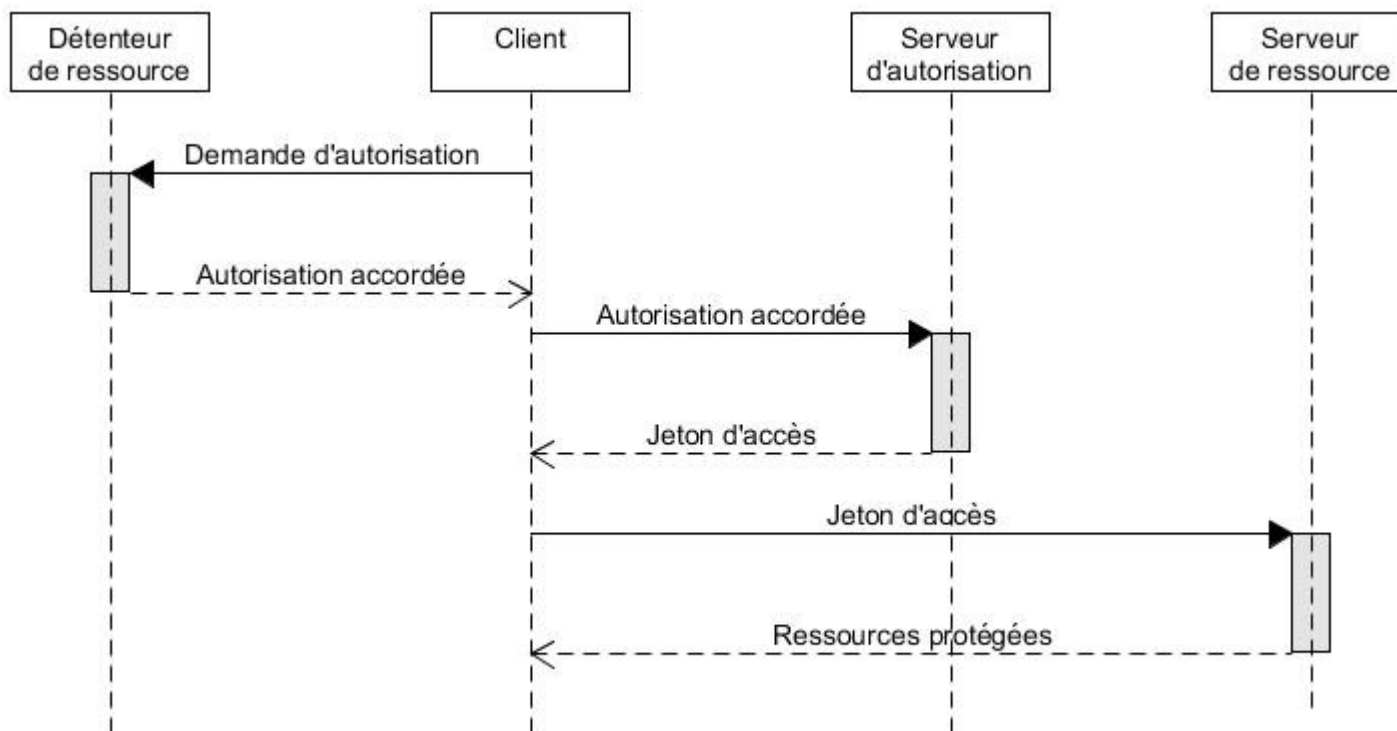
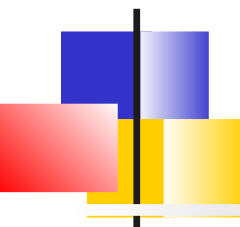
Le **Client** est l'application qui essaie d'accéder au compte utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

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

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

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

Rq: Un participant du protocole peut jouer plusieurs rôles





Scénario

1. Pré-enregistrer le client auprès du service d'autorisation (=> client ID et un secret)
2. Obtenir l'autorisation de l'utilisateur.
(4 types de grant)
3. Obtention du token (date d'expiration)
4. Appel de l'API pour obtenir les informations voulues en utilisant le token
5. Validation du token par le serveur de ressource



Tokens

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

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

Il y a 2 types de token

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



Périmètre d'accès

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

Le serveur d'autorisation définit les *scopes* disponibles

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



Enregistrement du client

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

Le client doit fournir :

- **Application Name**: Le nom de l'application
- **Redirect URLs**: Les URLs du client pour recevoir le code d'autorisation et le jeton d'accès
- **Grant Types** : Les types d'autorisations utilisables par le client
- **Javascript Origin** (optionnel): Le host autorisé à accéder aux ressources via *XMLHttpRequest*

Le serveur répond avec :

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



OAuth2 Grant Type

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

- **authorization code** :
 - L'utilisateur est dirigé vers le serveur d'autorisation
 - L'utilisateur consent sur le serveur d'autorisation
 - Il est redirigé vers le client avec un code d'autorisation
 - Le client utilise le code pour obtenir le jeton
- **implicit** : Jeton fourni directement. Certains serveurs interdisent de mode
- **password** : Le client fournit les créidentiels de l'utilisateur
- **client credentials** : Le client est l'utilisateur
- **device code** :



Usage du jeton

Le jeton est passé à travers 2 moyens :

- Les paramètres HTTP. (Les jetons apparaissent dans les traces du serveur)
- ***L'entête d'Authorization***

```
GET /profile HTTP/1.1
```

```
Host: api.example.com
```

```
Authorization: Bearer MzJmNDc3M2VjMmQzN
```

<http://www.bubblecode.net/en/2016/01/22/understanding-oauth2/>



Validation du jeton

Lors de la réception du jeton, le serveur de ressource doit valider l'authenticité du jeton et extraire ses informations différentes techniques sont possibles

- Appel REST vers le serveur d'autorisation
- Utilisation d'un support persistant partagé (ex. JdbcStore)
- Utilisation de JWT et validation via clé privé ou clé publique



JWT

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

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

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

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

Différentes implémentations existent en Java (*io.jsonwebtoken*, ...) ou le starter ***spring-security-oauth2-jose***



Apport de SpringBoot

Le support de OAuth via Spring a été revu :

- Le projet **spring-security-oauth2** a été déprécié et remplacé par SpringSecurity 5.

Voir :

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

- Il n'y a plus de support pour un serveur d'autorisation

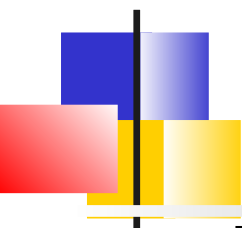
3 starters sont désormais fournis :

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



Solutions pour un serveur d'autorisation

- Utiliser un produit autonome
- Un projet Spring pour un serveur d'autorisation est en cours :
<https://github.com/spring-projects-experimental/spring-authorization-server>
- Une autre alternative est d'embarquer une solution *oAuth* comme *KeyCloak* dans un application SpringBoot
Voir par exemple :
<https://www.baeldung.com/keycloak-embedded-in-spring-boot-app>



Serveur de ressources

Dépendance :

```
<dependency>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>  
</dependency>
```

Le serveur de ressources doit vérifier la signature du jeton pour s'assurer que les données n'ont pas été modifiées.

- ***jwk-set-uri*** contient la clé publique que le serveur peut utiliser pour la vérification
- ***issuer-uri*** pointe vers l'URI du serveur d'autorisation de base, qui peut également être utilisé pour localiser le endpoint fournissant la clé publique



Exemple *application.yml*

```
server:
  port: 8081
  servlet:
    context-path: /resource-server

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://keycloak:8083/auth/realms/myRealm
          jwk-set-uri: http://keycloak:8083/auth/realms/myRealm/protocol/openid-connect/certs
```



Configuration typique *SpringBoot*

@Configuration

```
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.cors()
            .and()
            .authorizeRequests()
                .antMatchers(HttpMethod.GET, "/user/info", "/api/foos/**")
                    .hasAuthority("SCOPE_read")
                .antMatchers(HttpMethod.POST, "/api/foos")
                    .hasAuthority("SCOPE_write")
                .anyRequest()
                    .authenticated()
            .and()
                .oauth2ResourceServer()
                    .jwt();
    }
}
```



Personnalisations

Différents aspects de la configuration par défaut peuvent être personnalisées :

- Revendications personnalisées dans le jeton
- Charger la clé à partir d'un KeyStore



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

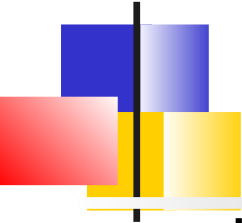
```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @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;

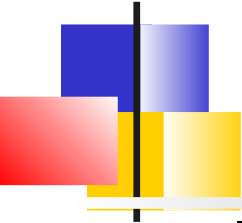
    @Autowired
    private Reverser reverser;

    @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



Messaging

Support pour les messages brokers



Introduction

Les communications asynchrones entre processus apportent plusieurs avantages :

- Découplage du producteur et consommateur de message
- Scaling et montée en charge
- Implémentation de patterns de micro-services
Saga¹, Event-sourcing Pattern²

Des difficultés :

- Gestion de l'asynchronisme
- Mise en place et exploitation d'un message broker

1. <https://microservices.io/patterns/data/saga.html>

2. <http://microservices.io/patterns/data/event-sourcing.html>

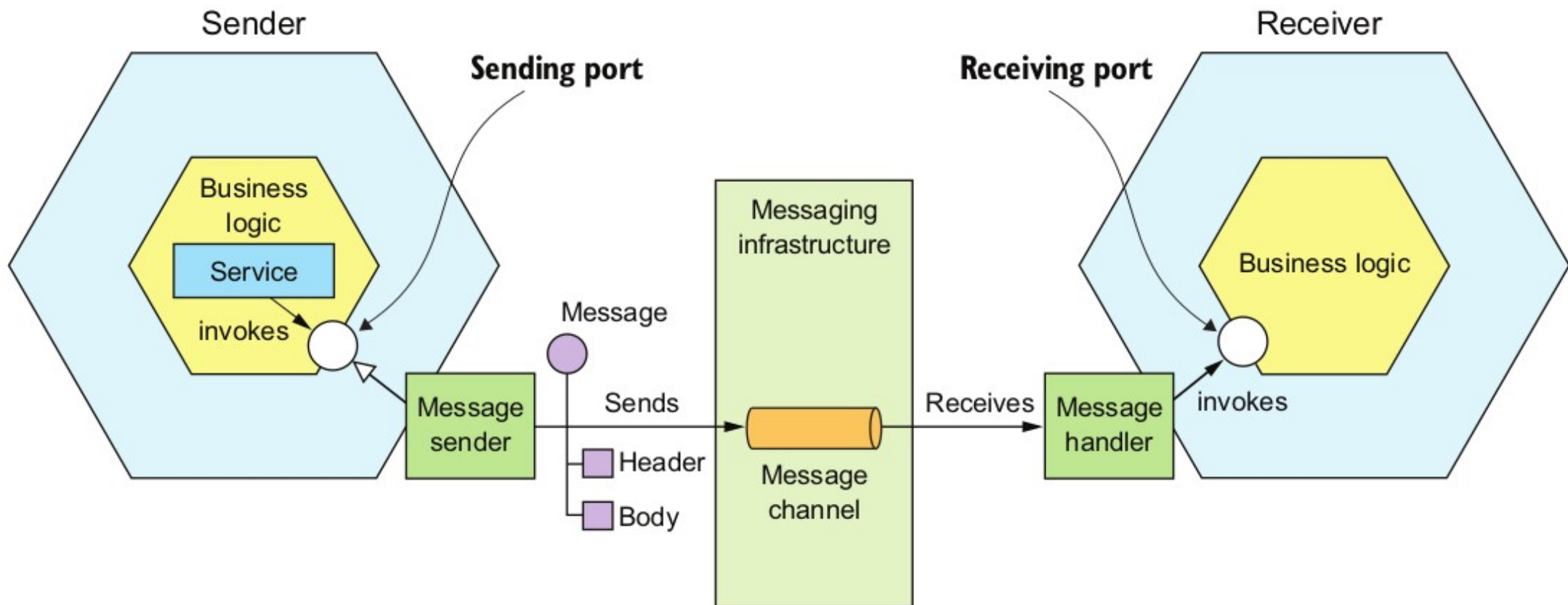


Messaging Pattern

Messaging Pattern¹ : Un client invoque un service en utilisant une messagerie asynchrone

- Le pattern messaging fait souvent intervenir un message broker
- Un client effectue une requête en postant un message asynchrone
- Optionnellement, il s'attend à recevoir une réponse

Architecture





Message

Un message est constitué d'entêtes (ensemble de clés-valeurs) et d'un corps de message

On distingue 3 types de messages :

- **Document** : Un message générique ne contenant que des données Le récepteur décide comment l'interpréter
- **Commande** : Un message spécifiant l'action à invoquer et ses paramètres d'entrée
- **Événement** : Un message indiquant que quelque chose vient de se passer. Souvent un événement métier



Canaux de messages

2 types de canaux :

- **Point-to-point** : Le canal délivre le message à un des consommateurs lisant le canal.
Ex : Envoie d'un message commande
- **PubAndSub** : Le canal délivre le message à tous les consommateurs attachés (les abonnés)



Styles d'interaction

Tous les styles d'interactions sont supportés :

- Requête/Réponse synchrone.
Le client attend la réponse
- Requête/Réponse asynchrone
Le client est notifié lorsque la réponse arrive
- One way notification
Le client n'attend pas de réponse
- Publish and Subscribe :
Le producteur n'attend pas de réponse
- Publish et réponse asynchrones
Le producteur est notifié lorsque les réponses arrivent



Spécification de l'API

La spécification consiste à définir

- Les noms des canaux
- Les types de messages et leur format.
(Typiquement JSON)

Par contre à la différence de REST et
OpenAPI pas de standard



Message Broker

Un message broker est un intermédiaire par lequel tous les messages transitent

- L'émetteur n'a pas besoin de connaître l'emplacement réseau du récepteur
- Le message broker bufferise les messages

Implémentations courantes :

- ActiveMQ
- RabbitMQ
- Kafka
- AWS Kinesis



Facteurs de choix (1°)

Langages de programmation supportés

C'est mieux si il en supporte plusieurs

Standard de messaging supportés

AMQP, STOMP ou propriétaire

Ordre des messages

Le message broker préserve t il l'ordre d'émission des messages

Garanties de livraison

At-most-Once, At-Least-Once ou Exactly-Once

Persistance

Les messages survivent-ils au crash ?



Facteurs de choix (2)

Durabilité

Si un consommateur se reconnecte au broker, récupère-t-il les messages qui ont été envoyés entre temps

Scalabilité

Le broker est-il scalable ?

Latence

Quel est le délai entre l'émission et la réception ?

Consommation concurrente

Le message broker permet-il que les messages d'un canal soient entre des récepteurs répliqués



Offre Spring

Starter messaging pur :

- RabbitMQ, ActiveMQ, Kafka, ActiveMQ Artemis, Solace PubSub

Pipeline de traitement d'évènements :

- Kafka Stream

Architecture micro-services *event-driven*

- Spring Cloud Stream
- Spring Data Flow



Exemple *spring-kafka*

Envoi de message

```
@Value("${app.my-channel}")
String PAYMENT_REQUEST_CHANNEL;

@Autowired
KafkaTemplate<Long, DomainEvent> kafkaOrderTemplate;

public Order doService(Domain model) {
    ...
    DomainEvent event = new DomainEvent(model);
    kafkaOrderTemplate.send(ORDER_STATUS_CHANNEL, event);
    ...
}
```

Réception de message :

```
@KafkaListener(topics = "#{ '${app.my-channel}' }", id = "oneHandler")
public void handleEvent(DomainEvent domainEvent) {
    ...
}
```



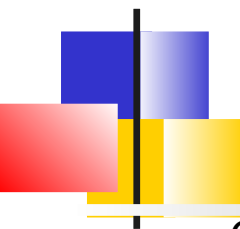
Reactive Spring

Programmation réactive

Spring Reactor

Spring Data Reactive

Spring Webflux

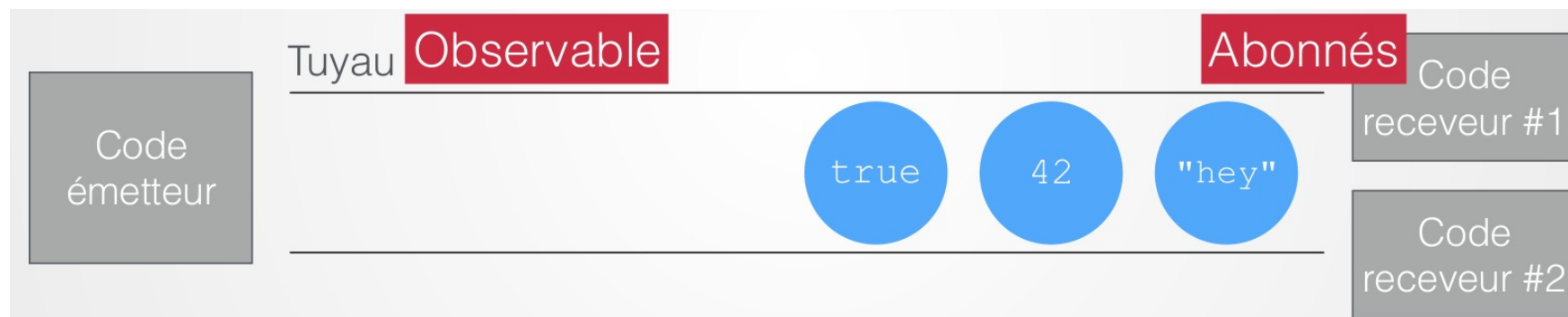


Modèle réactif

Consiste à construire son code à partir de flux de données : **stream**.

Certaines parties du code émettent des données : les **Observables**

D'autres réagissent : les **Subscribers** :



Le code émetteur peut envoyer un nombre illimité de valeurs dans le tuyau.

Le tuyau peut avoir un nombre illimité d'abonnés. Le tuyau est actif tant qu'il a au moins un abonné.

Les tuyaux sont des flux en temps réel : dès qu'une valeur est poussée dans un tuyau, les abonnés reçoivent la valeur et peuvent réagir.



Pattern et *ReactiveX*

La programmation réactive se base sur le pattern ***Observable*** qui est une combinaison des patterns *Observer* et *Iterator*

Elle utilise la programmation fonctionnelle permettant de définir facilement des opérateurs sur les éléments du flux

Elle est formalisé par l'API ***ReactiveX*** et de nombreuses implémentations existent pour différent langages (*RxJS*, *RxJava*, *Rx.NET*)



Combiner des transformations

Les données circulant dans le tuyau sont transformées grâce à une série d'opérations successives (aka "opérateurs") :

Les opérateurs à appliquer aux données sont déclarés une bonne fois pour toutes.

Ce fonctionnement déclaratif est pratique à utiliser et à déboguer.

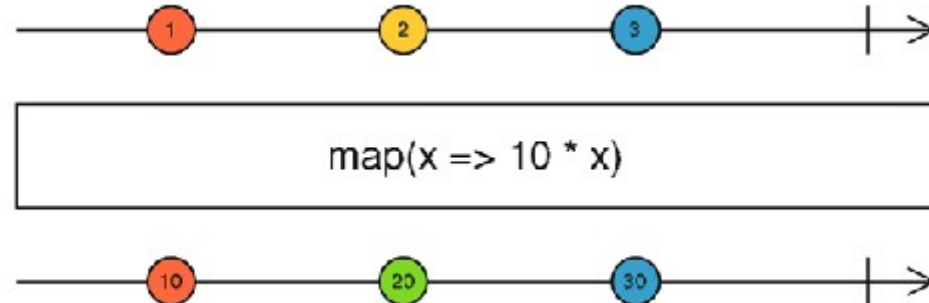




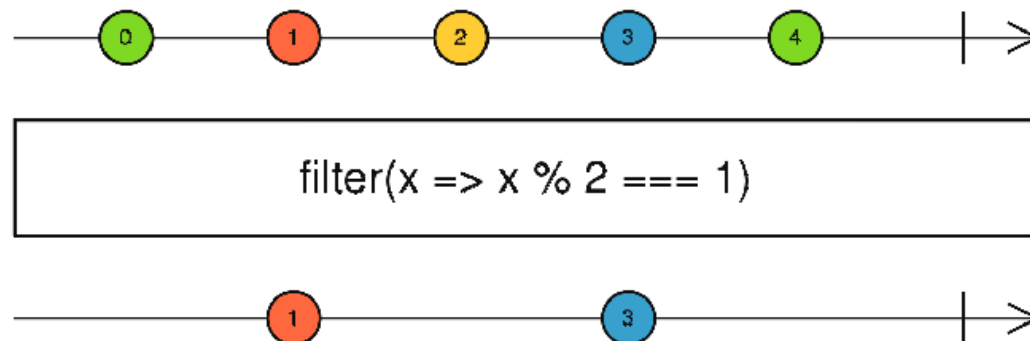
Marble diagrams

Pour expliquer les opérateurs, la doc utilise des ***marble diagrams***

Exemple *map* :



Exemple *filter*



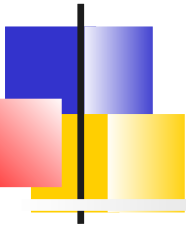


Reactive Streams

Reactive Streams a pour but de définir un standard pour le traitement asynchrone de flux d'événements offrant une fonctionnalité de ***non-blocking back pressure***

Il concerne les environnements Java et Javascript ainsi que les protocoles réseau

Le standard permet l'inter-opérabilité mais reste très bas-niveau



Interfaces Reactive Streams

```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}
```

```
public interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

```
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
}
```



Back pressure

La notion de ***back pressure*** décrit la possibilité des abonnés de contrôler la cadence d'émission des événements du service qui publie.

Reactive Stream permet d'établir le mécanisme et de fixer les limites de cadence.

Si *l'Observable* ne peut pas ralentir, il doit prendre la décision de bufferiser, supprimer ou tomber en erreur.



Reactor

Reactor se concentre sur la programmation réactive côté serveur.

Il est développé conjointement avec Spring.

- Il fournit principalement les types de plus haut niveau ***Mono*** et ***Flux*** représentant des séquences d'événements
- Il offre un ensemble d'opérateurs alignés sur *ReactiveX*.
- C'est une implémentation de *Reactive Streams*



Reactive Spring

Programmation réactive
Spring Reactor
Spring Data Reactive
Spring Webflux



Dépendance Maven

```
<dependency>
```

```
  <groupId>io.projectreactor</groupId>
```

```
  <artifactId>reactor-core</artifactId>
```

```
  <version>${version}</version>
```

```
</dependency>
```



2 Types

Reactor offre principalement 2 types Java :

- ***Mono*** : Flux de 0..1 éléments
- ***Flux*** : Flux de 0..N éléments

Tous les 2 sont des implémentations de l'interface ***Publisher*** de *Reactive Stream* qui définit 1 méthode :

```
void subscribe(Subscriber<? super T> s)
```

Le flux commence à émettre seulement si il y a un abonné

En fonction du nombre possible d'événements publiés, ils offrent des opérateurs différents



Flux

Un ***Flux*** $\langle T \rangle$ représente une séquence asynchrone de 0 à N événements, optionnellement terminée par un signal de fin ou une erreur.

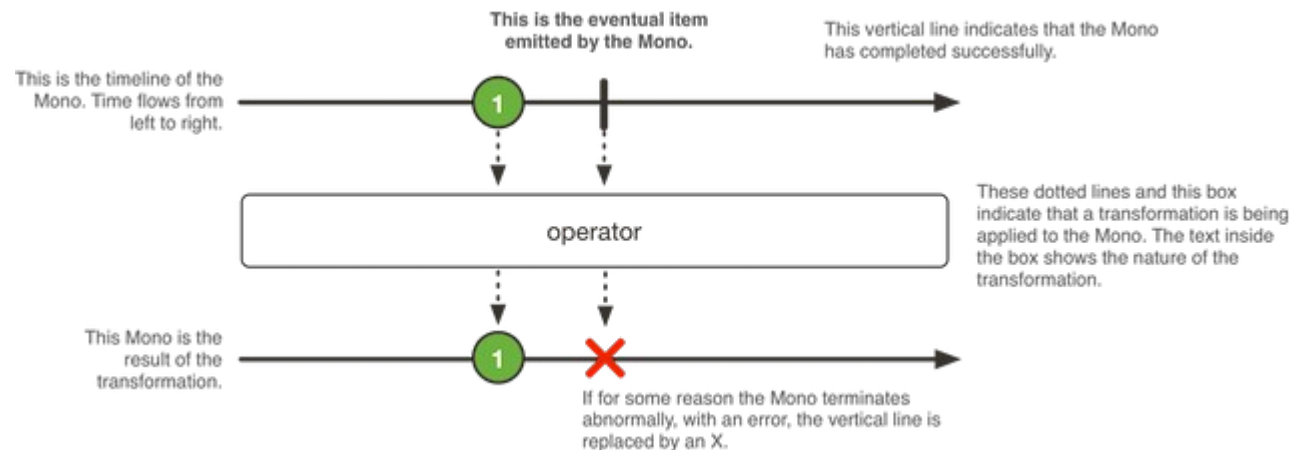
Les événements sont traduits par des appels de méthode sur les abonnés :

- Nouvelle valeur : *onNext()*
- Signal de fin : *onComplete()*
- Erreur : *onError()*

Mono

Mono $\langle T \rangle$ représente une séquence de 0 à 1 événements, optionnellement terminée par un signal de fin ou une erreur

Mono offre un sous-ensemble des opérateurs de Flux





Production d'un flux de données

La façon la plus simple de créer un *Mono* ou un *Flux* est d'utiliser les méthodes *Factory* à disposition.

```
Mono<Void> m1 = Mono.empty()
Mono<String> m2 = Mono.just("a");
Mono<Book> m3 = Mono.fromCallable(() -> new Book());
Mono<Book> m4 = mono.fromFuture(myCompletableFuture);
```

```
Flux<String> f1 = Flux.just("a","b","c");
Flux<Integer> f2 = Flux.range(0, 10);
Flux<Long> f3 = Flux.interval(Duration.ofMillis(1000).take(10);
Flux<String> f4 = Flux.fromIterable(bookCollection);
Flux<Book> f5 = Flux.fromStream(bookCollection.stream());
```



Abonnement

L'abonnement au flux s'effectue via la méthode ***subscribe()***

Généralement, des lambda-expressions sont utilisées

```
subscribe(); // Déclenche le flux
subscribe(Consumer<? super T> consumer);
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer);
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer);
// Chaînage
subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer);
```



Interface *Subscriber*

Sans utiliser les lambda-expression, on peut fournir une implémentation de l'interface ***Subscriber*** qui définit 4 méthodes :

```
void onComplete()  
void onError(java.lang.Throwable t)  
void onNext(T t)  
void onSubscribe(Subscription s)
```

Invoqué après

```
Publisher.subscribe(Subscriber)
```




Subscription

Subscription représente un abonnement d'un (seul) abonné à un *Publisher*.

Il est utilisé

- pour demander l'émission d'événement
`void request(long n)`
- Pour annuler la demande et permettre la libération de ressource
`void cancel()`



Exemple

```
Flux.just(1, 2, 3, 4)
    .log()
    .subscribe(new Subscriber<Integer>() {
        @Override
        public void onSubscribe(Subscription s) {
            s.request(Long.MAX_VALUE); // Provoque l'émission de tous les évts
        }

        @Override
        public void onNext(Integer integer) {
            elements.add(integer);
        }

        @Override
        public void onError(Throwable t) {}

        @Override
        public void onComplete() {}
    });
```



Opérateurs

Les opérateurs permettent différents types d'opérations sur les éléments de la séquence :

- Transformer
- Choisir des événements
- Filtrer
- Gérer des erreurs
- Opérateurs temporels
- Séparer un flux
- Revenir au mode synchrone



Transformation

1 vers 1 :

map (nouvel objet), *cast* (chgt de type), *index* (Tuple avec ajout d'un indice)

1 vers N :

flatMap + une méthode factory, *handle*

Ajouter des éléments à une séquence :

startsWith, *endsWith*

Agréger :

collectList, *collectMap*, *count*, *reduce*, *scan*,

Agréger en booléen :

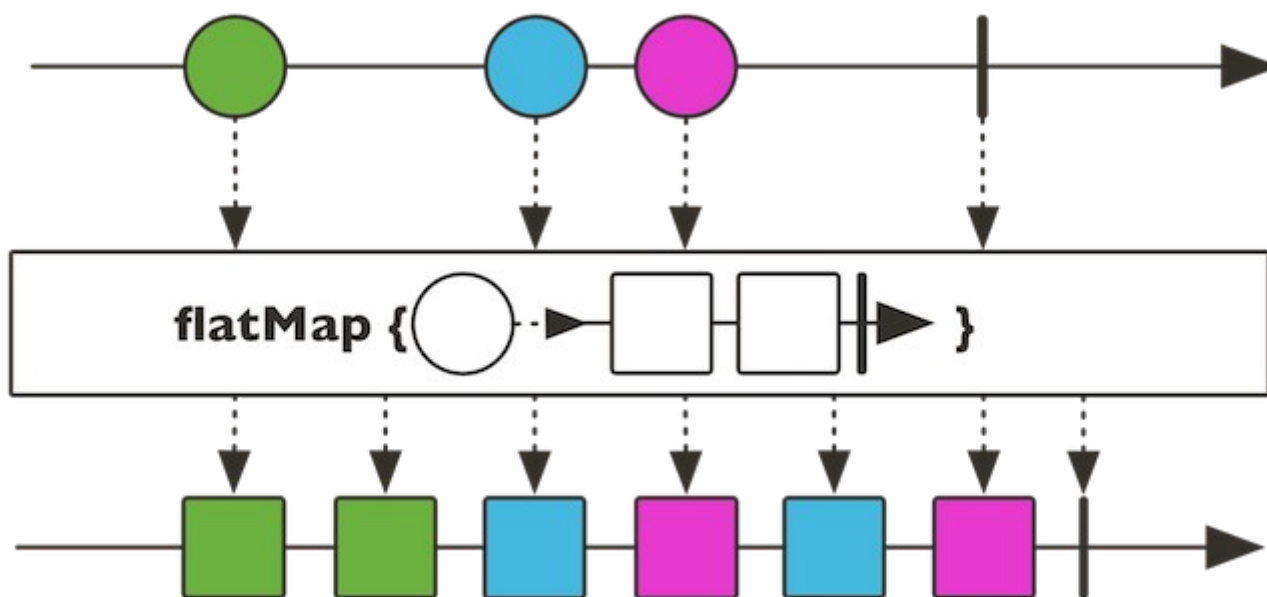
all, *any*, *hasElements*, *hasElement*

Combiner plusieurs flux :

concat, *merge*, *zip*



flatMap





Filtres

Filtre sur fonction arbitraire :

filter

Sur le type :

ofType

Ignorer toutes les valeurs :

ignoreElements

Ignorer les doublons :

distinct

Seulement un sous-ensemble :

take, takeLast, elementAt

Skipper des éléments :

skip(Long | Duration), skipWhile



Opérateurs temporels

Associé l'événement à un timestamp :
elapsed, timestamp

Séquence interrompue si délai trop important
entre 2 événements :
timeout

Séquence à intervalle régulier :
interval

Ajouter des délais
Mono.delay, delayElements, delaySubscription



Reactive Spring

Programmation réactive
Spring Reactor
Spring Data Reactive
Spring Webflux



Introduction

La programmation réactive s'invite également dans SpringData

Attention, cela ne concerne pas JDBC et JPA qui restent des APIs bloquantes

Sont supportés :

- MongoDB
- Cassandra
- Redis



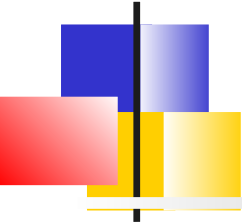
Accès réactifs aux données persistante

Les appels sont asynchrones, non bloquants, pilotés par les événements

Les données sont traitées comme des flux

Cela nécessite :

- Spring Reactor
- Spring Framework 5
- Spring Data 2.0
- Un pilote réactif (Implémentation NoSQL exclusivement)
- Éventuellement Spring Boot (2.0)



Mélange bloquant non-bloquant

S'il faut mélanger du code bloquant et non bloquant, il ne faut pas bloquer la thread principale exécutant la boucle d'événements.

On peut alors utiliser les *Scheduler* de Spring Reactor.



Apports

La fonctionnalité Reactive reste proche des concepts SpringData :

- API de gabarits réactifs (Reactive Templates)
- Repository réactifs
- Les objets retournées sont des Flux ou Mono



Reactive Template

L'API des classes *Template* devient :

```
<T> Mono<T> insert(T objectToSave)
<T> Mono<T> insert(Mono<T> object)
<T> Flux<T> insertAll(Collection<? extends T>
    objectsToSave)
<T> Flux<T> find(Query query, Class<T> type
...

```

Exemple :

```
Flux<Person> insertAll = template
.insertAll(Flux.just(new Person("Walter", "White", 50), //
new Person("Skyler", "White", 45), //
new Person("Saul", "Goodman", 42), //
new Person("Jesse", "Pinkman", 27)).collectList());

```



Reactive Repository

L'interface ***ReactiveCrudRepository<T,ID>*** permet de profiter d'implémentations de fonction CRUD réactives.

Par exemple :

```
Mono<Long> count()  
Mono<Void> delete(T entity)  
Flux<T> findAll()  
Mono<S> save(S entity)  
..
```



Requêtes

De plus comme dans SpringData, les requêtes peuvent être déduites du nom des fonctions :

```
public interface ReactivePersonRepository extends
    ReactiveCrudRepository<Person, String> {

    Flux<Person> findByLastname(String lastname);

    @Query("{ 'firstname': ?0, 'lastname': ?1}")
    Mono<Person> findByFirstnameAndLastname(String firstname, String
    lastname);

    // Accept parameter inside a reactive type for deferred execution
    Flux<Person> findByLastname(Mono<String> lastname);

    Mono<Person> findByFirstnameAndLastname(Mono<String> firstname, String
    lastname);
}
```



Exemple dépendance *MongoDB* avec *SpringBoot*

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.BUILD-SNAPSHOT</version>
</parent>
<!-- Ramène en particulier : spring-data-mongodb et reactor-core -->
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
  </dependency>
</dependencies>
```




Reactive Spring

Programmation réactive
Spring Reactor
Spring Data Reactive
Spring Webflux



Motivation

2 principales motivations pour Spring Webflux :

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



Introduction

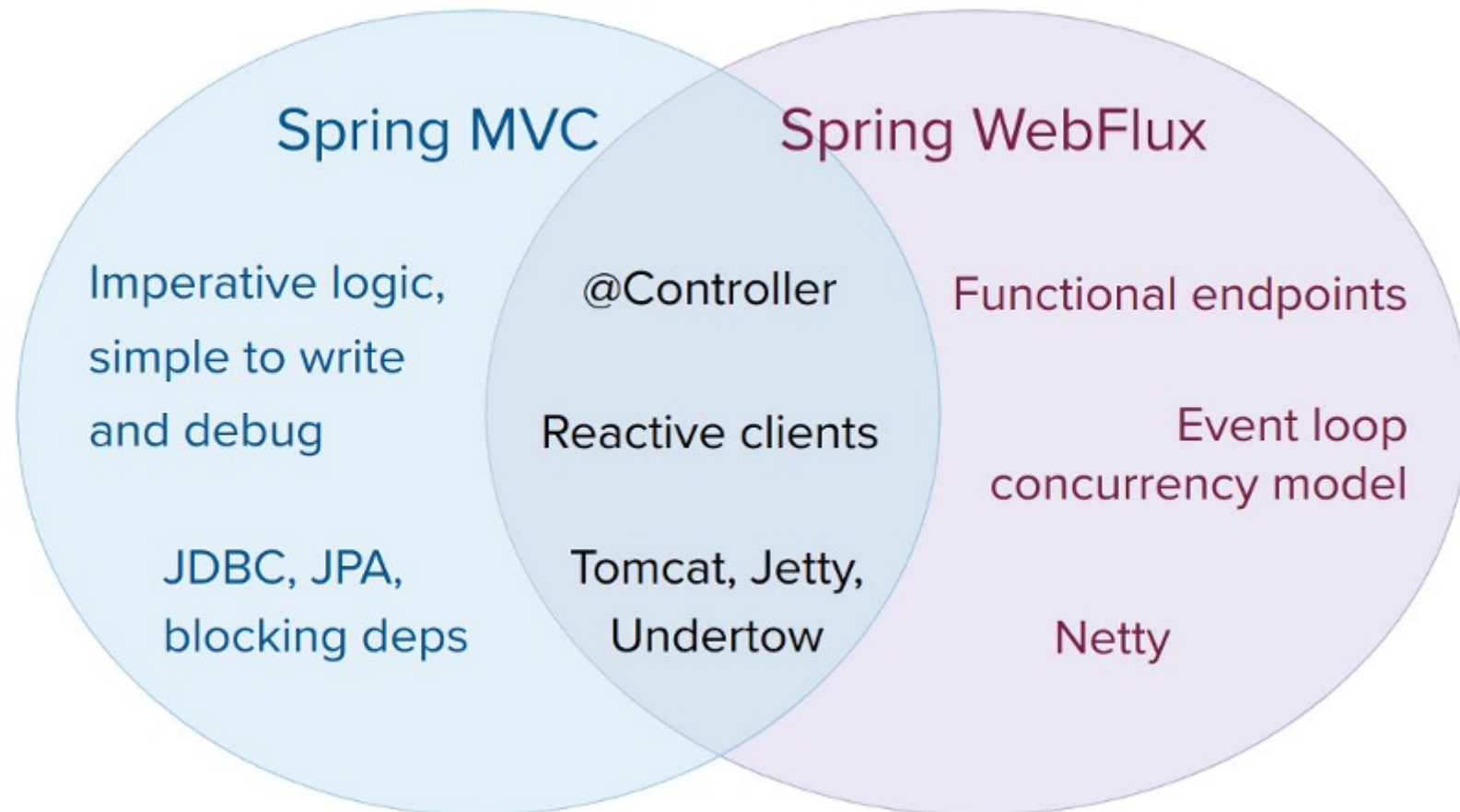
Le module *spring-web* est la base pour Spring Webflux.

Il offre en plus 2 modèles de programmation :

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



MVC et WebFlux





Serveurs

Spring WebFlux est supporté sur

- Tomcat, Jetty, et les conteneurs de Servlet 3.1+,
- Ainsi que les environnements non-Servlet comme Netty ou Undertow

Le même modèle de programmation est supporté sur tous ces serveurs

Avec *SpringBoot*, la configuration par défaut démarre un serveur embarqué Netty



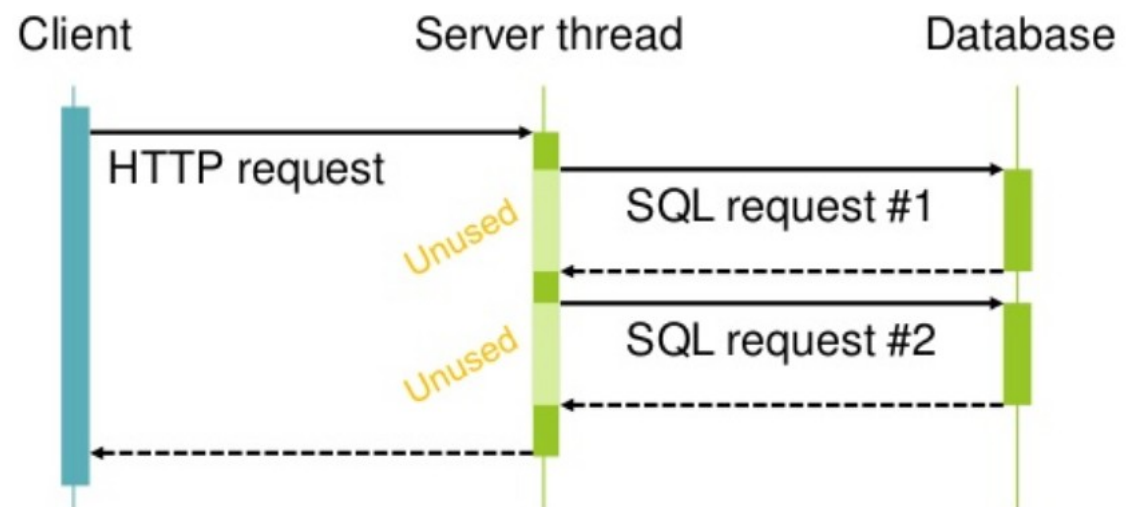
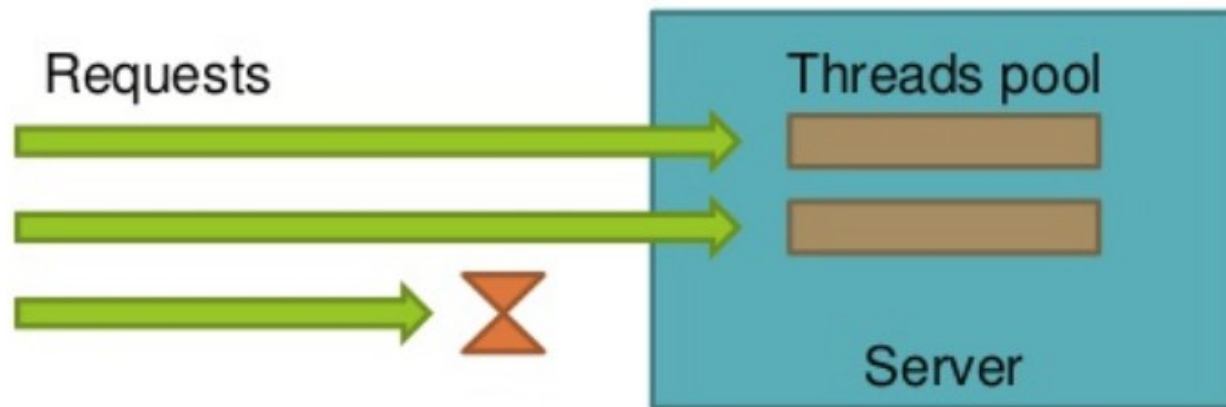
Performance et Scaling

Le modèle réactif et non bloquant n'apporte pas spécialement de gain en terme de temps de réponse. (il y a plus de chose à faire et cela peut même augmenter le temps de traitement)

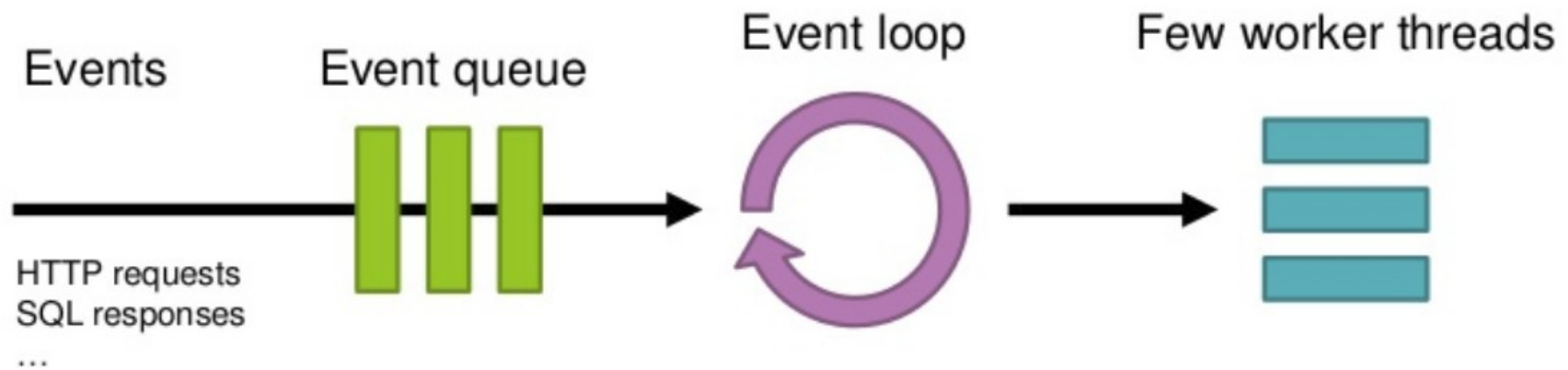
Le bénéfice attendu est la possibilité de **scaler** avec un petit nombre de threads fixes et moins de mémoire. Cela rend les applications plus résistantes à la charge.

Pour pouvoir voir ces bénéfices, il est nécessaire d'introduire de la latence, par exemple en introduisant des IO réseaux lents ou non prédictibles.

Modèle bloquant



Modèle non bloquant





Modèle de threads

Pour un serveur *Spring WebFlux* entièrement réactif, on peut s'attendre à 1 thread pour le serveur et autant de threads que de CPU pour le traitement des requêtes.

Si on doit accéder à des données JPA ou JDBC par exemple, il est conseillé d'utiliser des *Schedulers* qui modifie alors le nombre de threads

Pour configurer le modèle de threads du serveur, il faut utiliser leur API de configuration spécifique ou voir si *Spring Boot* propose un support.



Généralités API

En général l'API WebFlux

- accepte en entrée un *Publisher*,
- l'adapte en interne aux types *Reactor*,
- l'utilise et retourne soit un *Flux*, soit un *Mono*.

En terme d'intégration :

- On peut fournir n'importe quel *Publisher* comme entrée
- Il faut adapter la sortie si l'on veut quelle soit compatible avec une autre librairie que *Reactor*



Contrôleurs annotés

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

Les différences sont :

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



Example

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



Méthodes des contrôleurs

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

– Arguments :

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

– Valeurs de retour :

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

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



Endpoints fonctionnels

Dans ce modèle de programmation fonctionnelle, les fonctions (lambda-expression) sont utilisées pour router et traiter les requêtes.

Les interfaces représentant l'interaction HTTP (requête/réponse) sont immuables

=> Thread-safe nécessaire pour le modèle réactif



ServerRequest et ServerResponse

ServerRequest et ***ServerResponse*** sont donc des interfaces qui offrent des accès via lambda-expression aux messages HTTP.

- ***ServerRequest*** expose le corps de la requête comme Flux ou Mono.
Elle donne accès aux éléments HTTP (Méthode, URI, ..) à travers une interface séparée *ServerRequest.Headers*.
`Flux<Person> people = request.bodyToFlux(Person.class);`
- ***ServerResponse*** accepte tout *Publisher* comme corps.
Elle est créée via un builder permettant de positionner le statut, les entêtes et le corps de réponse
`ServerResponse.ok()
 .contentType(MediaType.APPLICATION_JSON).body(person);`



Traitement des requêtes via *HandlerFunction*

Les requêtes HTTP sont traitées par une ***HandlerFunction*** : une fonction qui prend en entrée un *ServerRequest* et fournit un *Mono<ServerResponse>*

Exemple :

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

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



Example

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



Mapping via *RouterFunction*

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

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

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

Exemple :

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



Combinaison

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

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

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

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



Example

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

```
RouterFunction<ServerResponse> personRoute = RouterFunctions.  
    route(RequestPredicates.GET("/person/{id}"))  
        .and(accept(APPLICATION_JSON)), handler::getPerson)  
    .andRoute(RequestPredicates.GET("/person"))  
        .and(accept(APPLICATION_JSON)), handler::listPeople)  
    .andRoute(RequestPredicates.POST("/person"))  
        .and(contentType(APPLICATION_JSON)), handler::createPerson);
```



Exécution sur un serveur

Pour exécuter une *RouterFunction* sur un serveur, il faut le convertir en *HttpHandler*.

Différentes techniques sont possibles mais la + simple consiste à spécialiser une configuration **WebFlux**

- La configuration par défaut apportée par **@EnableWebFlux** fait le nécessaire
- ou directement via SpringBoot et le starter *webflux*



Exemple

@Configuration

@EnableWebFlux

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



Filtres via *HandlerFilterFunction*

Les routes contrôlées par un fonction de routage peuvent être filtrées :

```
RouterFunction.filter(HandlerFilterFunction)
```

HandlerFilterFunction est une fonction prenant une *ServerRequest* et une *HandlerFunction* et retourne une *ServerResponse*.

Le paramètre *HandlerFunction* représente le prochain élément de la chaîne : la fonction de traitement ou la fonction de filtre.



Exemple :

Basic Security Filter

```
import static org.springframework.http.HttpStatus.UNAUTHORIZED;

SecurityManager securityManager = ...
RouterFunction<ServerResponse> route = ...

RouterFunction<ServerResponse> filteredRoute =
    route.filter((request, next) -> {
        if (securityManager.allowAccessTo(request.path())) {
            return next.handle(request);
        }
        else {
            return ServerResponse.status(UNAUTHORIZED).build();
        }
    });
```




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



Format YAML

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

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



Exemple *.yml*

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

Produit :

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



Listes

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

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

Devient :

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