



Démarrer avec Spring et Spring Boot

David THIBAU – 2023

david.thibau@gmail.com



Agenda

- **Introduction**

- Historique
- IoC et Dependency Injection

- **Annotations**

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

- **Spring Reactive**

- Introduction

- **Spring Boot**

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

- **Spring et la persistance**

- Spring Data
- Spring Data JPA

- **APIs REST**

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

- **Interactions entre services**

- Rest Client
- Messaging

- **Spring Security**

- Principes, Modèles stateful/stateless
- Autoconfiguration Spring Boot
- OpenIdConnect et oAuth2

- **Spring et les tests**

- Spring Test
- @SpringBootTest
- Les tests auto-configurés

- **Annexes**

- Actuator
- Déploiement MongoDB
- Spring MVC
- Spring Reactive



Introduction

Historique IoC et Dependency Injection



Historique et version

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

What can Spring do?



Microservices

Quickly deliver production-grade features with independently evolvable microservices.



Reactive

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



Cloud

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



Web apps

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



Serverless

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



Event Driven

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



Batch

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



Projets Spring

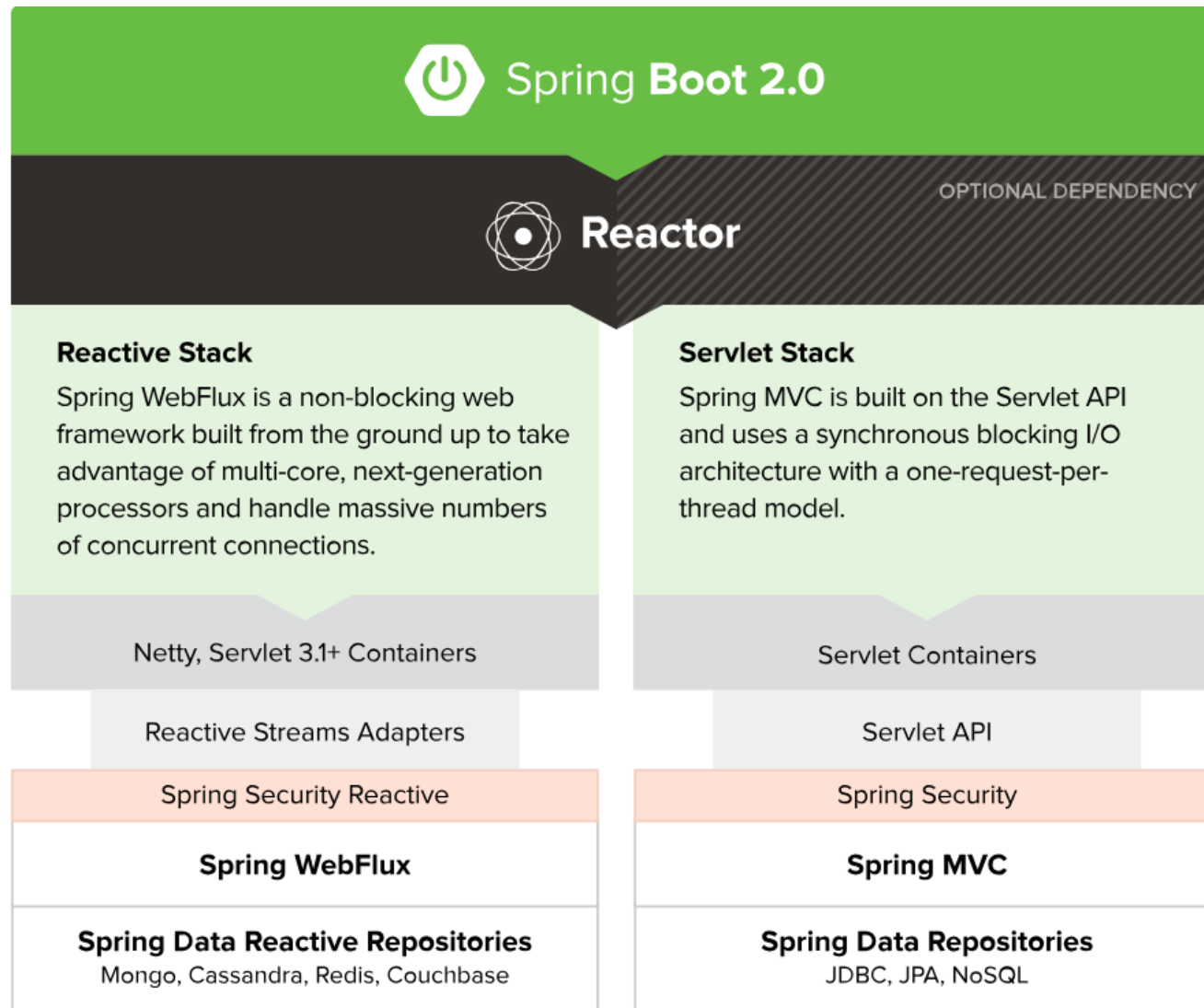
Spring est donc un ensemble de projets adaptés à toutes les problématiques actuelles.

Tous ces projets ont comme objectifs :

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

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

Stacks Web





Introduction

Historique
IoC et Dependency Injection



Pattern IoC

❖ Le problème :

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

❖ Réponse de tout développeur POO :

En définissant une interface, bien sûr



Illustration

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

- ❖ L'interface :

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

- ❖ La classe contrôleur :

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



Implémentation ?

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

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



IoC et Framework

Le pattern ***IoC (Inversion Of Control)*** signifie que ce n'est plus le code du développeur qui a le contrôle de l'exécution mais le framework

Le framework est alors responsable d'instancier les objets, d'appeler les méthodes, de libérer les objets, etc..

Le code du développeur est réduit au code métier.



IoC versus Dependency Injection

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



Types d'injection de dépendances

- ❖ Il y a différentes façons d'injecter les dépendances :
 - **Injection par constructeur** : Les attributs sont initialisés dans le constructeur
 - **Injection par méthode setter** : Les attributs sont initialisés via une méthode setter
 - **Injection par méthode** : Les attributs sont initialisés via une méthode spécifique
 - **Injection par attributs** : Les attributs sont directement mis à jour par le framework



Injection par constructeur

- ❖ La classe *MovieController* doit déclarer un constructeur permettant d'initialiser l'attribut *MovieFinder*.

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

- ❖ L'objet *finder* se fait injecter une propriété via le constructeur : le nom du fichier .

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



Configuration du framework

- ❖ La configuration du framework consiste à lui préciser les classes qu'il doit gérer (beans) et comment il injecte les dépendances.
- ❖ Cela s'effectue :
 - Via des fichiers de configuration XML. (Legacy)
 - Via des classes Java de configuration et des annotations Java (Depuis Java5)
 - En appliquant des configurations par défaut (SpringBoot)



Configuration XML

```
<beans>
```

```
  <bean id="movieController" class="spring.MovieController">
```

```
    <property name="finder" ref="movieFinder"/>
```

```
  </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");

    MovieController lister =
        (MovieController)ctx.getBean("movieController");

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



Aspects

Lors de la configuration, il est également possible d'ajouter des services techniques/transverses aux objets gérés par le framework :

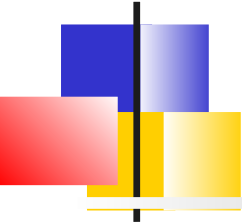
- Sécurité : *@RolesAllowed*
- Transaction : *@Transactional*
- ..



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 protégée par des ACLs sans utiliser de code de l'API de sécurité
- Définir une méthode gestionnaire de message sans utiliser l'API du broker



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 et des classes de configuration** Java pour définir des beans à instancier.
- ❖ Chaque approche a ses pour et ses contres. Elles peuvent également être combinées.

Dans les faits, la configuration via Java est beaucoup plus pratique et s'est donc imposée.



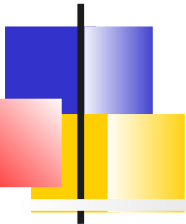
Classes de configuration

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

- Ces classes sont constituées de méthodes annotées par **@Bean** quiinstancient les implémentations d'interface
- Elles sont généralement utilisées pour instancier les beans d'intégration aux autres systèmes (DataSource, Client Elasticsearch,)

@Configuration

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



Composition de configuration

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

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

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

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

1. On pouvait également le faire en XML



Recherche des annotations

2 alternatives afin que le framework traite les annotations :

- Lui indiquer l'emplacement des classes annotées
- Lui indiquer un package à scanner

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



Configuration du framework via les annotations

Indication d'une classe de *@Configuration*

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

Scan de package :

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



Attributs de *@Bean*

@Bean a 3 attributs :

name : les alias du bean

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

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

@Configuration

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



Annotations *@Enable*

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

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

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



Exemples *@Enable*

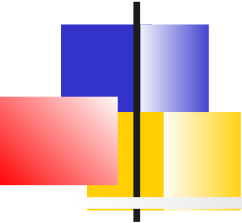
@EnableWebMvc : Instancie tous les beans nécessaires à Spring MVC

@EnableCaching : Configuration par défaut d'un cache. Permet d'utiliser les annotations *@Cachable*, ...

@EnableScheduling : Configuration par défaut d'un scheduler. Permet d'utiliser les annotations *@Scheduled*

@EnableJpaRepositories : Permet de scanner les classe Repository

...



Configuration via les annotations

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



Introduction

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

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

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



Exemple

@Component

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

@Component

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




@ComponentScan

Spring peut détecter automatiquement les classes annotées

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

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

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



Stereotypes

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

Spring introduit d'autres stéréotypes :

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

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

Par ex : La sérialisation JSON pour les @RestController



Cycles de vie

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

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



@Scope

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

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

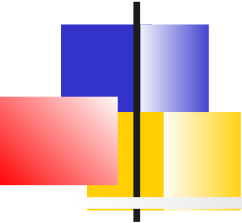


Méthodes de call-back

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

@Component

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



Configuration via les annotations

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



@Autowired

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

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

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



Examples

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




Examples (2)

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



Injection implicite

Dans les dernières versions de Spring, l'annotation *@Autowired* n'est plus nécessaire lorsque l'on utilise l'injection par constructeur

- En général, l'attribut est alors déclaré comme ***final***

C'est ce qu'on appelle l'injection implicite¹, c'est une injection par type

1. On retrouve la même idée dans le framework Angular

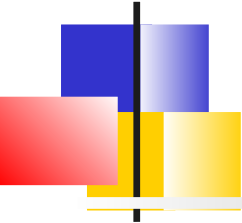


Injection implicite

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

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

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



Exceptions dues à l'auto-wiring

@Autowired peut provoquer des exceptions au démarrage de Spring.

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

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

- Cas 2 : Spring trouve plusieurs Beans du type demandé

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



@Resource, @Qualifier

@Resource et **@Qualifier** permettent d'injecter un bean par son nom.

- Les annotations spécifient alors le nom du bean
- Si le nom n'est pas précisé, le nom du bean à injecter correspond au nom de la propriété



Exemples

```
public class MovieRecommender {  
  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao cpDao;  
  
    // Le nom du bean recherché est "context"  
    @Resource  
    private ApplicationContext context;  
  
    // Plusieurs beans implémentent Formatter  
    @Autowired  
    @Qualifier("fooFormatter")  
    private Formatter formatter;  
  
    public MovieRecommender() {  
    }  
    // ...  
}
```



Annotations JSR 330

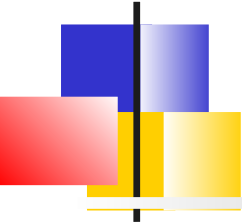
Équivalence

Spring supporte également les annotations de JSR 330 utilisées dans la spécification JakartaEE

@javax.inject.Inject correspond à
@Autowired

@javax.inject.Named correspond à
@Component

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



Configuration via les annotations

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



Introduction

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

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

Cela nécessite la présence d'un bean

PropertySourcesPlaceholderConfigurer¹

1. Disponible automatiquement dans un contexte SpringBoot



Exemple

@Configuration

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

public class AppConfig {

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

Integer myIntProperty ;

@Autowired

Environment env;

@Bean

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

}

@Bean

public TestBean testBean() {

TestBean testBean = new TestBean();

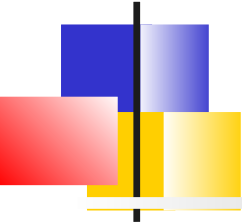
testBean.setIntProperty(myIntProperty) ;

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

return testBean;

}

}



Configuration via les annotations

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



Environment

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

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



Annotations utilisant les profils

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

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

    // ...

}

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

    // ...

}
```



Exemple

Bd de dév et BD de prod

@Configuration

@Profile("development")

```
public class StandaloneDataConfig {

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

@Configuration

@Profile("production")

```
public class JndiDataConfig {

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



Activation d'un profil

Programmatically :

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

Ligne de commande :

Propriété Java

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

Argument SpringBoot

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



Spring Reactif

Introduction



Pattern et *ReactiveX*

La programmation réactive :

- se base sur le pattern **Observable** qui est une combinaison des patterns *Observer* et *Iterator*
- utilise la programmation fonctionnelle pour définir facilement des opérateurs sur le flux
- est formalisée par l'API **ReactiveX**.
De nombreuses implémentations existent (*RxJS*, *RxJava*, *Rx.NET*)



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
- Il ne définit pas les opérateurs (chaque implémentation à ses propres opérateurs)
- Une implémentation existe dans Java9 : *Flow API*



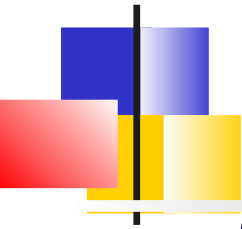
Interfaces Reactive Streams

```
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> {  
}
```

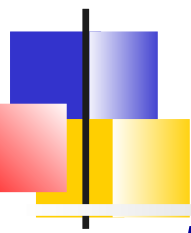


Reactor

Reactor se concentre sur la programmation réactive côté serveur.

Il est développé conjointement avec Spring.

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



Eco-système Reactive Spring

Reactor est la brique de base de la pile réactive de Spring.

Il est utilisé dans tous les projets réactifs :

- **Spring WebFlux** des services web back-end réactifs. Alternative à *Spring MVC*
- **Spring Data** : *MongoDB, Cassandra, R2DBC*
- **Spring Reactive Security** : Sécurité réactive
- **Spring Test** : Support pour le réactif
- **Spring Cloud** : Projets micro-services. En particulier *Spring Cloud Data Stream, Spring Gateway, ...*



2 Types

Reactor offre principalement 2 types Java :

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

Ce sont des implémentations de ***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



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.

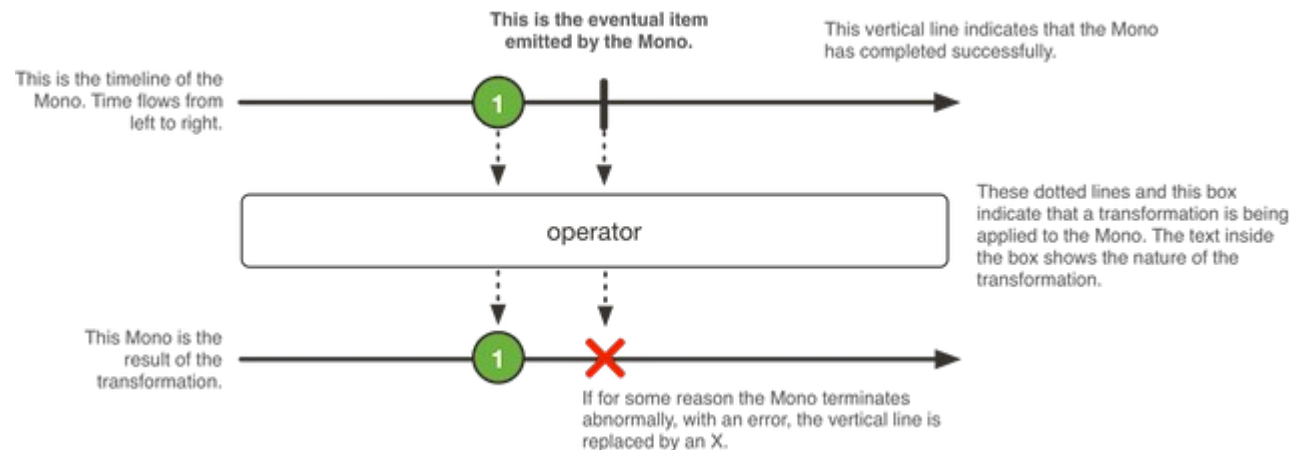
Les événements sont traduits par des appels de méthode sur les abonnés :

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

Mono

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

Mono offre un sous-ensemble des opérateurs de Flux





Exemple Reactor

```
userService.getFavorites(userId)
    .flatMap(favoriteService::getDetails)
    .switchIfEmpty(suggestionService.getSuggestions())
    .take(5)
    .subscribe(uiList::show, UiUtils::errorPopup);
```



SpringBoot

L'auto-configuration

Starters SpringBoot

Structure projet et principales
annotations

Propriétés de configuration



Introduction

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

- Ne nécessite aucune configuration préalable :

Dés la première ligne de code, on a une application fonctionnelle

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



Auto-configuration

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

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

- Permet de démarrer rapidement avec une configuration par défaut puis de graduellement surcharger la configuration pour ses besoins spécifiques

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



Auto-configuration (Java)

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

- Par exemple, si il s'aperçoit que des librairies Web sont présentes, il démarre un serveur Tomcat embarqué sur le port 8080 et applique la configuration par défaut de Spring MVC ou Spring Webflux
- Si il s'aperçoit que le driver H2 est dans le classpath, il crée automatiquement un pool de connexions vers la base
- Etc...



Personnalisation de la configuration

La configuration par défaut peut être surchargée par différents moyens

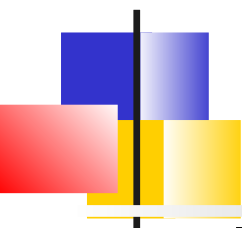
- Les propriétés qui modifient les valeurs par défaut des beans techniques via :
 - Des **variables d'environnement**
 - Des **arguments de la ligne de commande**
 - Des **fichiers de configuration externe** (*.properties* ou *.yml*).
Différents fichiers peuvent être activés en fonction de profils
-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



Autres apports de SpringBoot

En dehors de l'auto-configuration, SpringBoot offre d'autres bénéfices pour les développeurs :

- Simplification de la gestion des dépendances vers les librairies OpenSource via les **starters**
- Assistant de création de projet avec **SpringInitializer**
- Point central de la configuration des propriétés avec ***application.properties/yml***
- Plugins pour les **IDEs** (SpringTools suite ou autre)
- Plugins pour les **outils de build** (Gradle et Maven)



Gestion des dépendances

La gestion des dépendances est simplifiée grâce aux **starters**.

Ce sont des groupes de dépendances permettant d'intégrer une technologie ou apportant une fonctionnalité qui sont déclarés dans le fichier de build

=> Pour les développeurs, ce mécanisme simplifie à l'extrême la gestion des versions des dépendances.

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

https://start.spring.io/



Project

☒ Gradle - Groovy ☐ Gradle - Kotlin ☐ Maven

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 3.2.0 (SNAPSHOT) ☐ 3.2.0 (M1) ☐ 3.1.3 (SNAPSHOT) ☒ 3.1.2 ☐ 3.0.10 (SNAPSHOT) ☐ 3.0.9 ☐ 2.7.15 (SNAPSHOT) ☐ 2.7.14

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 20 ☒ 17 ☐ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... CTRL + B

Spring Reactive Web

WEB

Build reactive web applications with Spring WebFlux and Netty.

Spring for Apache Kafka

MESSAGING

Publish, subscribe, store, and process streams of records.



Fichiers générés par l'assistant

- *.gitignore, Help.md*
- Scripts de build (*mvnw* ou *gradlew*)
- Classe de démarrage de SpringBoot (*src/main/java*)
- Classe de test de la configuration (*src/test/java*)
- Fichier de configuration des propriétés (*src/main/resources*)



Exemple Gradle

```
plugins {  
    id 'java'  
    id 'org.springframework.boot' version '3.1.2'  
    id 'io.spring.dependency-management' version '1.1.2'  
}  
  
group = 'org.formation'  
version = '0.0.1-SNAPSHOT'  
  
java {  
    sourceCompatibility = '17'  
}  
  
repositories { mavenCentral() }  
  
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-webflux'  
    implementation 'org.springframework.kafka:spring-kafka'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
    testImplementation 'io.projectreactor:reactor-test'  
    testImplementation 'org.springframework.kafka:spring-kafka-test'  
}  
  
tasks.named('test') { useJUnitPlatform() }
```



Plug-in Maven/Gradle de SpringBoot

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

- Ce sont des wrappers de l'outil de build garantissant que tous les développeurs utilisent la même version de l'outil de build.

La commande la + importante dans un contexte Maven :

./mvnw clean package

=> Génère un *fat-jar*

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

java -jar target/myAppli.jar

Avec gradle :

`gradle build`

`java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar`



Autres tâches de build

Packager une image OCI :

spring-boot:build-image

Exécuter l'application

spring-boot:run

Exécuter avec le classpath de tes

spring-boot:test-run

Construire une image native

package -Pnative

Exécuter des tests d'intégration

spring-boot:start

spring-boot:stop

Renseigner le endpoint */actuator/info* avec les informations de build

spring-boot:build-info



Classe de démarrage

La classe de démarrage est annotée via **@SpringBootApplication**, annotation qui englobe :

- **@Configuration** : Permet de définir des méthodes *@Bean*
- **@ComponentScan** : Scan des annotations dans les sous-packages
- **@EnableAutoConfiguration** : Activation du mécanisme d'auto-configuration de SpringBoot

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        // Création du contexte Spring
        SpringApplication.run(DemoApplication.class, args);
    }

}
```



Classe de test

La classe de test est annotée via

@SpringBootTest :

- Permet de créer le contexte Spring avant l'exécution du test junit5

```
@SpringBootTest
class DemoApplicationTests {

    @Test
    void contextLoads() {
        // Si le test passe, la configuration SpringBoot est OK
    }

}
```



SpringBoot

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



Starters les + importants

Web

- *-**web** : Application web ou API REST. Framework *Spring MVC*
- *-**reactive-web** : Application web ou API REST en mode réactif. Framework *Spring WebFlux*

Cœurs :

- *-**logging** : Utilisation de logback (Tjs présent)
- *-**test** : Test avec Junit, Hamcrest et Mockito (Tjs présent)



Starters développement

- *-**devtools** : Fonctionnalités pour le développement
- *-**lombok** : Simplification du code Java
- *-**configuration-processor** : Complétion des propriétés de configuration applicatives disponibles dans l'IDE
- *-**docker-compose**¹ : Support pour démarrer les services de support via docker-compose (BD, Kafka, etc..)
- *-**graalvm-native**¹ : Support pour construire des images natives
- *-**modulith**¹ : Support pour construire des applications monolithes modulaires



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 SQL

jdbc : JDBC avec pool de connexions Tomcat

Spring Data JPA : Spring Data avec Hibernate et JPA

Spring Data JDBC : Spring Data avec jdbc

Spring Data R2DBC : Spring Data avec *jdbc reactif*

MyBatis : Framework MyBatis

LiquiBase Migration : Migration de schéma avec Liquibase

Flyway Migration : Migration de schéma avec Flyway

JOOQ Access Layer : API fluent pour construire des requêtes SQL

*-***<drivers>*** : Accès aux driver JDBC (MySQL, Postgres, H2, HyperSonic, DB2)



Starters NOSQL

- *-**data-cassandra**, *-**data-reactive-cassandra**: Base distribuée Cassandra
- *-**data-neo4j** : Base de données orienté graphe de Neo4j
- *-**data-couchbase** *-**data-reactive-couchbase** : Base NoSQL CouchBase
- *-**data-redis** *-**data-reactive-redis** : Base NoSQL Redis
- *-**data-geode** : Stockage de données via Geode
- *-**data-elasticsearch** : Base documentaire indexée ElasticSearch
- *-**data-solr**: Base indexée SolR
- *-**data-mongodb** *-**data-reactive-mongodb** : Base NoSQL MongoDB



Messaging

- *-**integration**: Spring Integration (Abstraction de + haut niveau pour implémenter des patterns d'intégration de façon déclaratif)
- *-**kafka**: Intégration avec Apache Kafka
- *-**kafka-stream**: Intégration avec l'API KafkaStream
- *-**rabbitmq**: Intégration avec Rabbit MQ
- *-**activemq5** : ActiveMQ avec JMS
- *-**artemis** : ApacheMQ avec Artemis
- *-**pulsar**, *-**reactive-pulsar** : Messagerie PULSAR
- *-**websocket** : Servlet avec STOMP et SockJS
- *-**rsocket** : SpringMessaging et Netty
- *-**camel** : Intégration avec Apache Camel
- *-**solacePubSub** : Intégration avec Solace



Autres Starters Web

Moteur de templates HTML

- *-**thymeleaf** : *Spring MVC avec des vues Thymeleaf*
- *-**mustache** : *Spring MVC avec Mustache*
- *-**groovy-templates** : *Spring MVC avec gabarits Groovy*
- *-**freemarker** : *Spring MVC avec freemarker*

Autres

- *-**graphql** : *API GraphQL*
- *-**rest-repository, restrepository-explorer, *-hateoas** :
Génération API Rest à partir des repositories de Spring Data
- *-**jersey** : *API Restful avec JAX-RS et Jersey*
- *-**webservices** : *Services SOAP*
- *-**vaadin, *-hila** : *Framework pour applis web*



Autres Starters

I/O

- *-**batch** : Gestion de batchs
- *-**mail** : Envois de mails
- *-**cache** : Support pour un cache
- *-**quartz** : Intégration avec Scheduler
- *-**shell** : Support pour commandes en ligne

Ops

- *-**actuator** : Points de surveillance REST ou JMX
- *-**spring-boot-admin (client et serveur)** : UI au dessus d'actuator
- *-**sentry** : Intégration sentry (monitoring performance)



Spring Cloud

Services cloud : Facilité de déploiement

Amazon, Google Cloud, Azure, Cloud Foundry, Alibaba

Architecture Micro-services

Services de discovery, de configuration externalisée,
de répartition de charge, de gateway, de circuit
breaker

Spring Cloud Contract : Génération de tests et mock
servers

Service de monitoring, de tracing, etc ...



Observabilité

Depuis la version 3.x, SpringBoot s'appuie fortement sur ***Micrometer***.

- Des starters permettent de publier les métriques *micrometer* vers des système de visualisation :
DataDog, Dynatrace, Influx, Graphite, New Relic, Prometheus, Wavefront
- D'autres starters permettent la tracabilité des requêtes dans les architecture micro-services :
Brave et Zipkin



SpringBoot

L'auto-configuration

Starters SpringBoot

**Structure projet et principales
annotations**

Propriétés de configuration



Structure projet

Aucune obligation mais des recommandations :

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



Structure typique

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



@Configuration

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

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



Auto-configuration

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

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

Ex :

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



Beans et Injection de dépendance

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

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



Exemple

```
package com.example.service;

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

@Service
public class DatabaseAccountService implements AccountService {

    private final RiskAssessor riskAssessor;

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

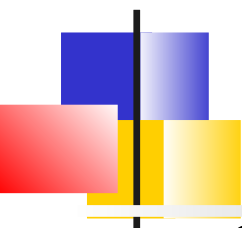
    // ...

}
```



SpringBoot

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



Propriétés de configuration

Spring Boot permet d'externaliser la configuration des beans :

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

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

Les valeurs des propriétés sont ensuite injectées dans les beans :

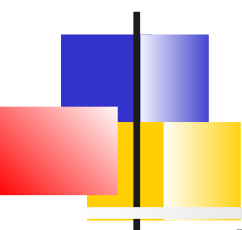
- Directement via l'annotation **@Value**
- Ou associer à un objet structuré via l'annotation **@ConfigurationProperties**



Priorités

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

1. *spring-boot-devtools.properties* si *devtools* est activé (SpringBoot)
2. Les propriétés de test
3. **La ligne de commande. Ex : `--server.port=9000`**
4. Environnement REST, Servlet, JNDI, JVM
5. **Variables d'environnement de l'OS. Ex : `export SERVER_PORT=8000`**
6. Propriétés avec des valeurs aléatoires
7. **Propriétés spécifiques à un profil**
8. ***application.properties* , *yaml***
9. Annotation *@PropertySource* dans la configuration
10. Les propriétés par défaut spécifiées par *SpringApplication.setDefaultProperties*



application.properties (.yml)

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

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

En respectant ces emplacements standards, SpringBoot les trouve tout seul



Valeur filtrée

Les fichiers supportent les valeurs filtrées.

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

Les valeurs aléatoires :

```
my.secret=${random.value}  
my.number=${random.int}  
my.bignumber=${random.long}  
my.uuid=${random.uuid}
```



Injection de propriété : *@Value*

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

```
@Value("${my.property}")  
private Integer 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 validation des propriétés de configuration à l'initialisation du conteneur.

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



Exemple

@Component

@ConfigurationProperties("app")

@Validated

public class MyAppProperties {

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

private String adminContactNumber;

@Min(1)

private int refreshRate;

.....

}



Propriétés spécifiques à un profil

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

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



Exemple fichier *.yml*

```
server:
  address: 192.168.1.100
- - -
spring:
  config:
    activate:
      on-profile:
        -prod
server:
  address: 192.168.1.120
```



Activation des profils

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

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

Plusieurs profils peuvent être activés simultanément



Persistence

Principes de SpringData SpringData JPA



Introduction

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

Spring Data est donc le projet qui encadre de nombreux sous-projets spécialisés sur une API de persistance (jdbc, JPA, Mongo, ...)



Apports de *SpringData*

Les apports sont :

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



Interfaces *Repository*

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

L'interface prend en arguments de type

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

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

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



Interface *CrudRepository*

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

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

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

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



Déduction de la requête

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

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

Spring doit déduire les requêtes à effectuer :

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



Exemple

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

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

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

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

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

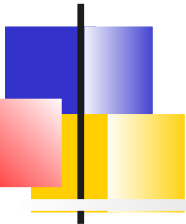


Limite

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

Elles peuvent être précisées avec un numérique

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

Mots-clés supportés pour JPA

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



Utilisation des *NamedQuery* JPA

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

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

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



Utilisation de *@Query*

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

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

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



Persistence

Principes de SpringData
SpringData JPA



Apports Spring Boot

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

- Hibernate
- Spring Data JPA .
- Spring ORMs

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

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

Rappels : Classes entités et associations

@Entity

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

@Entity

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



Configuration source de données / Rappels

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

Une data source se configure via :

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



Support pour une base embarquée

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

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

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




Base de production

Les bases de production peuvent également être auto-configurées.

Les propriétés requises à configurer sont :

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

`spring.datasource.username=dbuser`

`spring.datasource.password=dbpass`

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

Voir *DataSourceProperties* pour l'ensemble des propriétés disponibles

L'implémentation du pool sous-jacent est 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
```



Mode *create, create-drop*

Le mode *create, create-drop* est très pratique pour le développement et le test

La base est recréée à chaque démarrage et il est possible d'exécuter un script d'initialisation des données (Par défaut *import.sql* à la racine du classpath)

=> La base est dans un état connu à chaque démarrage



Propriétés

Les bases de données JPA embarquées sont créées automatiquement.

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

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

Ou utiliser les propriétés natives d'Hibernate

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

Ex :

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



Comportement transactionnel des Repository

Par défaut, les méthodes CRUD sont transactionnelles.

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

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



@Transactional et *@Service*

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

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



Example

@Service

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

@Transactional

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



Configuration des Templates

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

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

Ex :

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




Example

@Repository

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

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

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

@Autowired

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

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

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

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

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

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

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

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

```
    return user;
```

```
}
```

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

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

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

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

```
}
```

```
}
```



Code JDBC ou JPA

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

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



OpenInView

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

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



APIs Rest avec SpringBoot

Spring MVC et les APIs REST

Principes RESTFul

Dé/Sérialisation avec Jackson

Exceptions, CORS et OpenAPI



Introduction

SpringBoot est adapté pour le développement web

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

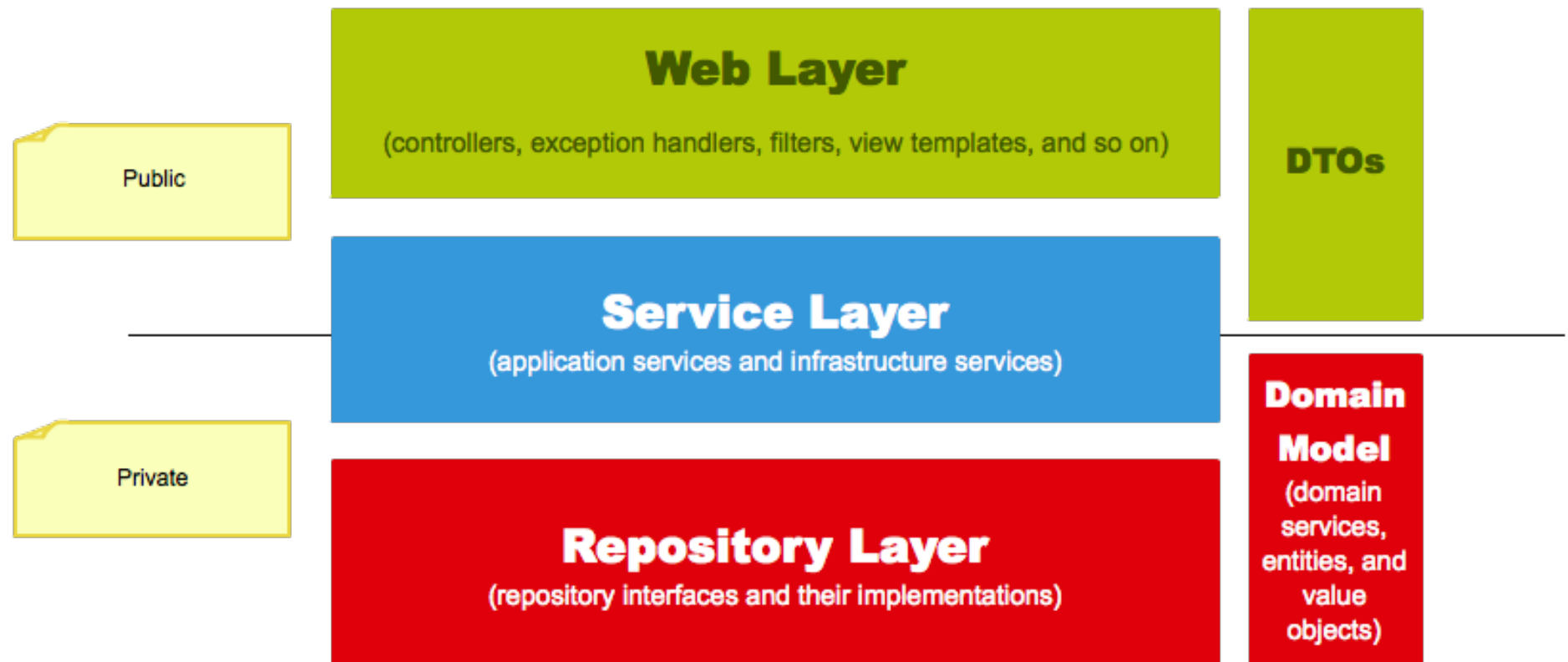
Spring MVC permet de déclarer des beans de type

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

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



Architecture classique projet





@RestController

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

@RestController

@RequestMapping("/api")

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



Annotation *@RequestMapping*

@RequestMapping

- Placée au niveau de la classe, elle indique que toutes les méthodes du contrôleur seront relatives à ce chemin
- Au niveau d'une méthode, l'annotation précise les attributs suivants :
 - **path** : Valeur fixe ou gabarit d'URI
 - **method** : Pour limiter la méthode à une action HTTP
 - **produce/consume** : Préciser le format des données d'entrée/sortie
Dans le cadre d'une API Rest il n'est pas nécessaire de préciser ces attributs car le format est toujours JSON



Variantes *@RequestMapping*

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

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



Types des arguments de méthode

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

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

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



Annotations sur les arguments de méthode

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

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



Gabarits d'URI

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

`http://www.example.com/users/{userId}`

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

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



Paramètres HTTP avec *@RequestParam*

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

    // ...

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

        return pet;
    }

    // ...

}
```

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



@RequestBody et convertisseur

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

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



Exemple *@RequestBody*

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

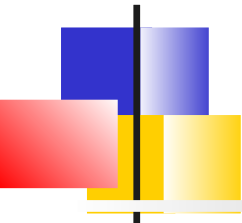
    // ...

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

        return pet;
    }

    // ...

}
```



Types des valeurs de retours des méthodes

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

- Une classe **Modèle ou DTO** qui sera converti en JSON via la librairie Jackson.
Le code retour par défaut est alors 200
- Un objet ***ResponseEntity<T>*** permettant de positionner les codes retour et les entêtes HTTP voulues
- Il est également possible d'annoter la méthode avec ***@ResponseStatus*** pour indiquer le code de retour HTTP



Exemples

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

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

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

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

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public User register(@Valid @RequestBody User user) {

        return userRepository.save(user);
    }
}
```



APIs Rest avec SpringBoot

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



Auto-description

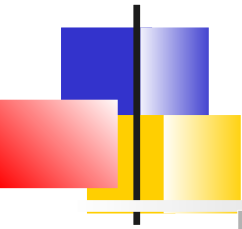
Une API bien conçue est compréhensible par un développeur sans qu'il ait à lire la documentation

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

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

GET https://api.uber.com/<version>/partners/125/payments

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



Principe de Base

Une ressource \Leftrightarrow Une entité

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



Règles de nommage

Pour les objets liés à l'entité principale. Il est recommandé d'utiliser une structure hiérarchique : /objets/{objet_id}/sous_objets

GET /calendar/meetings/{meeting_id}/meeting_room

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

GET /calendar/meetings/{meeting_id}/attendees/search

Paramètres de requête

I

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



Codes retour

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

Ils sont standards

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



Succès

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

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



Erreurs client

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

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



Erreurs serveur

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

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



APIs Rest avec SpringBoot

Spring MVC et les APIs REST

Principes RESTFul

Dé/Sérialisation avec Jackson

Exceptions, CORS et OpenAPI



Sérialisation JSON

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

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

Mais, généralement le développeur doit régler certaines problématiques :

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



Comportement par défaut

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

Devient :

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



Concepts Jackson

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

// Sérialisation

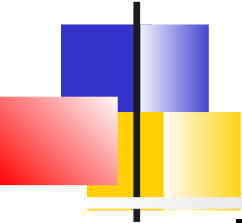
```
Member m = memberRepository.findById(4l) ;  
ObjectMapper objectMapper = new ObjectMapper() ;  
String jsonString = objectMapper.writeValueAsString(m) ;
```

...

// Désérialisation

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

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



Solutions aux problématiques de sérialisation

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

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



Exemple DTO

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

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

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

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

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




Format de Dates

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

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



Relations bidirectionnelles

Le problème

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

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



Relations bidirectionnelles

Une solution

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

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

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

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



Relations bidirectionnelles

Une autre solution

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

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

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

Sérialisation d'un Item :

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



Relations bidirectionnelles

Une autre solution

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

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

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



@JsonView

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

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

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

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



@JsonView

Annotations sur la classe du modèle

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



Activation d'une vue

```
@RestController
public class StaffController {

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

    ...

    ObjectMapper mapper = new ObjectMapper();

    Staff staff = createStaff();

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




Autres annotations Jackson

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

@JsonRootName : Arbre JSON

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

....



Sérialiseur spécifique

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

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

@JsonComponent

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



APIs Rest avec SpringBoot

Spring MVC et les APIs REST

Principes RESTFul

Dé/Sérialisation avec Jackson

Exceptions, CORS et *OpenAPI*



Personnalisation de la configuration Spring MVC

- Le personnalisation de la configuration par défaut de SpringBoot peut être effectuée en définissant un bean de type ***WebMvcConfigurer*** et en surchargeant les méthodes proposée.
- Dans le cadre d'une API Rest, une méthode permet de configurer le CORS¹

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



Exemple Cross-origin

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

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

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



Gestion des erreurs

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

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

Pour remplacer le comportement par défaut :

- Modèle 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*, *@RestControllerAdvice*

@RestControllerAdvice

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

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

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

@Override

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



SpringDoc

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

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

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

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>

  <!-- OU : springdoc-openapi-starter-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 la propriété :
`springdoc.api-docs.enabled=false`



Interactions entre services

RestClient
Messaging



Appels de service REST

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

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



Exemple

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

    public MyBean(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplate = restTemplateBuilder.rootUri("
            http://localhost:8080/api")
            .basicAuthentication("user", "password")
            .build();
    }
    public Details someRestCall(String name) {
        return this.restTemplate.getForObject("/{name}/details",
            Details.class,
            name);
    }
}
```



WebClient

WebClient est la nouvelle interface apportée par Spring Webflux permettant d'effectuer les requêtes Web.

La solution offre du support pour les interactions synchrones et asynchrones, elles peuvent donc être utilisées sur les 2 stacks web (servlet et reactive)

L'interface a une unique implémentation :
DefaultWebClient



Création d'un WebClient

3 alternatives pour créer un WebClient

// Config par défaut

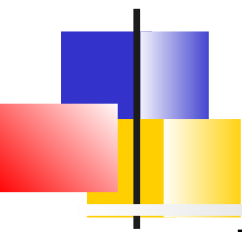
```
WebClient client = WebClient.create();
```

// Base Uri

```
WebClient client = WebClient.create("http://localhost:8080");
```

// Builder

```
WebClient client = WebClient.builder()  
    .baseUrl("http://localhost:8080")  
    .defaultCookie("cookieKey", "cookieValue")  
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)  
    .defaultUriVariables(Collections.singletonMap("url",  
        "http://localhost:8080"))  
    .build();
```

Préparation de la requête

La préparation de la requête consiste à préciser la méthode HTTP, l'URL, le corps et les entêtes.

```
requestSpec = client.post()  
    .uri("/resource")  
    .bodyValue("data")  
    .header(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)  
    .accept(MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML)
```



Récupération de la réponse

Pour récupérer la réponse on peut utiliser ***exchangeToMono*** et ***exchangeToFlux*** qui permettent d'inspecter la réponse (entête, code status,)

Ou tout simplement ***retrieve*** qui permet de récupérer le corps de la réponse



Examples

// exchangeToMono

```
Mono<String> response = requestSpec.exchangeToMono(response -> {  
    if (response.statusCode().equals(HttpStatus.OK)) {  
        return response.bodyToMono(String.class);  
    } else if (response.statusCode().is4xxClientError()) {  
        return Mono.just("Error response");  
    } else {  
        return response.createException()  
            .flatMap(Mono::error);  
    }  
});
```

// Retrieve simple

```
Mono<String> response = headersSpec.retrieve()  
    .bodyToMono(String.class);
```



Interactions entre services

RestClient
Messaging



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

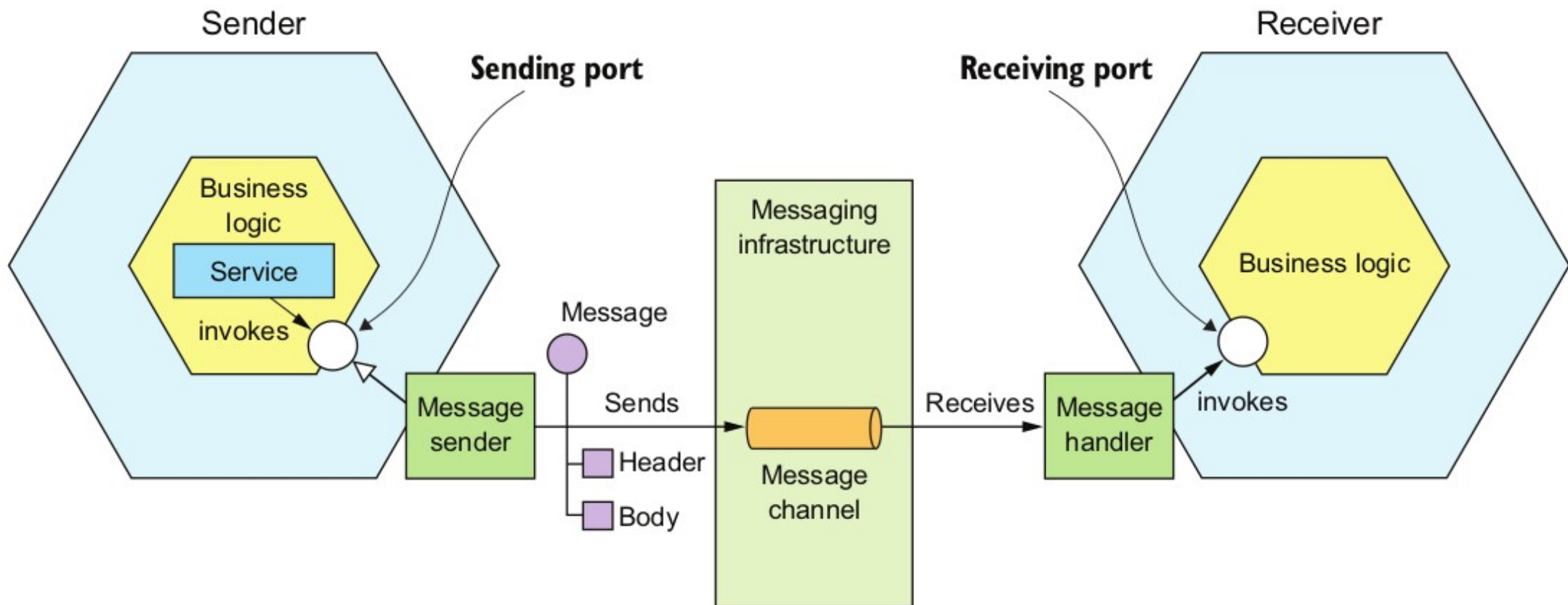
Des difficultés :

- 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>

Architecture





Sémantique des messages

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



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 ORDER_STATUS_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) {
    ...
}
```



Spring Security

Principes

Modèles stateful/stateless
Auto-configuration Spring Boot
OpenIdConnect et 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, OpenIDConnect, Kerberos, PAM, CAS)
- permettant la configuration des contraintes d'accès aux URLs et aux méthodes des services métier



Principes et mécanisme

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

- Le bean **springSecurityFilterChain** encapsule une chaîne de filtres interceptant toutes les requêtes HTTP. Chaque filtre est responsable d'un aspect de la sécurité. La chaîne de filtre est hautement configurable et s'adapte à toutes les approches

Si en plus on désire ajouter de la sécurité au niveau des méthodes, il faut explicitement l'activer (même dans un contexte SpringBoot avec **@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 de la configuration consiste :

- A personnaliser le filtre *springSecurityFilterChain* en créant un bean de type **SecurityFilterChain**
la classe **HttpSecurity** est un builder facilitant sa création
- À personnaliser l'authentification :
 - en créant un bean de type **AuthenticationManager**
La classe *AuthenticationManagerBuilder* facilite la création de Realm (inMemory, jdbc, ldap, ...)
 - ou complètement personnalisé par l'implémentation d'un bean **UserDetailsService**
- A ignorer la sécurité pour certaines ressources en définissant un Bean de type **WebSecurityCustomizer** :
lambda prenant un objet **WebSecurity** en argument



Exemple SecurityFilterChain

// Ex : Spring MVC

@Bean

```
public SecurityFilterChain filterChain(HttpSecurity http) throws  
    Exception {  
    http  
        .authorizeRequests(authorize ->  
            authorize.anyRequest().authenticated())  
        .formLogin(withDefaults())  
        .httpBasic(withDefaults());  
    return http.build();  
}
```



Exemple : SecurityWebFilterChain

// Ex : Spring WebFlux

@Bean

```
public SecurityWebFilterChain securityWebFilterChain(  
    ServerHttpSecurity http) {  
    return http.authorizeExchange()  
        .pathMatchers("/actuator/**").permitAll()  
        .pathMatchers("/auth/**").permitAll()  
        .anyExchange().authenticated()  
        .and()  
        .oauth2Login().csrf().disable().build();  
}
```



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 SecurityConfiguration {
```

```
    // Implémentation UserDetailsService
```

```
    @Bean
```

```
    public InMemoryUserDetailsManager userDetailsService() {
```

```
        UserDetails user = User.withDefaultPasswordEncoder()
```

```
            .username("user")
```

```
            .password("password")
```

```
            .roles("USER")
```

```
            .build();
```

```
        return new InMemoryUserDetailsManager(user);
```

```
    }
```

```
}
```



Personnalisation via *UserDetailsService*

L'implémentation de ***UserDetailsService*** peut être complètement personnalisée

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.



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

Auto-configuration Spring Boot

OpenIdConnect et 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



Authentication stateful

Appli web standard ou monolithique Rest

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 authentication basique du navigateur .
3. Le navigateur renvoie une réponse au serveur :
 1. Soit le POST de la page de login
 2. Soit les entêtes HTTP d'authentification.
4. Le serveur décide si les crédits sont valides :
 1. si oui. L'authentification est stockée dans la session, la requête originelle est réessayée, si les droits sont suffisants la page est retournée sinon un code 403
 2. Si non, le serveur redemande une authentication.
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()`

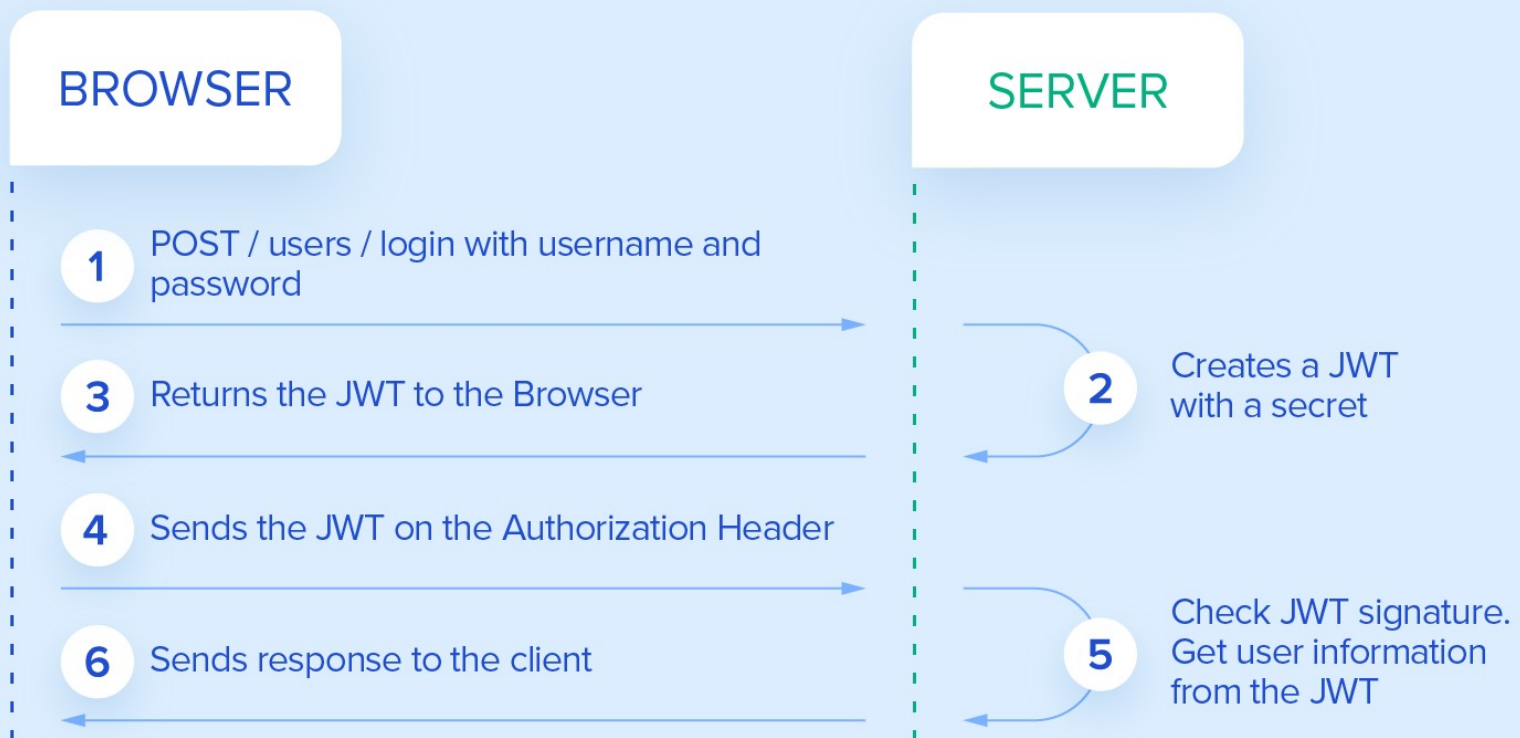


Authentication stateless

API REST - microservice

1. Le client demande une ressource protégée.
2. Le serveur renvoie une réponse indiquant que l'on doit s'authentifier en envoyant une réponse 401.
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

OpenIdConnect et 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

OpenIdConnect et OAuth2



Rôles du protocole

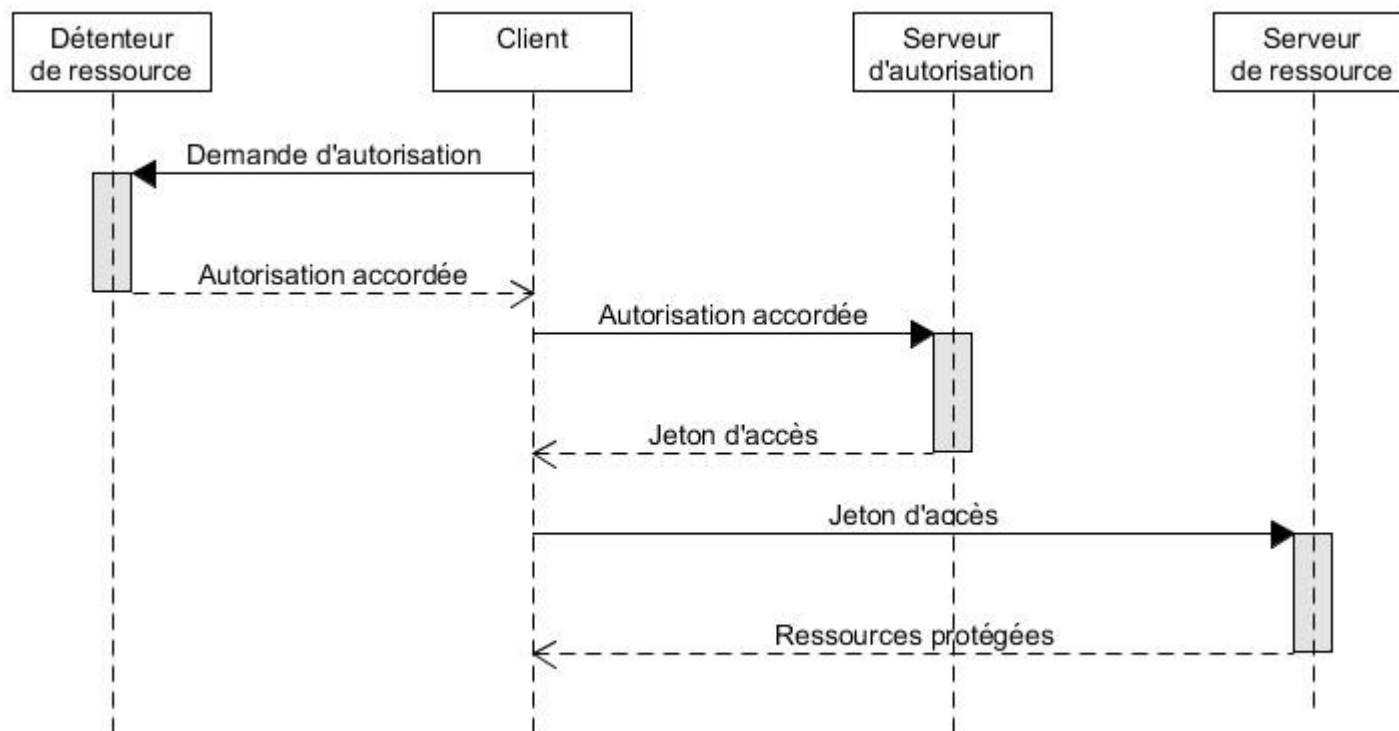
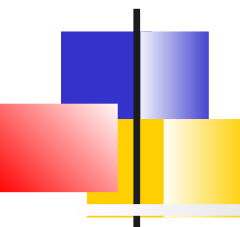
Le **Client** est l'application qui essaie d'accéder à des ressources détenues par l'utilisateur. Elle a besoin d'obtenir une permission de l'utilisateur pour le faire.

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



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



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



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 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.

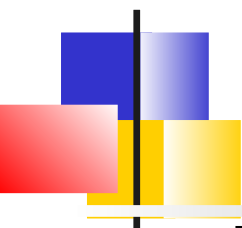
Voir :

<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 fourni :

- **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



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();
    }
}
```

Voir : <https://github.com/Baeldung/spring-security-oauth.git>



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

SpringBoot 3, Spring 6, JUnit5

Première version Fin 2022



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

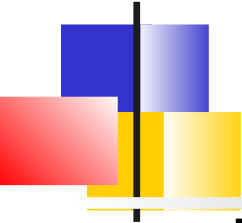
```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes = TestConfig.class)
class SimpleTests {

    @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*

```
@SpringBootTest
public class MyTests {

    @MockBean
    private RemoteService remoteService;

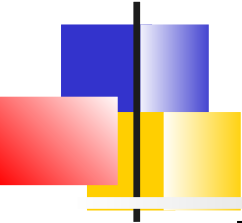
    @Autowired
    private Reverser reverser;

    @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*

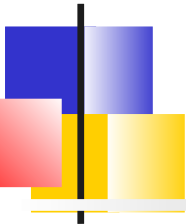
Un *TestEntityManager* peut être injecté ainsi qu'un *JdbcTemplate*



Example

@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



Annexes

Actuator
Déploiement
MongoDB
Spring MVC
Spring Reactive



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



Annexes

Actuator
Déploiement
MongoDB
Spring MVC
Spring Reactive



Introduction

Plusieurs alternatives pour déployer une application Spring-boot :

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



Application stand-alone

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

```
mvn package
```

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

Pour l'exécuter :

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



Fichier Manifest

Manifest-Version: 1.0

Implementation-Title: documentService

Implementation-Version: 0.0.1-SNAPSHOT

Archiver-Version: Plexus Archiver

Built-By: dthibau

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

Implementation-Vendor-Id: org.formation.microservice

Spring-Boot-Version: 1.3.5.RELEASE

Created-By: Apache Maven 3.3.9

Build-Jdk: 1.8.0_121

Implementation-Vendor: Pivotal Software, Inc.

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



Création de war

Pour créer un war, il est nécessaire de :

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

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



Exemple

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



Création de service Linux

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

=> target/artifactId.jar is executable !

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



Cloud

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

La documentation de référence offre du support pour :

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



Exemple CloudFoundry/Heroku

Cloud Foundry

```
cf login
```

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

Heroku

Mise à jour d'un fichier Procfile :

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

```
git push heroku master
```



Annexes

Actuator
Déploiement
MongoDB
Spring MVC
Spring Reactive



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



Annexes

Actuator
Déploiement
MongoDB
Spring MVC
Spring Reactive



Spring MVC

Rappels Spring MVC

Support pour les APIs Rest
Spring Boot pour les APIs Rest



Introduction

SpringBoot est adapté pour le développement web

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

Spring MVC permet de déclarer des beans de type

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



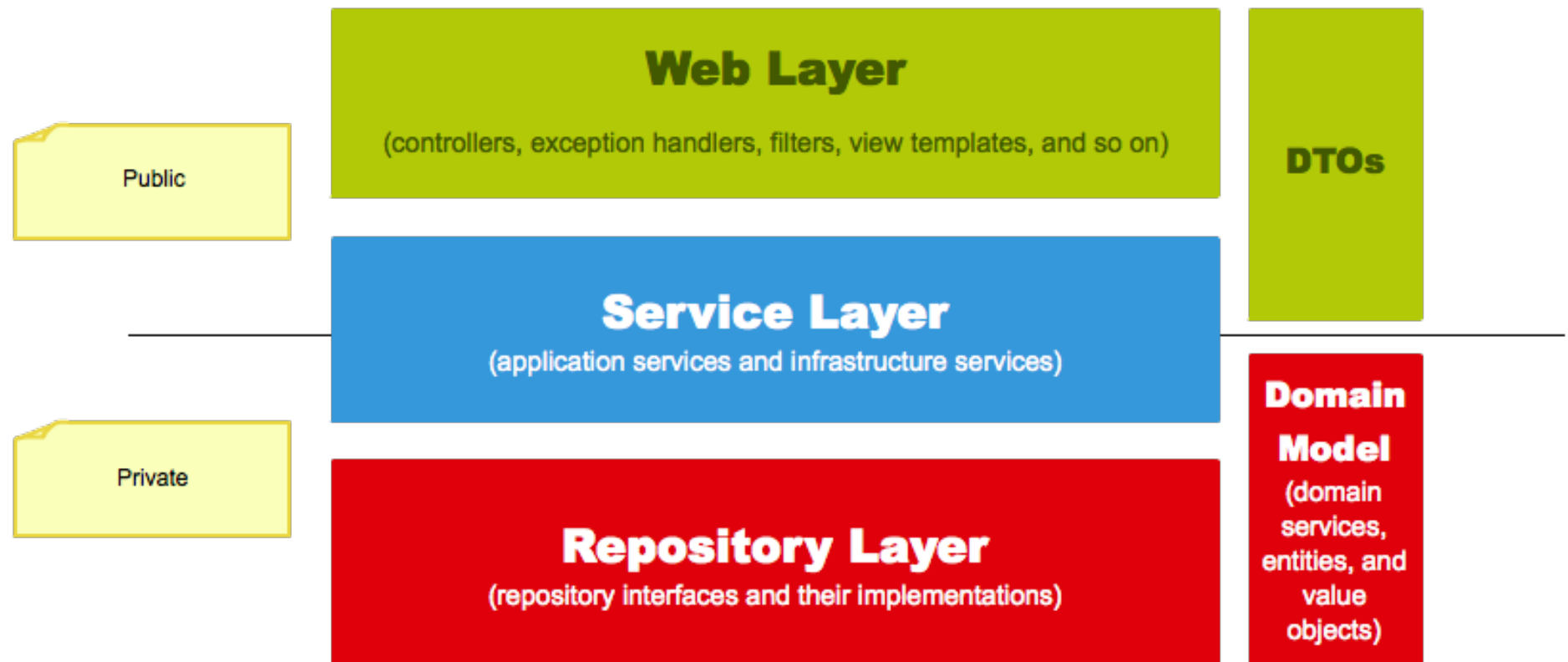
Rappels Spring MVC

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

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



Architecture classique projet





@Controller, @RestController

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

@Controller

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



@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
- **@RequestAttribute** : 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*

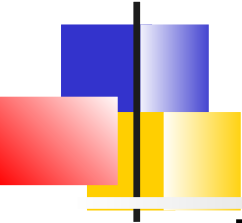
```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @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
- ...



Spring Boot et Spring MVC



Auto-configuration

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

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



Personnalisation de la configuration

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



Exemple : Définition de *ViewController*

```
@Configuration
```

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

```
    @Override
```

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

```
}
```



Exemple *Intercepteurs*

```
@SpringBootApplication
public class MyApplication {

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

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

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



Convertisseur de messages HTTP

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

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

```
@Configuration
public class MyConfiguration {

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

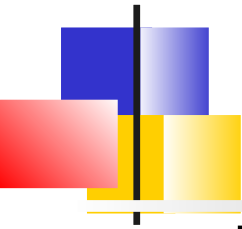


Contenu statique

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

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

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



Webjar

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

- Les webjars permettent une gestion des dépendances par Maven,

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

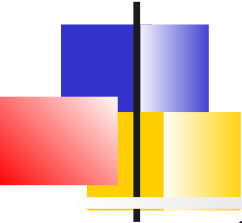


Exemple

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

...

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



Technologies de vue et templating

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

Spring Boot permet l'auto-configuration de

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

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



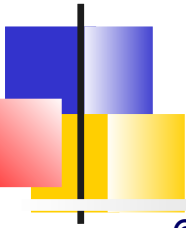
Gestion des erreurs

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

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

Pour remplacer le comportement par défaut :

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



Exemple *@ControllerAdvice*

@ControllerAdvice

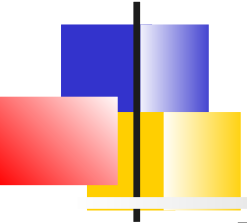
```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {
```

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

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

@Override

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



Page d'erreur

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

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



Pages d'erreur personnalisées

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

src/

+ - main/

+ - java/

| + <source code>

+ - resources/

+ - public/

+ - error/

| + - 404.html

| + - 5xx.ftl



Mécanisme auto-configuration

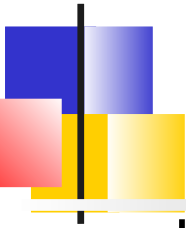


Auto-configuration

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

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

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



Annotations conditionnelles

Les conditions peuvent se baser 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 {

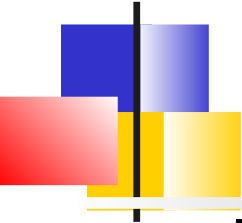
    private final SolrProperties properties;

    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 *SolR* tire

- Les classes de Configuration conditionnelles
- Les librairies SolR

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

```
@Component
public class MyBean {

    private SolrClient solrClient;

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



Format *.yaml*



Format YAML

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

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



Exemple *.yml*

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

Fournit les clés suivantes :

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



Listes

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

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

Devient :

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




Reactive Spring

Programmation réactive

Spring Reactor

Spring Data Reactive

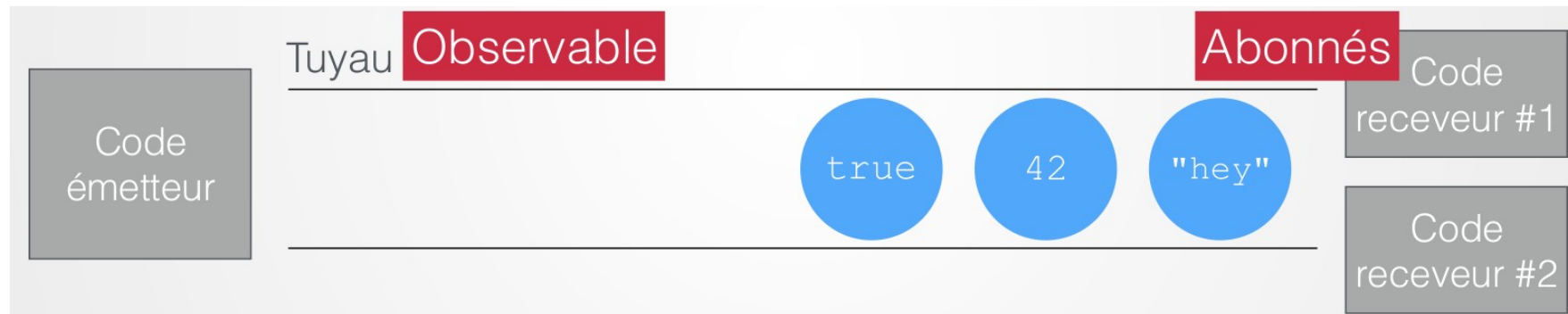
Spring Webflux

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.

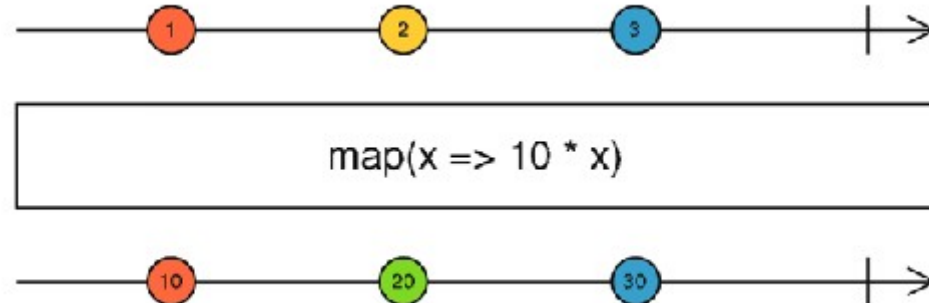




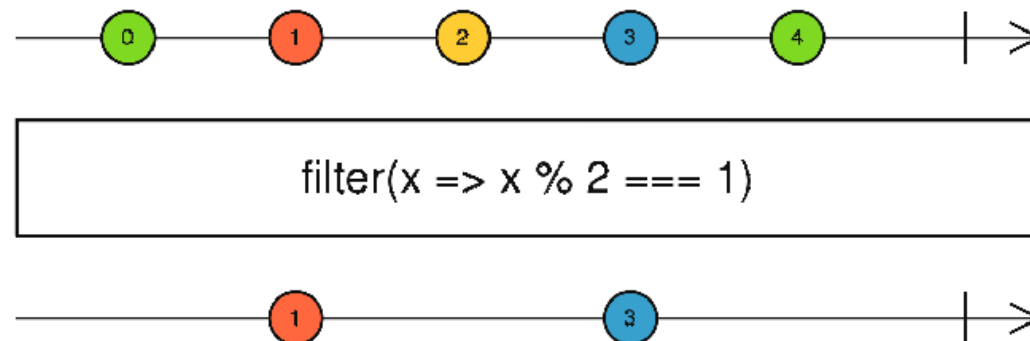
Marble diagrams

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



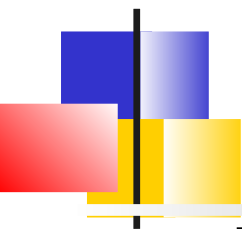
Interfaces Reactive Streams

```
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> {  
}
```

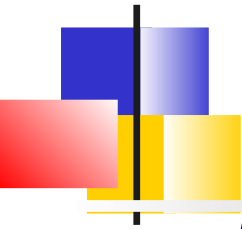


Back pressure

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.

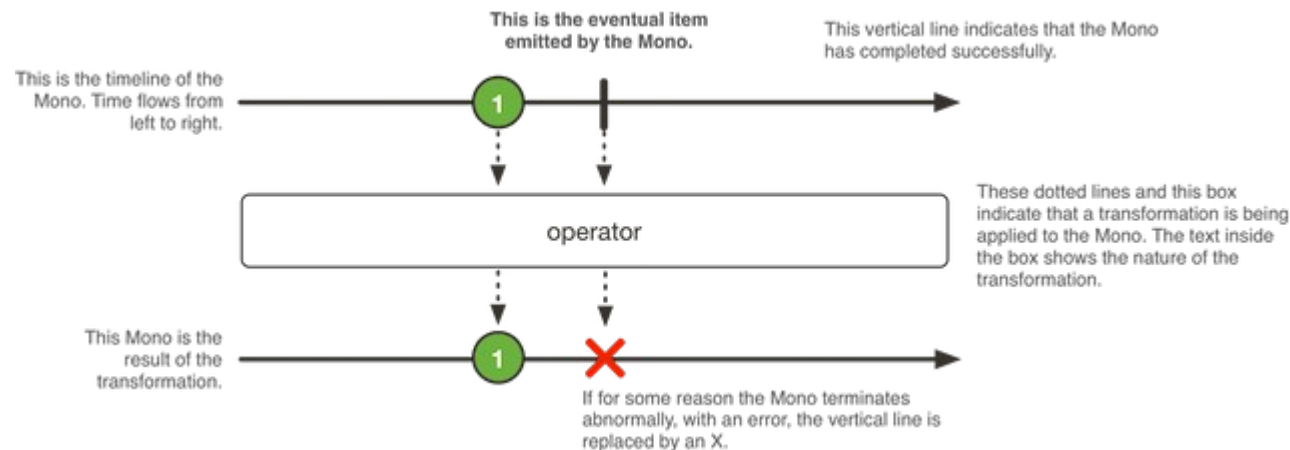
Les événements sont traduits par des appels de méthode sur les abonnés :

- 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 :

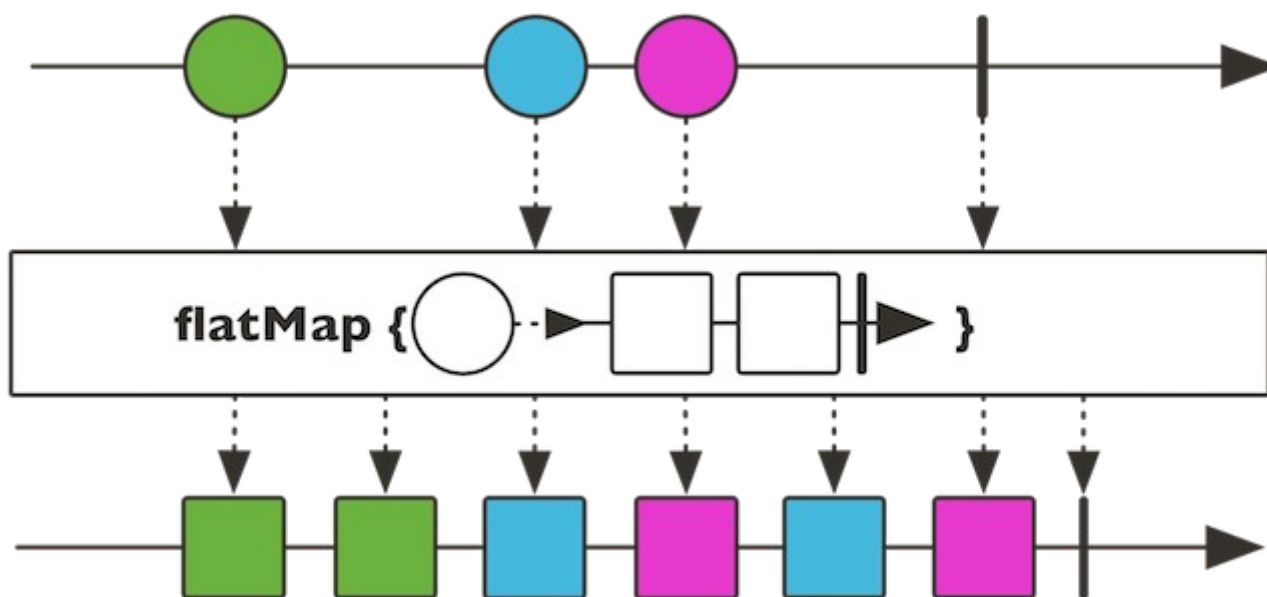
all, *any*, *hasElements*, *hasElement*

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



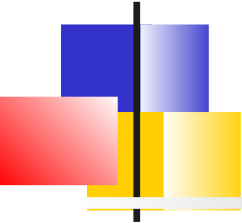
Accès réactifs aux données persistante

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

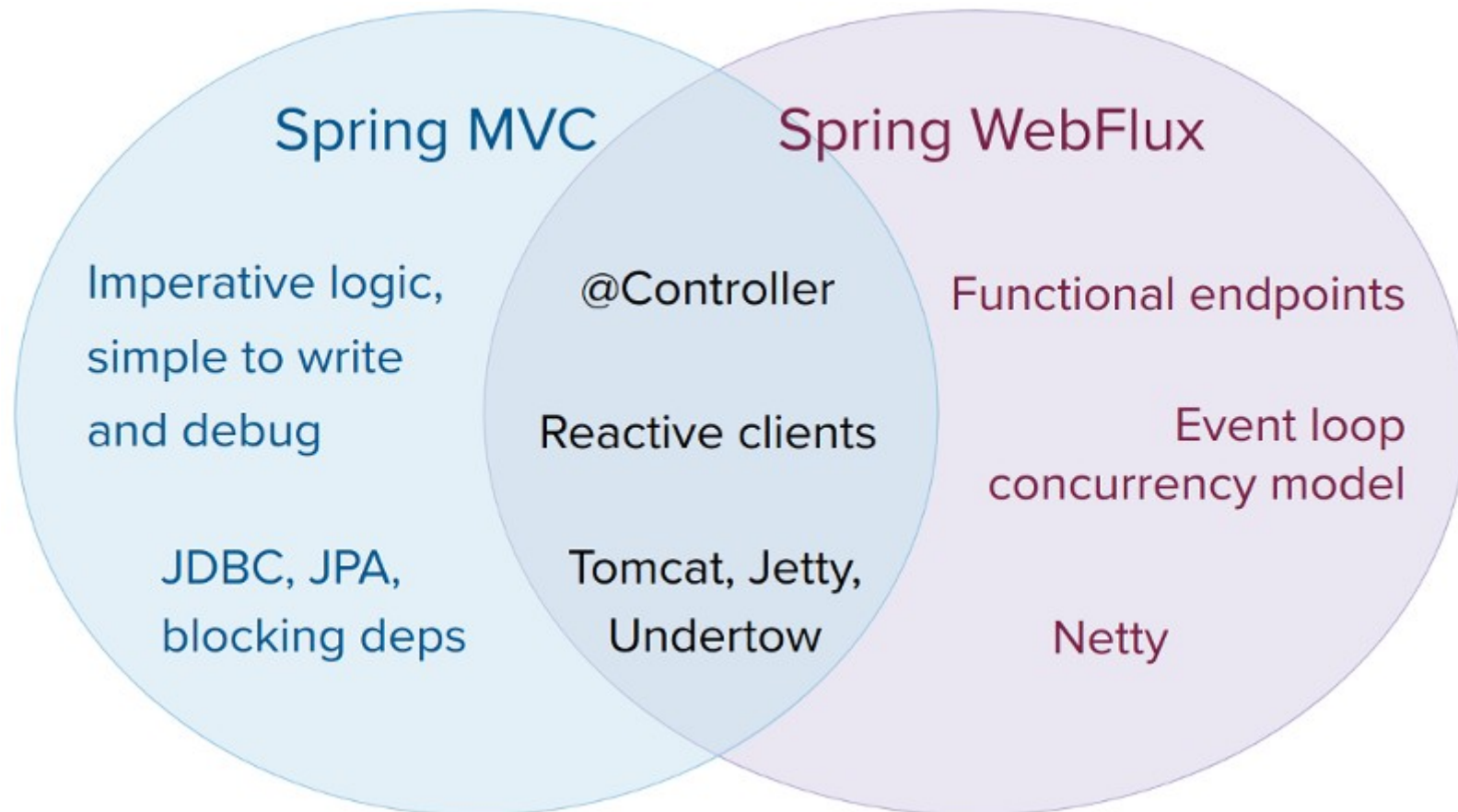
Le module *spring-web* est la base pour Spring Webflux.

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



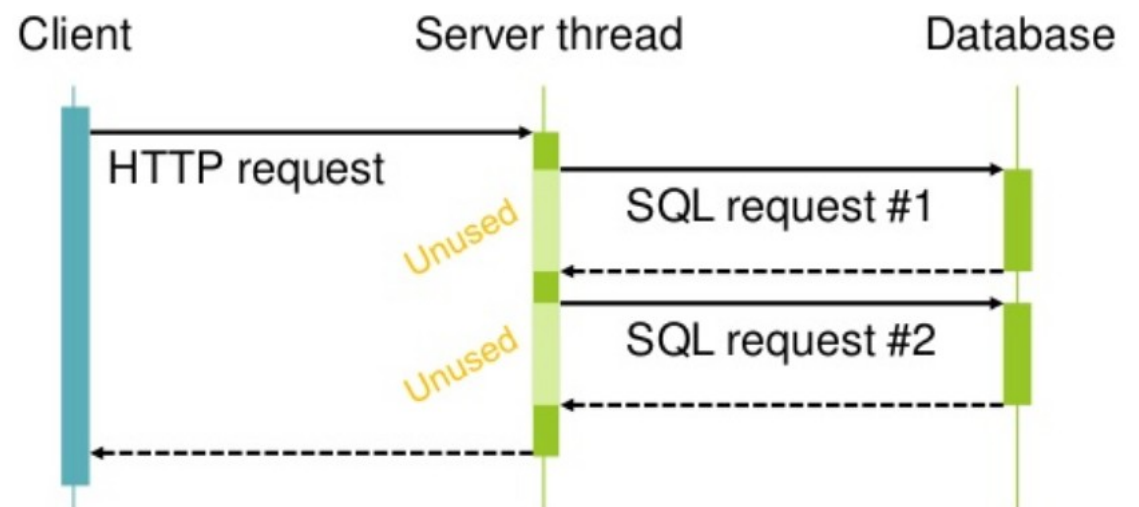
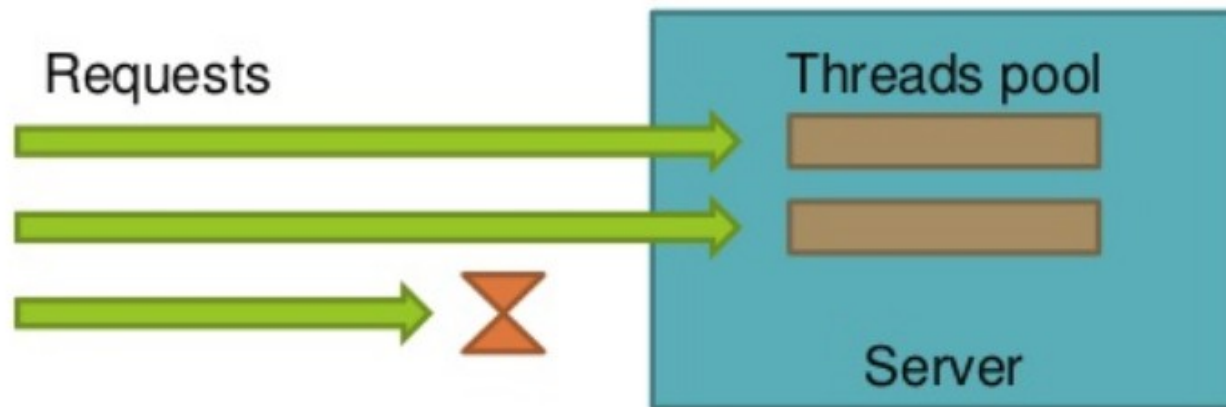
Performance et Scaling

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)

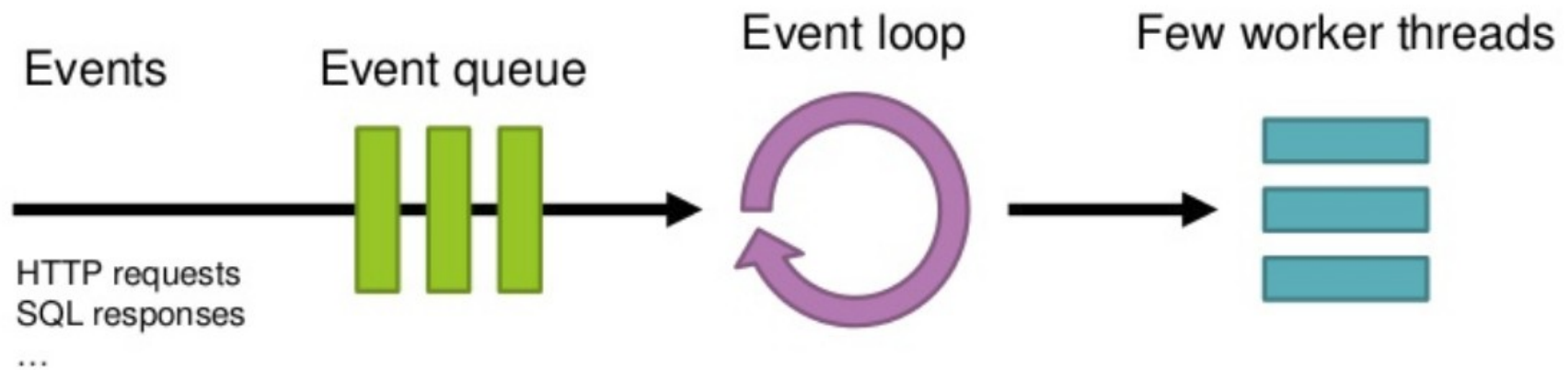
Le bénéfice attendu est la possibilité de **scaler** avec un petit nombre de threads fixes et moins de mémoire. Cela rend les applications plus résistantes à la charge.

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*.



Exemple

```
@RestController
public class PersonController {
    private final PersonRepository repository;
    public PersonController(PersonRepository repository) {
        this.repository = repository;
    }
    @PostMapping("/person")
    Mono<Person> create(@RequestBody Person person) {
        return this.repository.save(person);
    }
    @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*

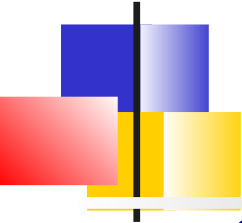


Endpoints fonctionnels

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 :

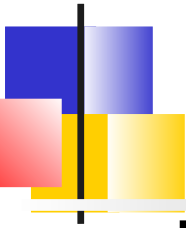
```
HandlerFunction<ServerResponse> helloWorld =  
    request ->  
    ServerResponse.ok().body(fromObject("Hello World"));
```

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();
        }
    });
```