

Rappels Spring, La stack web

David THIBAU - 2021

david.thibau@gmail.com



Agenda

- **Introduction**

- Rappels Spring
- L'accélérateur Spring Boot
- Les projets Spring

- **La stack web**

- Spring Data
- Spring MVC
- Spring WebFlux
- Spring Security
- Spring Test



Introduction

Rappels Spring

L'accélérateur Spring Boot

Les projets Spring



IoC et *ApplicationContext*

Spring est un conteneur de beans qui applique le pattern **IoC**

- A partir d'une configuration, Spring instancie des objets et injecte leurs dépendances.
- La plupart des objets sont des singletons
- L'ensemble des beans est accessible via un ***ApplicationContext***



Avantages de l'injection de dépendances

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



Configuration XML (legacy)

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



Configuration via annotations

```
@Import(DataSourceConfig.class)
```

```
@Configuration
```

```
public class SimpleConfiguration {
```

```
@Autowired
```

```
Connection connection;
```

```
@Bean
```

```
Database getDatabaseConnection(){  
    return connection.getDBConnection();
```

```
}
```

```
// More code here....
```

```
}
```



Configuration via annotations

@RestController

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

@Autowired

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

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




Injection implicite

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

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

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



Cycles de vie des objets gérés

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

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



Introduction

Rappels Spring
L'accélérateur Spring Boot
Les projets Spring



Introduction

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

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

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



Auto-configuration

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

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

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

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



Boot

Au démarrage de l'application, les packages d'auto-configuration évaluent un certains nombres de conditions qui si elles sont vérifiées provoquent l'instanciation de beans

- Par exemple : J'ai SolR dans le classpath, j'instancie alors un bean SolRClient avec la configuration par défaut

Un rapport du résultat de l'auto-configuration peut être obtenu en démarrant l'application avec l'option **-debug**



Conditions

Les conditions d'activation de la configuration :

- La présence ou l'absence d'une classe
- La présence ou l'absence d'un bean
- Une propriété
- La présence d'une ressource
- Le fait que l'application est une application Web ou pas
- Une expression SpEL



Gestion des dépendances

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

- Il organise les fonctionnalités de Spring en modules.
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des "**starter modules**".
- "**Spring Initializr**", qui peut être utilisée via un navigateur ou un IDE, permet de générer des configurations Maven ou Gradle à partir d'un assistant

=> Le projet ne gère alors plus qu'un seul n° de version , celui de SpringBoot



Exemple Maven (1)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.4</version>
  </parent>
  <groupId>org.formation</groupId> <artifactId>demo</artifactId> <version>0.0.1-SNAPSHOT</version>

  <properties>
    <java.version>11</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```



Exemple Maven (2)

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

Cibles :

Packaging fat-jar : *./mvnw package*

Image docker : *./mvnw spring-boot:build-image*

Infos de build : *./mvnw spring-boot:build-info*

Démarrage de l'appli : *./mvnw spring-boot:bootRun*



Principaux Starter Modules

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

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

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

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

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



Projet Java

```
package com.infoq.springboot;

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

@RestController
@SpringBootApplication
public class Application {

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

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



@SpringBootApplication

@SpringBootApplication regroupe les 3 annotations suivantes :

- **@EnableAutoConfiguration** : Autoriser le mécanisme d'auto-configuration
- **@ComponentScan** : Marqueur racine pour la recherche des beans applicatifs
- **@Configuration** : Possibilité de fournir des méthodes de création de Bean



Structure typique

com

+ - example

+ - myproject

+ - Application.java

+ - OneConfig.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java



Personnalisation de la configuration par défaut

Du point de vue du développeur, la configuration par défaut peut être surchargée par différents moyens

- Des **fichiers de configuration externe** (*.properties* ou *.yml*) permettent de fixer des valeurs des variables de configuration. Également notion de profils
- Des classes utilitaires Spring ***Configurer** ou ***Customizer** permettant de surcharger la configuration par défaut via l'API
- Utiliser des **classes spécifiques** au bean que l'on veut surcharger (exemple *AuthenticationManagerBuilder*)
- La **définition de Beans** remplaçant les beans par défaut
- La **désactivation** de l'auto-configuration



Adaptation de la configuration à un environnement

Du point de vue de l'exploitant, plusieurs moyens pour surcharger les valeurs de configuration :

1. La **ligne de commande**.

Ex : `--server.port=9000`

2. **Environnement JVM**, REST, Servlet, JNDI

3. **Variables d'environnement** de l'OS

4. Activation d'un **profil spécifique**.

Ex : `--spring.profiles.active=prod`

5. Fourniture d'un **fichier complet**

Ex

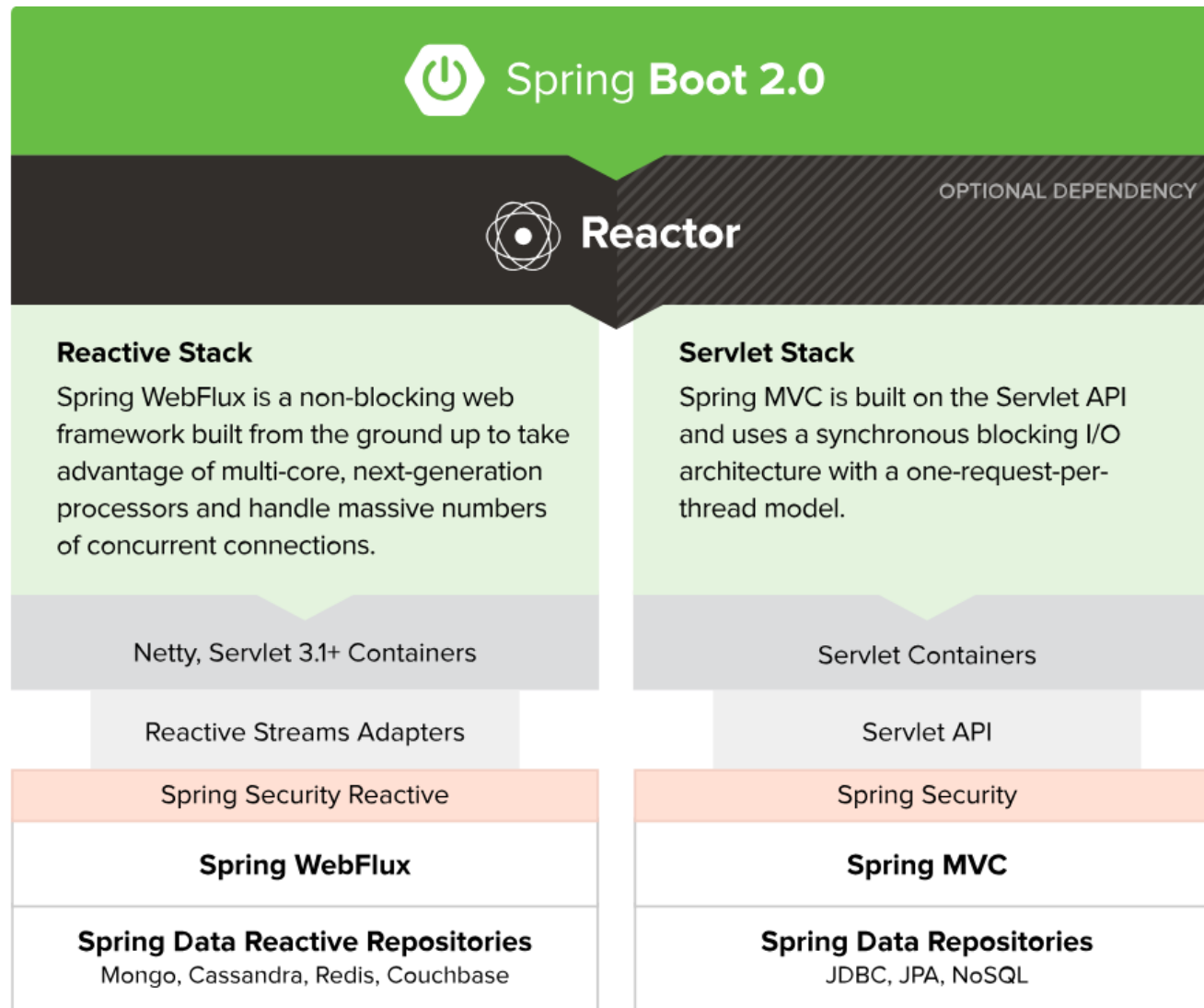
`--spring.config.location=classpath:/default.properties,
file:./override.properties`



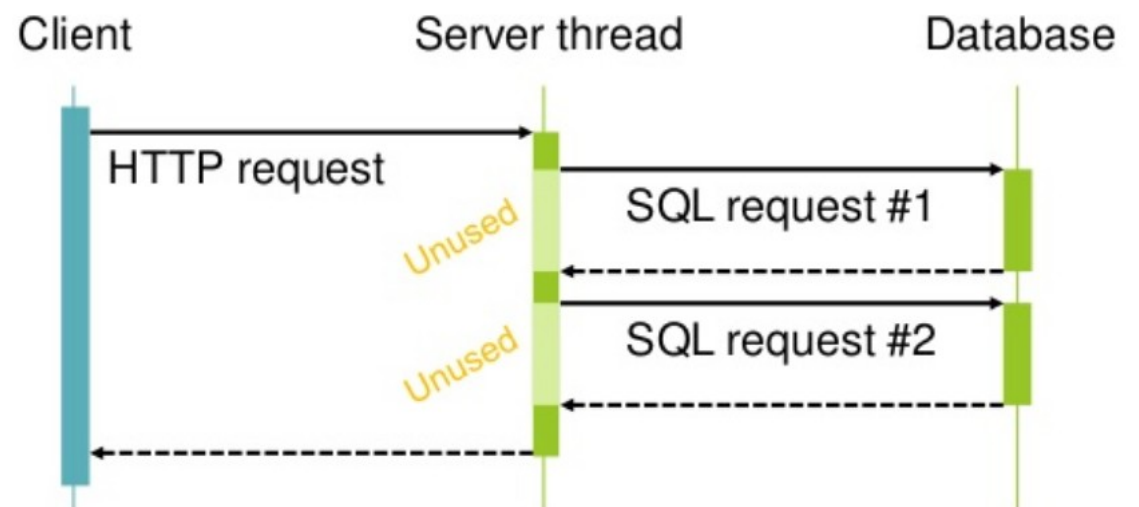
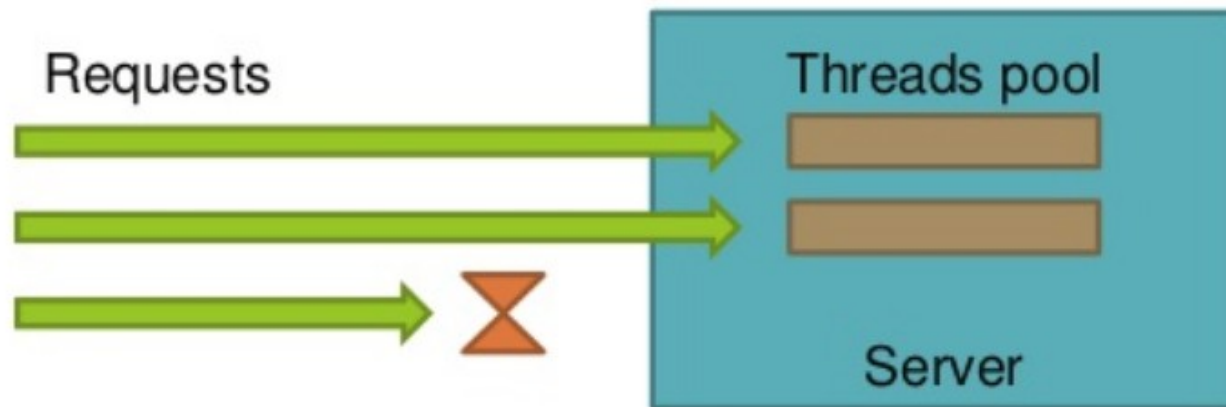
Introduction

Rappels Spring
L'accélérateur Spring Boot
Les projets Spring

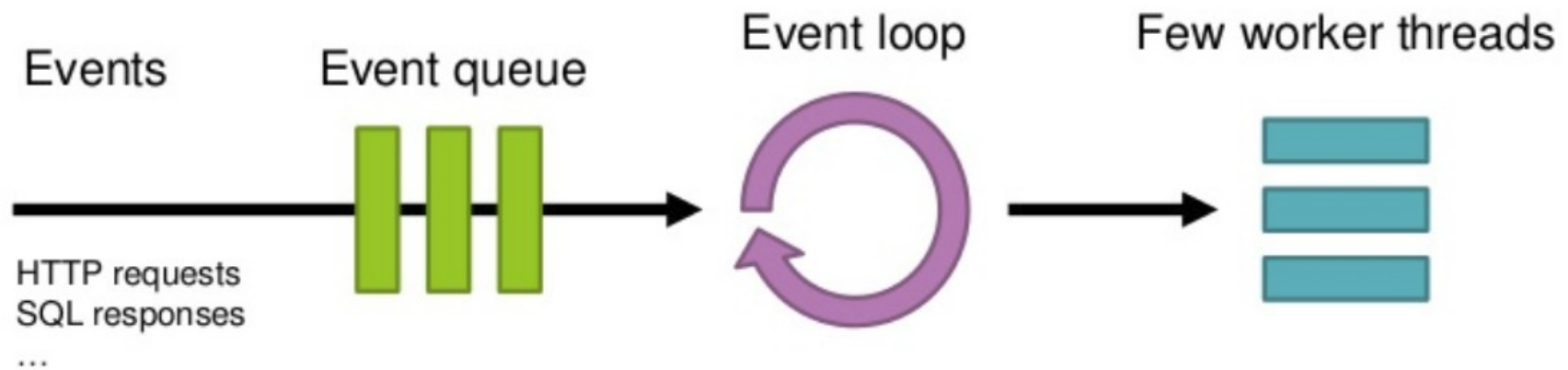
Spring 5.x et SB 2.0



Modèle bloquant



Modèle non bloquant





Projets d'intégration

Spring Batch : Applications batch robustes, performantes et modulaires

Spring Integration : Application des patterns EAI,
– Adapteur pour AMQP, JMS, Kafka, MongoDB, SysLog,
...
– Transformateur de données

Spring AMQP, Apache Kafka : IoC pour AMQP et Kafka

Spring Cloud Stream : Intégration appliqué aux micro-services



Projets Web

Spring Session : API et implémentations pour gérer les informations d'un utilisateur. (Supporte le clustering)

Spring HATEOAS : APIs pour créer des représentations REST suivant le principe HATEOAS (Découverte de l'API via des liens hypertextes)

Spring Data Rest : Crée automatiquement une API Rest à partir des Repository de Spring Data

Spring REST Docs : Documentation API Rest via *OpenAPI* 3.0

Spring GraphQL support pour les services web utilisant GraphQL Java

Spring Web Service : Support pour les services SOAP



Spring Cloud

Spring Cloud fournit des outils pour implémenter rapidement les patterns des systèmes distribués

Ex : Service de discovery, gateway, Gestion centralisée de configuration, élection de leader, circuit-breaker pattern

Il s'attache aussi à fournir du support pour différents types d'infrastructures Cloud :
Cloud Foundry, Azure, Amazon, Alibaba, Kubernetes

Il s'applique naturellement aux architectures micro-services



Projets Spring

SpringData

Spring MVC

Spring Web Flux

Spring Security

Spring Test



Introduction

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

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



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*, ...



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 JPA

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

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

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



Méthodes de sélection de données

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

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

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

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

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



Gestion des paramètres

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

Les valeurs de retours peuvent alors être :

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

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



Apports Spring Boot pour JPA

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***



Support pour une base embarquée

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

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

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




Base de production

Les bases de production peuvent également être auto-configurées.

Les propriétés requises à configurer sont :

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

`spring.datasource.username=dbuser`

`spring.datasource.password=dbpass`

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

Voir *DataSourceProperties* pour l'ensemble des propriétés disponibles

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



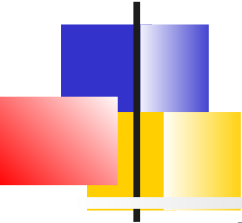
Configuration du pool

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

Par exemple pour Hikari :

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

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



Propriété *ddl-auto*

Les bases de données JPA embarquées sont créées automatiquement à chaque démarrage.

Pour les autres, il faut préciser la propriété

spring.jpa.hibernate.ddl-auto

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

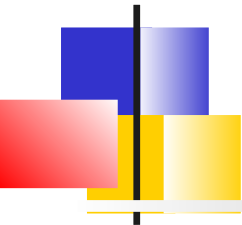


Comportement transactionnel des Repository

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);  
        }  
    }  
}
```

@Transactional



NoSQL

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") ;
        ...
    }
}
```



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 d'écoute est déterminé

aléatoirement ou fixé par la propriété :

`spring.data.mongodb.port`

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



Reactive

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)



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>
```



Projets Spring

Spring Data
Spring MVC
Spring Web Flux
Spring Security
Spring Test



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

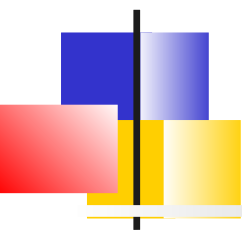
- ***@Controller*** ou ***@RestController***
- Dont les méthodes peuvent être associées à des requêtes http via ***@RequestMapping***



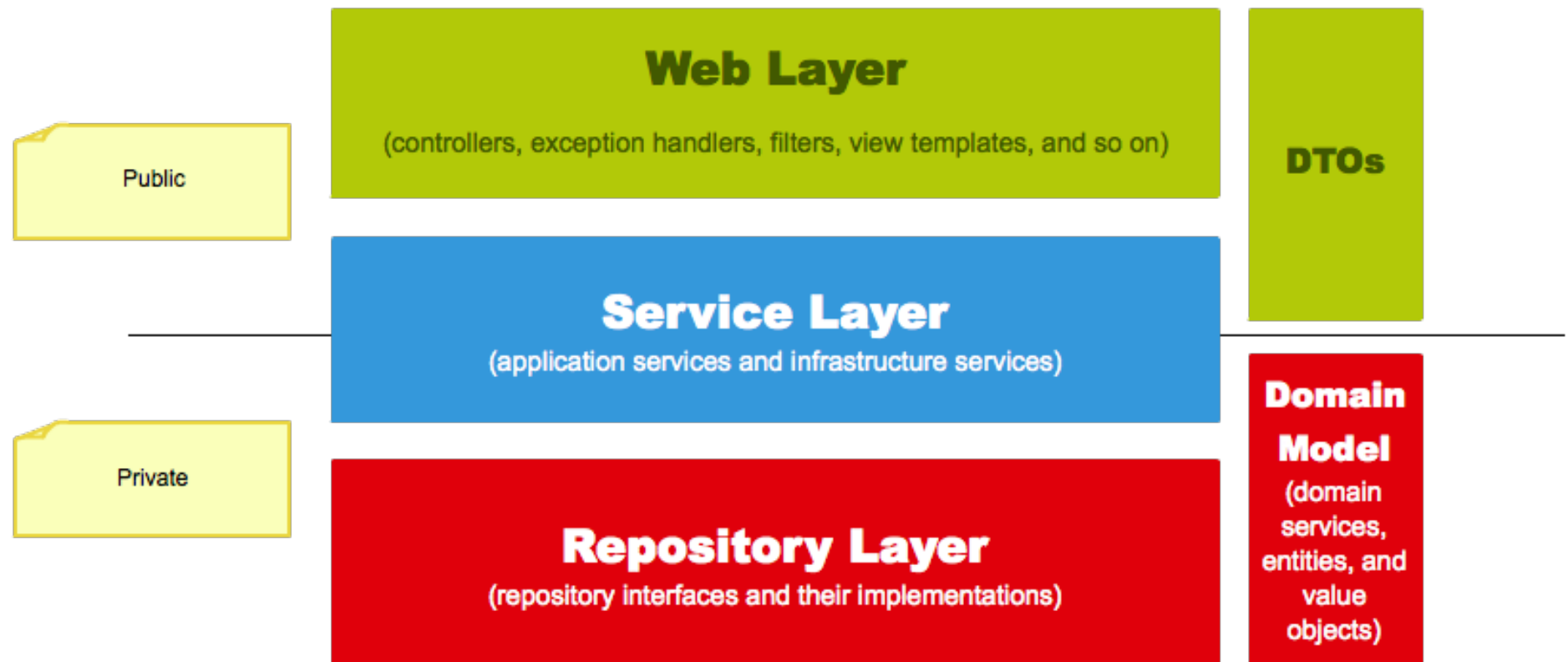
Rappels Spring MVC

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*

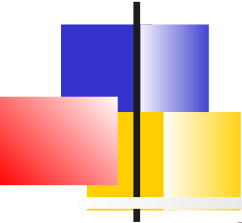
```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @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



@RestController

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

Ces contrôleurs ne produisent que des réponses au format JSON



Exemple pour des données JSON

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

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

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

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

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

        member = memberRepository.save(member);

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



Sérialisation JSON

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

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

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

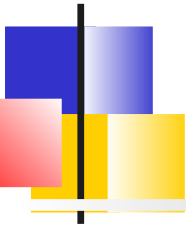


Alternatives Jackson à la sérialisation

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

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

- Créer des classes DTO spécifiques
- Utiliser les annotations de champs proposées par Jackson
- Utiliser les JsonView sur son modèle
- Implémenter ses propres *ObjectMapper*



Annotations de base Jackson

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

@JsonRootName : Arbre JSON

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

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

....



JsonView

```
public class CompanyViews {  
    public static class Normal{};  
    public static class Manager extends Normal{};  
    public static class HR extends Normal{};  
}  
..  
public class Staff {  
  
    @JsonView(CompanyViews.Normal.class)  
    private String name;  
  
    @JsonView(CompanyViews.Normal.class)  
    private int age;  
  
    // two views  
    @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);
```



Personnalisation de la configuration

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



Exemple Cross-origin

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

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

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



Gestion des erreurs

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

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

Pour remplacer le comportement par défaut :

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



Exemple

@ResponseStatus(value = HttpStatus.NOT_FOUND)

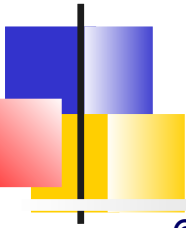
```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String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



ResponseStatusException

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

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



Exemple *@ControllerAdvice*

@ControllerAdvice

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

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

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

@Override

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



Appels de service REST

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

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



Exemple

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

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

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



SpringDoc

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

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

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

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




Fonctionnalités

Par défaut,

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

SpringDoc prend en compte

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

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



Projets Spring

Spring Data
Spring MVC
Spring Web Flux
Spring Security
Spring Test



Programmation réactive

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

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

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



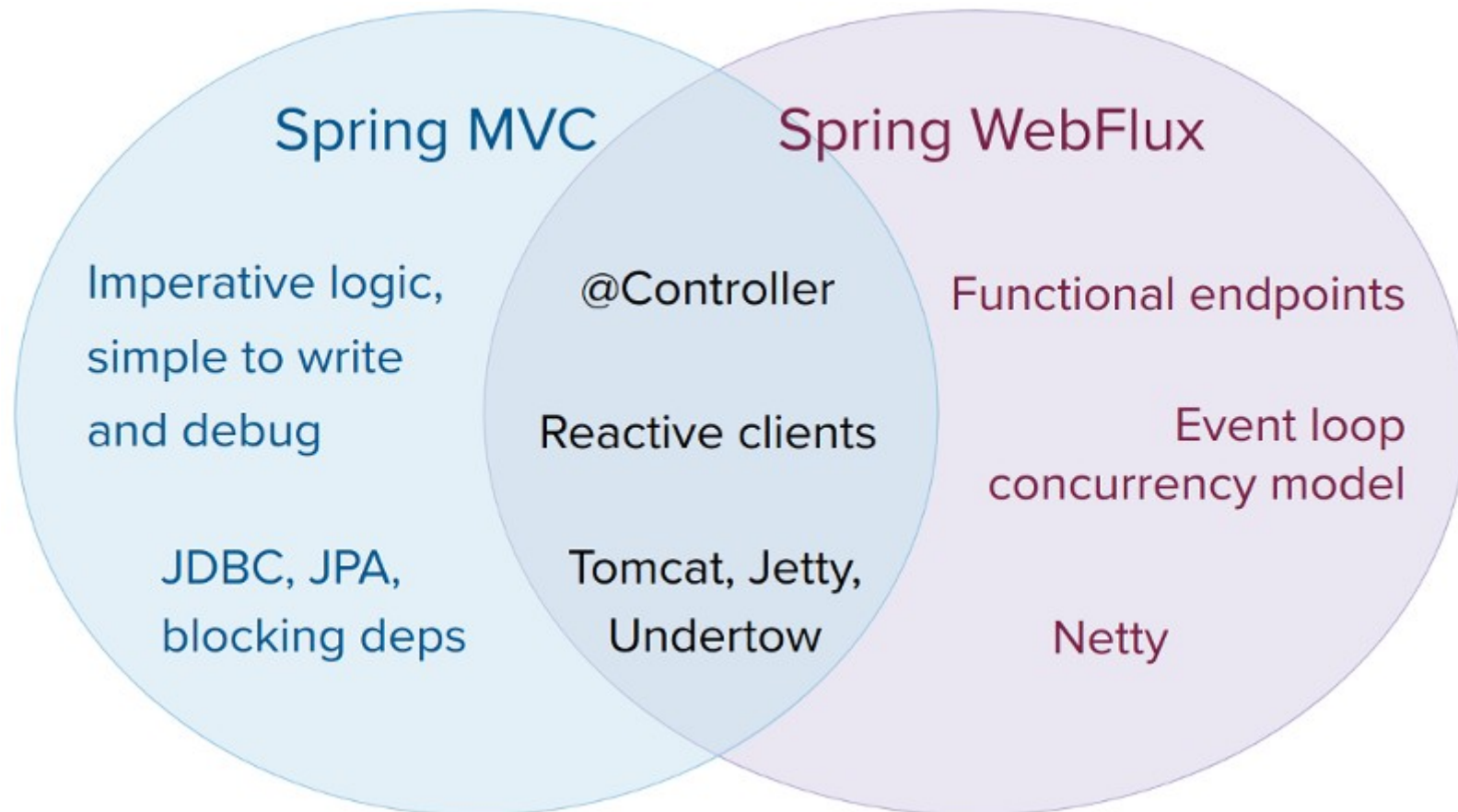
Motivation *Webflux*

2 principales motivations pour Spring Webflux :

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



MVC et WebFlux





Introduction

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

Il offre en plus 2 modèles de programmation :

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



Contrôleurs annotés

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

Les différences sont :

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



Example

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




Méthodes des contrôleurs

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

– Arguments :

- ***ServerWebExchange*** : Encapsule, requête, réponse, session, attributs
- ***ServerHttpRequest*** et ***ServerHttpResponse*** : Requêtes et réponses Http réactives

– Valeurs de retour :

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

– @RequestMapping (consume/produce) : *text/event-stream*



Endpoints fonctionnels

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

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



Traitement des requêtes via *HandlerFunction*

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

Exemple :

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

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



Example

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



Mapping via *RouterFunction*

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

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

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

Exemple :

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



Combinaison

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

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

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

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



Example

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

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

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

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

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

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

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

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

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



Configuration WebFlux

@Configuration

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




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();
        }
    });
```



Projets Spring

Spring Data
Spring MVC
Spring Web Flux
Spring Security
Spring Test



Rappels *Spring Security*

Spring Security gère principalement 2 domaines de la sécurité :

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

Spring Security facilite la mise en place de la sécurité en

- se basant sur des fournisseurs d'authentification :
 - Standards ((LDAP, OpenID, Kerberos, PAM, CAS, OAuth2)
 - Complètement spécialisés
- permettant la configuration des contraintes d'accès aux URLs et aux méthodes des services métier



Principe et mécanisme

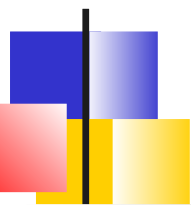
Sécurisation des URLs

La configuration crée alors une chaîne de filtres via le bean ***springSecurityFilterChain*** responsable de tous les aspects de la sécurité :

Protection des URLs, Redirection page de login, Extraction d'entêtes ou de jetons, ...

Sécurisation des services

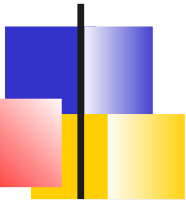
- Activation : ***@EnableGlobalMethodSecurity***
- Annotation des méthodes :
 - Par rôle : ***@Secured, @RolesAllowed***
 - Par Spel : ***@PreAuthorize, @PostAuthorize***
 - Filtrage des arguments : ***@PreFilter*** et ***@PostFilter***



Personnalisation *WebSecurity*

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

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



Exemple : Configuration du filtre

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



Exemple : Configuration authentication

```
@Configuration
```

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

```
@Override
```

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

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

```
}
```

```
}
```




Personnalisation via *UserDetailsService*

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

L'interface contient une seule méthode :

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

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



Exemple

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

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



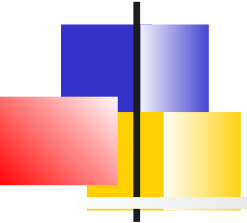
Password Encoder

Spring Security 5 nécessite que les mots de passes soient encodés

Il faut alors définir un bean de type
PasswordEncoder

L'implémentation recommandée est
BcryptPasswordEncoder

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



{noop}

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

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



Apports de *SpringBoot*

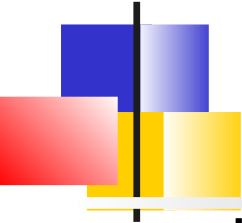
Si *Spring Security* est dans le classpath, la configuration par défaut :

- 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



Projets Spring

Spring Data
Spring MVC
Spring Web Flux
Spring Security
Spring Test



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 Mocks



@SpringBootTest

L'annotation **@SpringBootTest** permet de définir le contexte applicatif (*ApplicationContext*) utilisé lors des tests :

- Soit via l'attribut classes

Ex :

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

- Soit en remontant les packages jusqu'à ce qu'il trouve *@SpringApplication* (Dans ce cas, l'intégralité du contexte applicatif est disponible lors du test)



Attribut *WebEnvironment*

L'attribut *WebEnvironment* permet de préciser l'environnement web désiré :

- ***MOCK*** : Fournit un environnement de serveur mocké
On peut également utiliser :
@AutoConfigureMockMvc
- ***RANDOM_PORT*** : Le conteneur est démarré sur un port aléatoire
- ***DEFINED_PORT*** : Le conteneur est démarré sur un port spécifié
- ***NONE*** : Pas d'environnement web.



Mocking des beans

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

Cela permet de remplacer ou de créer de nouveaux beans

L'annotation peut être utilisée :

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

Les beans Mockito sont automatiquement réinitialisés après chaque test



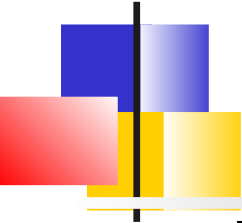
Exemple *MockBean*

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

    @MockBean
    private RemoteService remoteService;

    @Autowired
    private Reverser reverser;

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

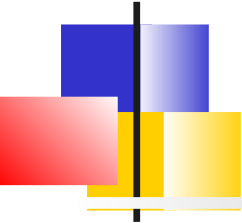


Tests auto-configurés

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 auto-configurés

@DataJpaTest configure une base de donnée mémoire, les `JpaRepository`, un `TestEntityManager` ...

@DataMongoTest : Configure une base mémoire Mongo, les `MongoDBRepository`. ...

@WebMvcTest limite le scan aux annotations de Spring MVC et configure *MockMvc*

@WebFluxTest : Test des contrôleurs Spring Webflux

@JsonTest configure automatiquement l'environnement *Jackson* ou *Gson* et des utilitaires *JacksonTester*, *GsonTester* ou *BasicJsonTester*

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



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

```
@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-security-test propose plusieurs annotations pour exécuter les tests d'une application sécurisée par SpringSecurity.

- ***@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